Automatically Converting Code-Writing Exercises to Variably-Scaffolded Parsons Problems

Logan Caraco
Nate Weinman
Stanley Ko
Armando Fox

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-173
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-173.html

June 27, 2022

Acknowledgement

This work was supported by a National Science Foundation Graduate Fellowship and by grant #OPR19186 from the California Education Learning Lab (calearninglab.org), a program of the California Governor's Office of Planning and Research.

Automatically Converting Code-Writing Exercises to Variably-Scaffolded Parsons Problems

LOGAN B. CARACO, NATHANIEL WEINMAN, STANLEY KO**, and ARMANDO FOX, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, USA

We present a system for automatically converting existing code-writing exercises in Python into Faded Parsons Problems (FPPs) that students solve interactively in a Web browser. FPPs were introduced by Weinman et al. [8] as a novel exercise interface for teaching programming patterns or idioms. Like original Parsons Problems [7], FPPs ask students to arrange lines of code to reconstruct a correct solution. Unlike original Parsons Problems, FPPs can also ask students to fill in some blanks in the provided lines of code, in addition to ordering the lines. In our system, which extends the open-source PrairieLearn platform, the student uses a Web browser to fill in blanks, reorder the lines of code, or both. The student can check their work at any time using an autograder that runs the student-submitted code against instructor-provided test cases; feedback to the student can be as fine-grained as the test cases allow. Converting existing code-writing exercises to FPPs is nearly automatic. Manually changing the amount of scaffolding in the FPP is easy and amenable to future automation. Instructors can thereby take advantage of initial study findings [9] that FPPs outperform code-writing and code-tracing exercises as a way of teaching programming patterns, and how FPPs improve overall code-writing ability at a level comparable to code-writing exercises but are preferred by students.

CCS Concepts: • **Applied computing → Learning management systems**; **Interactive learning environments**; • **Software and its engineering →** *Patterns*; *Domain specific languages*; *Markup languages.*

Additional Key Words and Phrases: PrairieLearn, Domain-Specific Languages, Educational Tools

1 INTRODUCTION AND BACKGROUND

Much effort has gone into creating, deploying, and evaluating interactive tools to help students develop code-writing skills **cite**, especially tools that allow for automated evaluation of the student's code ("code autograding"). As an alternative to writing code from scratch, **Parsons Problems** [7] ask students to arrange lines of code to reconstruct a correct solution. **Faded Parsons Problems (FPPs)**, recently introduced by Weinman et al. [9], may also require students to fill in some blanks in the provided lines of code, in addition to ordering the lines. The inventors of FPPs report that FPPs outperform code-writing and code-tracing exercises for teaching patterns and idioms, and they improve overall code-writing ability but are preferred by students to comparable code-writing exercises. Our goal is to make it easy for instructors to create autograded assessments based on FPPs.

As our vehicle for enabling instructors to create FPPs, we use the open-source PrairieLearn [1, 10], developed primarily at the University of Illinois at Urbana-Champaign and in use at several other campuses as well. PrairieLearn is an online problem-driven learning system for creating formative and summative assessments that supports autograding for code in addition to traditional question types such as selected-response. While many Learning Management Systems and code-autograding systems exist, we have found PrairieLearn to be the best-designed and most open-ended for authoring and administering novel, rich, interactive assessments. PrairieLearn can be extended with new types of

*The author was a summer research assistant at UC Berkeley while doing this work.

Authors' address: Logan B. Caraco, logan_caraco@berkeley.edu; Nathaniel Weinman, nweinman@berkeley.edu; Stanley Ko, stanley.ko@gmail.com; Armando Fox, fox@berkeley.edu, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 390 Soda Hall MS#1776, Berkeley, CA, USA, 94720-1776.

"elements" (building blocks) for new assessment types, making it easy for instructors to create assessments of that type without writing explicit code to render or grade them.

The rest of this paper describes our extensions to PrairieLearn for authoring and administering Faded Parsons Problems. The specific contributions are:

- A tool that automatically converts existing code-writing exercises with reference solutions and test cases into a Parsons Problem that students can solve using PrairieLearn
- An easy way to vary the amount of scaffolding in the generated Parsons Problem, thereby turning it into a Faded Parsons Problem
- A facility that allows the student submission to be graded according to whether instructor-provided tests pass (as opposed to the more simplistic approach of checking for a total or partial ordering of code lines in the problem), allowing for concise, arbitrary test granularity and scoring

Section 3.1 storyboards the student experience of solving a Faded Parsons Problem using our tool and explains what content the instructor needs to author. Section 3 describes how our tool works, including grading the student's work and giving the student feedback on incorrect answers, by both leveraging and extending existing PrairieLearn machinery. Section 4 report on our experience so far and ongoing work on additional tooling and pedagogy **todo: (especially in more-advanced CS courses)** that will exploit FPPs and the affordances of the tool described here.

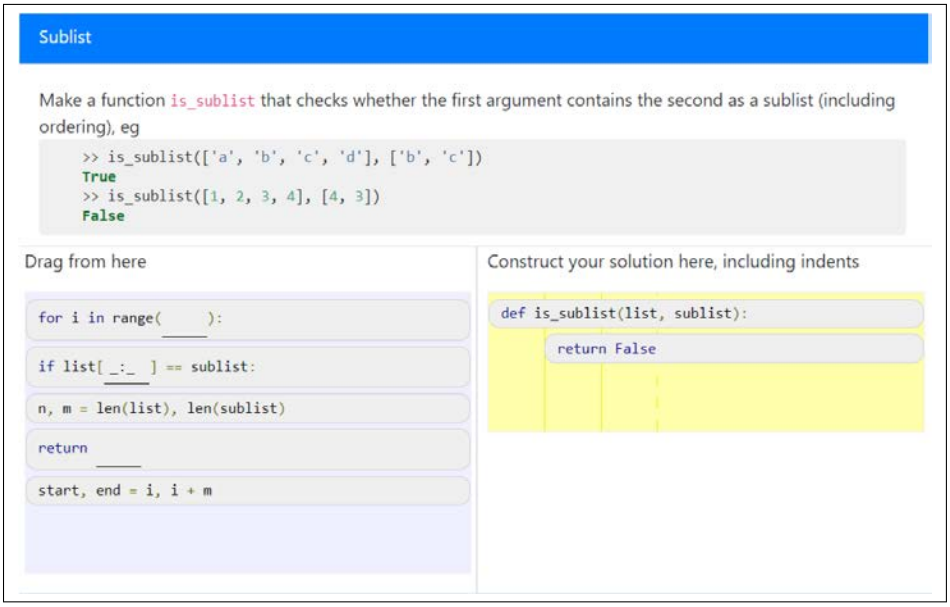

Fig. 1. Student view of the start of an fpp.
The initial state includes five lines in the code-line tray (left) and two lines of "starter code" pre-indented in the solution tray (right).

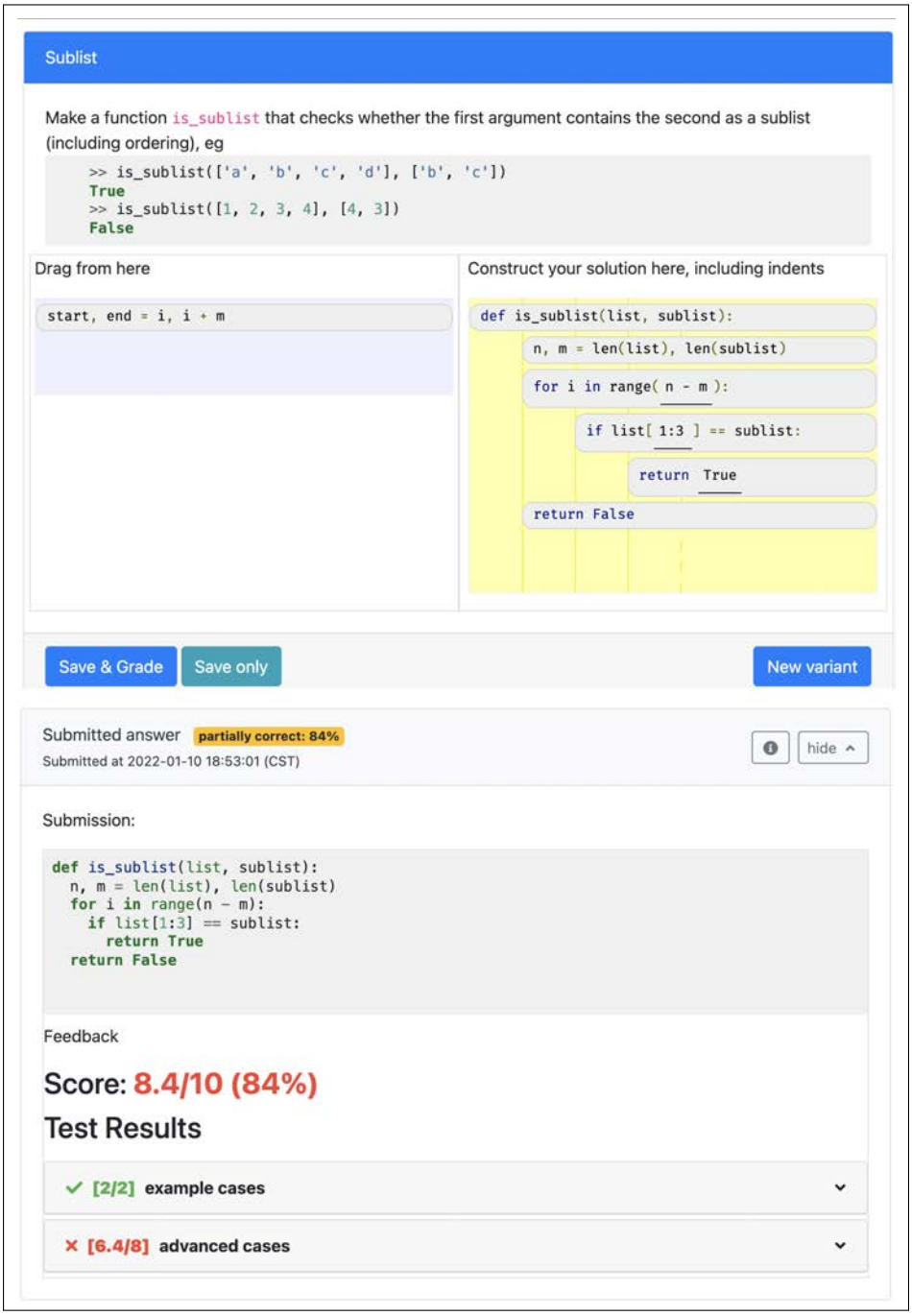

Fig. 2. Student view of a submitted, partially-correct solution
The student moved all but one line into the solution tray, and used magic constants for the missing pieces. This is why their solution passes only a subset of the tests, as the feedback indicates.

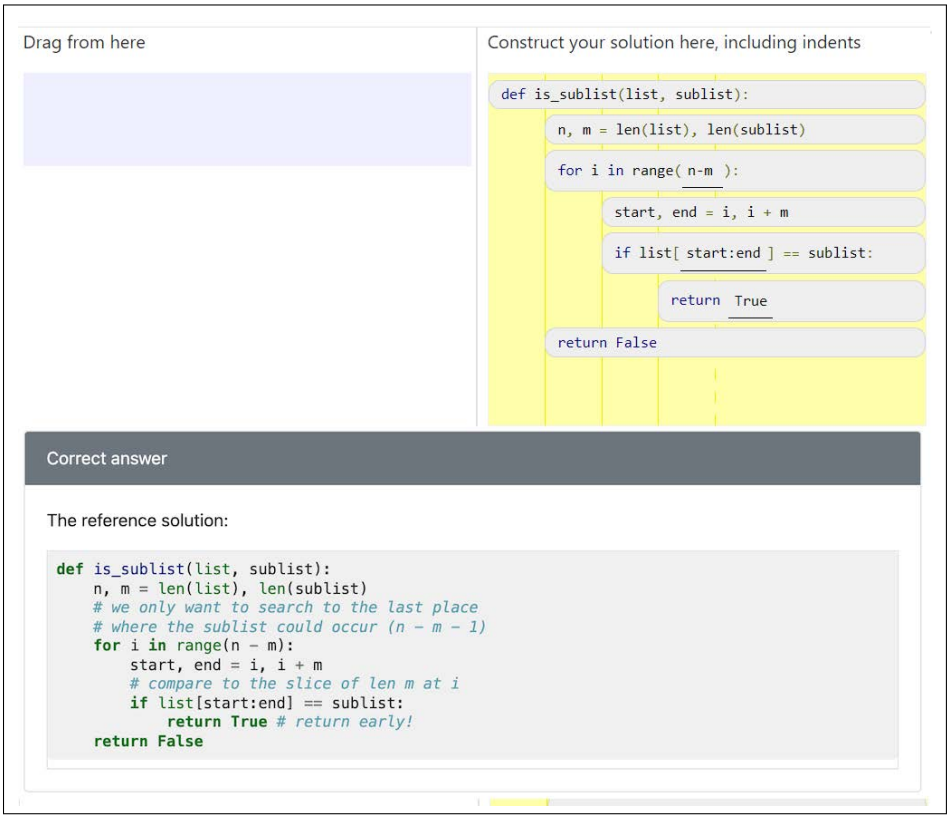

Fig. 3. Student view of a finished FPP.
A fully-correct subsequent submission by the student, with reference solution. See text regarding display of reference solution.

2 AUTHORING AND DELIVERING FADED PARSONS PROBLEMS

2.1 Student View: Solving FPPs In PrairieLearn

Figures 1,2,3 show a three-screenshot sequence illustrating a hypothetical student experience working on an FPP using our courseware.

Sally is a student in an introductory programming course. She is working on a FPP exercise about sublist detection. When she opens the problem, she sees a question prompt, along with a set of incomplete lines (Figure 1). The *code-line tray* on the left contains scrambled lines of code, some with blanks displayed as underlined text. The *solution tray* on the right begins with 2 lines given to Sally.

Looking at the lines of code, Sally realizes that she cannot solve this with a while loop like she originally planned, but must use for loops, which she has just started learning. She drags the for loop over, followed by a nested if statement, and then finally a nested return statement. She knows the return statement must return True, so fills in that blank. She is still not sure yet how to compute how many times to go through the for loop. Looking at the lines on the left, it clicks that the loop should iterate “n - m” times. She drags the line over defining “n” and “m”, remembering that Python allows you to initialize 2 variables on the same line, then fills in the blank in the for loop. Before finishing the problem,

Markup	In drag-and-drop boxes	When reference solution displayed
?*string*?	Displays a blank rather than *string*	Displays *string*
#blank *string*	Specifies initial text that appears in this line's blank	Not displayed
#*n*given	Includes this line as part of the starter code (solution) at language indent level n ($n = 0$ means no indent)	Not displayed
## *region delimiter* ##	Not displayed	Not displayed
Any comment not matching the above	Not displayed	Displayed

Table 1. How markup in the reference solution is converted to scaffolding when displaying the exercise. In the current implementation, at most one blank per line is allowed.

though, she wants to make sure she is on the right track. She hardcodes "[1:3]" into the if statement since she knows it should pass the examples in the instructions. She hits "Save & Grade" to see what feedback she will get.

Sally sees that her code is partially correct (Figure 2). She can see that she passed the two example test cases, as she had hoped. But, as expected, her solution is not quite complete yet. The state of her previously submitted solution (line ordering, indentation, content of blanks) is preserved. Confident that she is on the right track, she drags over the last line of code inside the for loop, and updates the "if" statement blank to use these new variables "start" and "end". She hits "Save & Grade" again.

Now that she has the correct answer, Sally can also see the instructor's reference solution (Figure **??**, test feedback cropped). She can read the comments in the reference solution to make sure she fully understands why the code works.

PrairieLearn allows each assessment to configure if and when to display the reference solution, the number of allowed response attempts, and the time limit per attempt.

2.2 Instructor View: Authoring FPPs

Consider Prof. Kipp, a hypothetical instructor. They have an existing code-writing problem about sublist detection that they would like to turn into an FPP such as that displayed in the screenshot. We assume Prof. Kipp begins with a reference solution (Figure 4) and one or more Python test cases that simply verify correct outputs for given sets of fixed inputs. To convert the function documentation into the question prompt for the problem, Prof. Kipp moves it to the first line of the file. The prompt is contained in a Python docstring that can include both HTML markup (such as `<code>` in the prompt text) and standard PrairieLearn XML markup such as `<pl-code>`, which causes a code excerpt to be displayed with language-aware syntax highlighting in PrairieLearn's client-side rendering. As is typical for Python, this example question prompt shows some expected outputs for specific input cases, though neither these nor any other content in the question prompt affect how the question is graded.

At this point the FPP can be previewed in PrairieLearn, so that Prof. Kipp can decide how to vary the scaffolding as we describe next, but it cannot be administered to students until test cases are added for grading. Prof. Kipp copies their originally-authored test cases from their short-form code-writing assignment into the main file, bookended with the *region delimiter* '`test`'. Figure 6 shows the the copied code (lines 23-62) in the context of the whole file, surrounded by `test` delimiters.

Prof. Kipp now uses PrairieLearn to preview the problem, and decides to vary the scaffolding. Table 1 shows the options available for doing so by annotatin the reference solution. In particular, Prof. Kipp decides to include the `def`

```python
def is_sublist(lst, sublist):
    """ A function that checks whether the first argument
    contains the second as a sublist (including ordering):
    >> is_sublist(['a', 'b', 'c', 'd'], ['b', 'c'])  # True
    >> is_sublist([1, 2, 3, 4], [4, 3])              # False
    """
    n = len(lst)
    m = len(sublist)
    # we only want to search to the last place
    # where the sublist could occur (n - m - 1)
    for i in range(n - m):
        start, end = i, i + m
        # compare to the slice of len m at i
        if lst[start:end] == sublist:
            return True # return early!
    return False
```

```python
from pl_helpers import name, points
from pl_unit_test import PLTestCase
from code_feedback import Feedback


class Test(PLTestCase):
    @points(2)
    @name("example cases")
    def test_0(self):
        is_sublist = self.st.is_sublist

        assert is_sublist(['a', 'b', 'c', 'd'], ['b', 'c'])
        Feedback.set_score(0.5)

        assert not is_sublist([1, 2, 3, 4], [4, 3])
        Feedback.set_score(1.0)

    @points(8)
    @name("advanced cases")
    def test_1(self):
        is_sublist = self.st.is_sublist
        big = [1, 2, 3, 4, 5, 6]

        assert is_sublist(big, big)       # equal     - T
        assert is_sublist(big, big[2:4])  # slice     - T
        Feedback.set_score(0.5)

        assert is_sublist(big, [1] * 6)   # unrelated - F
        assert is_sublist(big, big[1::2]) # skipping  - F
        assert is_sublist(big, big[::-1]) # reversing - F
        Feedback.set_score(1.0)
```

Fig. 4. A short-answer-style Python solution & tests.
The starting code Prof. Kipp begins to transition to an FPP and the PrairieLearn tests for it. Note that the point value for the test is set with the `points` decorator, and that scores reported to Feedback are always in $[0.0, 1.0]$

and `return False` lines as part of the solution's "starter code" as scaffolding. Adding `#0given` to the end of line 1 indicates that the `def` should appear in the correct position in the solution tray and not be indented. `#1given` at the end of line 16 indicates that `return` should appear in the correct position in the solution tray and should be indented once.

Finally, Prof. Kipp proceeds to fade some scaffolding, starting with blanking out the argument to `range` by surrounding the `n - m` with question marks, and similarly with the early-return value, replacing `True` with `?True?`. They also want to reinforce the concept of slices with this problem, making clear to the student that a slice argument is expected on line 14. By appending `#blank_:_` to the end of the line, the indexing brackets will contain a blank field initially populated with underscores around a colon, hinting that a slice argument is needed. Blanks and their default values if any can appear in lines in either the code-line tray or the solution tray.

```
""" Make a function <code>is_sublist</code> that checks
    whether the first argument contains the second as a
    sublist (including ordering), eg

    <pl-code language="python">
    >> is_sublist(['a', 'b', 'c', 'd'], ['b', 'c'])
    True
    >> is_sublist([1, 2, 3, 4], [4, 3])
    False
    </pl-code>
"""
```

```
def is_sublist(lst, sublist): #0given
    n, m = len(lst), len(sublist)
    # we only want to search to the last place
    # where the sublist could occur (n - m - 1)
    for i in range(?n - m?):
        start, end = i, i + m
        # compare to the slice of len m at i
        if lst[?start:end?] == sublist: #blank _:_
            return ?True? # return early!
    return False #1given
```

Fig. 5. Example question prompt (top) and reference solution (bottom).

As Table 1 describes, comments in the reference solution can indicate where to leave blanks in lines and which lines should be provided in their correct positions as "starter code." Combining the top and bottom with the code from Figure 6 creates a FPP complete with auto-grading.

One final run of the transpiler creates the finished form of the PrairieLearn FPP question, complete with blanks, autograding, starting text, starting solution lines, and a cleaned, fully-commented, reference solution available to the students after the problem is graded.

The next section describes how our transpiler generates the various files needed by PrairieLearn for presenting and autograding this problem from the contents of this single file.

3 RELEVANT IMPLEMENTATION DETAILS

A PrairieLearn *element* is a building block for authoring a certain type of question, such as multiple choice, fill in the blank, and so on. Each element type parses a set of well-defined XML that specifies the properties of a specific question of that type. To create a question, the instructor creates an HTML file that mixes standard HTML with element-specific XML. When the question is presented to a student, PrairieLearn delegates the rendering of each element's XML to element-specific code, resulting in a full JavaScript-enabled HTML page presented to the student. When the student presses the "Save and Grade" button to submit their response, element-specific server-side code is invoked to produce a numerical grade in $[0.0, 1.0]$ and optional feedback. PrairieLearn itself can also be configured to display the reference solution (or not) when the student either submits an acceptable answer or runs out of attempts.

As the narrative of Section 3.1 suggests, from the instructor's point of a view, a minimal but complete FPP requires a question prompt, a reference solution, and one or more test cases, and may optionally include annotations regarding scaffolding. To ease the FPP writing process, we created a simple Python-superset markup dialect abbreviated FPPD, for Faded Parsons Problem Dialect. Our transpiler parses instructor input in FPPD and generates the collection of files and markup chunks arranged as PrairieLearn expects. FPPD is designed to have three properties:

(1) *Centralized*: unify problem-specific data into just one file per problem that the instructor must author
(2) *Incremental*: generate a valid FPP from an existing Python solution, while allowing later annotations to adjust the scaffolding

```
## test ##
from pl_helpers import name, points
from pl_unit_test import PLTestCase
from code_feedback import Feedback


class Test(PLTestCase):
    @points(2)
    @name("example cases")
    def test_0(self):
        correct = 0
        ans, ref = self.st.is_sublist, self.ref.is_sublist

        lists = ['a', 'b', 'c', 'd'], ['b', 'c']
        if Feedback.call_user(ans, *lists) == ref(*lists):
            correct += 1

        lists = [1, 2, 3, 4], [4, 3]
        if Feedback.call_user(ans, *lists) == ref(*lists):
            correct += 1

        Feedback.set_score(correct / 2.0)

    @points(8)
    @name("advanced cases")
    def test_1(self):
        correct = 0
        ans, ref = self.st.is_sublist, self.ref.is_sublist
        big = [1, 2, 3, 4, 5, 6]
        tests = (
            big,          # equal     - True
            [1] * 6,      # unrelated - False
            big[2:4],     # slice     - True
            big[1::2],    # skipping  - False
            big[::-1]     # reversing - False
        )
        for small in tests:
            result = Feedback.call_user(ans, big, small)
            if result == ref(big, small):
                correct += 1
        Feedback.set_score(correct / len(tests))
## test ##
```

Fig. 6. Instructor-authored tests for the example.
A continuation of the file in Figure 5 — the scores reported to Feedback are normalized to sum to 1.0. Names are used to display the feedback, as in Figure 2.

(3) *Simple*: provide a simple syntax for common actions for translating a text problem into a FPP

3.1 Authoring: The Faded Parsons Problem Dialect

A PrairieLearn problem (as distinct from an element) is a subdirectory whose files contain structured question/answer data that must be organized in a specific way, as Figure 8 shows. `info.json` contains metadata for the question, such as its title, topic, and a unique ID, and also where the author specifies that a custom external grader (rather than PrairieLearn's default Python grader) is to be used for autograding. Our tool populates this file with scraped values and reasonable defaults for an FPP. `tests/ans.py` must contain a reference solution for possible display to the student; this file is easily constructed by extracting the reference solution from the FPPD file and stripping the scaffolding-related markup. Similarly, `question.html` must contain the displayable question prompt and initial question display (HTML plus element-specific XML), which is also generated by extracting from the reference solution and its documentation, as Figure 7 shows. `tests/test.py` and `tests/setup_code.py` are related to autograding and will be discussed in the next section.

```
<!-- AUTO-GENERATED FILE -->
<pl-question-panel>
  Make a function <code>is_sublist</code> that checks
    whether the first argument contains the second as a
    sublist (including ordering), eg

  <pl-code language="python">
  >> is_sublist(['a', 'b', 'c', 'd'], ['b', 'c'])
  True
  >> is_sublist([1, 2, 3, 4], [4, 3])
  False
  </pl-code>

 </pl-question-panel>

 <pl-faded-parsons>
   def is_sublist(list, sublist): #0given
      n, m = len(list), len(sublist) #1given
        for i in range(!BLANK):
            start, end = i, i + m
            if list[!BLANK] == sublist: #blank _:_
                return !BLANK
      return False #1given
 </pl-faded-parsons>
```

Fig. 7. HTML/XML generated by our transpiler.
PrairieLearn will inline this file into an overall HTML document template and expand the `<pl-code>` and `<pl-faded-parsons>` element tags.

Finally, `server.py` contains handlers invoked to render the question page and to handle student submission. In every Python problem, there may be given variables, functions, or classes. The `tests/setup_code.py` is where the definition for these bindings are provided, however PrairieLearn does not automatically supply these definitions to the student. The `server.py` file must detail the names of which bindings are allowed to pass from the setup code into the student submission and reference solution scopes, and in turn, which names are required to leave the solution and reference scopes. Using Python's `ast` package, this information is automatically extracted from the optional `setup_code` region and the FPPD solution code. Note that simply by prepending the setup code to the solution as a region, Python linters automatically detect given names and bindings as valid, whereas PrairieLearn's multiple-file, custom-import system would raise false errors.

For all problem-specific data that isn't explicitly required by PrairieLearn or the Python autograder, FPPD allows for arbitrary regions. The generator will convert unrecognized regions into files of the same name in the final output, eg adding the region named `data/census.json` generates the file `question/data/census.json`. This allows for problem data to be integrated directly into the FPPD file. Configured this way, regular Python linters will provide more accurate feedback on names in scope, or at the very least keep the structure of the data visible during problem authoring.

All of these generated source files are designed to be human-editable, but human intervention should never be necessary, as the dialect supports all available features for the element's body.

3.2 Rendering: Client-Side Code

The client-side code builds on the existing open-source project `js-parsons` [6], a JavaScript library implementing some of the basic functionality allowing drag-and-drop between the trays and adjusting indentation in the solution tray. Our JavaScript code is included by reference on any PrairieLearn-rendered HTML page that includes an FPP.

We added substantial functionality to parse the FPPD markup in Table 1 and create custom JQuery-UI "code-line" elements. During the rendering, input fields are created for all of the blanks, their initial values are populated, and the

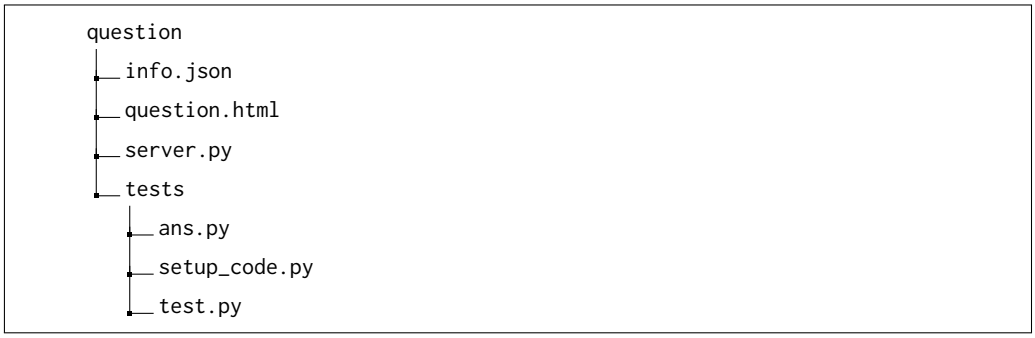

Fig. 8. The basic structure of a Python Problem or FPP
The default autograder for FPPs is the official Python autograder, which sets the dependencies for the Python files in the question.

code lines are rendered in either the problem tray or the solution tray. If this is the first attempt, these decisions are based on the annotations using the syntax of Table 1; on subsequent attempts, it is based on the data structure representing the FPP state that is preserved by the server across attempts and handed back to the client at page-rendering time. All code-line elements are decorated with language-specific syntax highlighting.

When the student submits work for grading, the text of each line in the solution tray is extracted, including embedding the text inside each input field, and each line is indented to match the visual indentation of the sortable tray. In addition, the student solution state–indents, line ordering, values of filled-in blanks—are parsed and stored in a separate server-side data structure that is maintained across attempts of the same problem, so that subsequent attempts can continue where the previous attempt left off. If the student fills in a blank in such a way as to cause a runtime error (for example, filling in something other than a valid argument to `range` in line 6), the student will see the errors generated by Python itself (in the case of entering a non-valid argument in `range`, the student would see a `TypeError`) and receive a score of 0 on that submission. If a student submits a solution with at least one empty blank, the JavaScript within the FPP element will detect this and warn the student *before* submission that they have an empty blank. The student can then decide whether they want to submit their solution or continue editing it without a grading penalty.

3.3 Grading: Autograding and Tests

The FPP element judges the correctness of student code based solely on whether it passes the instructor's test cases, *not* by checking for a total or partial ordering of the code-lines. For instance, in our simple example, lines 2–3 can be reordered or omitted in the solution without affecting correctness. A corollary of this decision is that *only* those behaviors tested by instructor-provided test cases are checked.

PrairieLearn provides facilities for autograding Python code, including a mechanism to install necessary libraries for the student code to work, running pre-suite and post-suite code, isolating the student code in a sandbox, disabling particular library calls (raising an exception if the student tries to use them), and providing a mechanism to give fine-grained feedback to the student.

In particular, PrairieLearn provides the Python class `Feedback` class for providing student feedback and the `PLTestCase` base class. The problem test suite must extend `PLTestCase` to define an entry point. Each test therein is written as a method decorated with `points` and `name`, and named `test_n` (e.g. Figure 6).

```
{
    "functionName": "is_sublist",
    "tests": [
        {
            "name": "example cases",
            "points": 2,
            "inputs": [
                "['a', 'b', 'c', 'd'], ['b', 'c']",
                "[1, 2, 3, 4], [4, 3]"
            ]
        },
        {
            "name": "advanced cases",
            "points": 2,
            "inputs": [
                "[1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6]",
                "[1, 2, 3, 4, 5, 6], [1] * 6",
                "[1, 2, 3, 4, 5, 6], [3, 4]",
                "[1, 2, 3, 4, 5, 6], [2, 4, 6]",
                "[1, 2, 3, 4, 5, 6], [6, 5, 4, 3, 2, 1]"
            ]
        }
    ]
}
```

Fig. 9. JSON form of test cases in Figure 6.

The JSON format is standard for PrairieLearn, and is used to generate equivalent tests as in Figure 6. Because JSON syntax is a subset of Python 3 dict literal syntax, this can be included as a part of a FPPD file within a 'test' region error-free.

While each test case can be arbitrarily complex, a common approach is to check if the student code produces the same outputs for a given set of inputs as the reference solution would; specifically, that the reference solution consists of a single exception-free function that takes literal or constructable arguments and returns a comparable value. For this common case, if our tool detects the presence of well-formed JSON in a region of the FPPD file delimited by `test`, it will automatically generate a compliant `test/test.py` containing those test cases as a subclass of `PLTestCase`. Score is calculated as a proportion of equal results running the tests against the reference solution and running the same tests over the student submission. As a consequence, correct *output* is not part of the JSON specification, only inputs and point value for a set of inputs.

For more complex reference solutions or test cases, the tool generates a human-editable starting file to which test cases and/or setup code can be added manually. The `Feedback` class exposes many static methods for checking submission correctness and returning scores and textual feedback. As an easy interface to this class, we often utilize our `score_cases` helper function that compares a submission to the reference solution over a set of inputs, normalizes the total score to $[0, 1]$, and reports the score to PrairieLear. All generated test files from our tool automatically include the `score_cases` function for convenient extension to the suite.

4 DISCUSSION, LIMITATIONS, RELATED AND ONGOING WORK

FPPs are already in regular use in one of our rigorous but entry-level CS courses, and we plan to explore using them to teach more advanced programming concepts in upper-division courses, since the original FPP study [9] suggests that FPPs are useful for teaching and exposing students to more sophisticated programming patterns.

4.1 Related Work

Parsons Problems [7] are similar to blocks-based programming environments (e.g. Scratch, Snap!, Alice, Blockly) in that they eliminate certain types of syntax errors beginning students might otherwise make, thereby reducing

cognitive load and allowing the student to focus on the conceptual challenges of solving a programming problem. But Parsons Problems support the full capabilities of textual programming languages, rather than introducing a new language. Parsons Problems have been implemented in Runestone [5], a platform hosting for hosting open-source ebooks. PrairieLearn differs in its focus on delivering personalized formative and summative assessments.

Zavala et al. [11] found support that code comprehension (e.g., code tracing exercises), code manipulation (e.g., Parsons Problems), and code writing are separate skills that should be mastered in that order, suggesting students would benefit from more opportunities for code manipulation exercises. Cunningham et al. [2] found that progressing from Parsons Problems to Faded Parsons Problems to code writing exercises was a productive progression for conversational programmers.

Ericson [4] found that Parsons Problems combined with worked examples provide similar learning gains to code-writing or bug detection in a CS1 classroom while taking only 70% of the time, and later found [3] similar results for Adaptive Parsons Problems, which dynamically adjust difficulty by adding or removing unnecessary or incorrect lines of code ("distractors"). FPP offer a different way to adjust the difficulty of problems by selecting what and how much code is removed.

4.2 Future Work

Only two aspects of our tool are Python-specific. The first is the use of Python's `ast` package to manage the propagation of Python names between the given code, student-submitted code, reference solution, and the testing scope. The second is the application of a Python-specific syntax highlighter in the client code to the draggable code lines seen by the student. We are adjusting these dependencies to make it easy to add other interpreted languages.

For formative assessments, we would like not only to see each submission attempt from the student, but also to potentially log each drag-and-drop or fill-in-the-blanks interaction that occurs between submission attempts. Adding such logging would allow detection of whether students are "driving in circles," attempting brute force solutions, or getting stuck in more subtle ruts.

Another feature of interest is an *auto-fader*. Automatic fading would ideally be used to detect or suggest expressions to fade in Python code, automatically creating code such as that in Figure 5 from an unfaded reference solution such as Figure 4. A weaker version of the auto-fader that analyzes if a problem is suitable for fading, or even an FPP linter, would further speed the process of writing FPPs from scratch.

An excellent complement to an auto-fader would be a FPP analyzer which could report which concepts, patterns and tactics should be targeted by fading a problem. Paired with an auto-fader, an analyzer could help decide which parts of the scaffolding to fade as part of a program of either proficiency-based (mastery) learning or spiral learning **cite**. Such a tool could provide information on the relative importance of different blanks in terms of these skills. In turn, the relative importance could inform the tests instructors write.

5 CONCLUSION

Faded Parsons Problems (FPPs) have been shown to be a novel and effective type of exercise for exposing students to programming patterns and practicing code-writing, giving similar learning gains to code-writing exercises but preferred by students. Our extensions to the freely-available, open-source PrairieLearn system allow instructors to easily convert a question prompt and reference solution into an auto-gradable FPP with varying amounts of scaffolding, promoting both proficiency-based and spiral learning.

ACKNOWLEDGMENTS

This work was supported by a National Science Foundation Graduate Fellowship and by grant #OPR19186 from the California Education Learning Lab (calearninglab.org), a program of the California Governor's Office of Planning and Research.

REFERENCES

[1] Zilles C, West M, Mussulman D, and Bretl T. 2018. Making testing less trying: Lessons learned from operating a computer-based testing facility. In *2018 Frontiers in Education Conference (FIE 2018)*. http://lagrange.mechse.illinois.edu/pubs/ZiWeMuBr2018/ZiWeMuBr2018.pdf

[2] Kathryn Cunningham, Barbara J. Ericson, Rahul Agrawal Bejarano, and Mark Guzdial. 2021. Avoiding the Turing Tarpit: Learning Conversational Programming by Starting from Code's Purpose. In *2021 ACM CHI Virtual Conference on Human Factors in Computing Systems (CHI 2021)*. Yokohama, Japan (online virtual conference).

[3] Barbara J. Ericson, James D. Foley, and Jochen Rick. 2018. Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ACM, New York, NY, USA, 60–68. https://doi.org/10.1145/3230977.3231000

[4] Barbara J. Ericson, Lauren E. Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research - Koli Calling '17*. ACM Press, New York, NY, USA, 20–29. https://doi.org/10.1145/3141880.3141895

[5] Barbara J. Ericson and Bradley N. Miller. 2020. *Runestone: A Platform for Free, On-Line, and Interactive Ebooks*. Association for Computing Machinery, New York, NY, USA, 1012–1018. https://doi.org/10.1145/3328778.3366950

[6] Petri Ihantola and Ville Karavirta. 2010. Open source widget for parson's puzzles. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education - ITiCSE '10*. ACM Press, New York, NY, USA, 302. https://doi.org/10.1145/1822090.1822178

[7] Dale Parsons and Patricia Haden. 2006. Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52* (Hobart, Australia) *(ACE '06)*. Australian Computer Society, Inc., AUS, 157–163.

[8] Nathaniel Weinman, Armando Fox, and Marti Hearst. 2020. Exploring Challenging Variations of Parsons Problems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, USA, 1349. https://doi.org/10.1145/3328778.3372639

[9] Nathaniel Weinman, Armando Fox, and Marti A. Hearst. 2021. Improving Instruction of Programming Patterns With Faded Parsons Problems. In *2021 ACM CHI Virtual Conference on Human Factors in Computing Systems (CHI 2021)*. Yokohama, Japan (online virtual conference).

[10] Matthew West, Geoffrey L. Herman, and Craig Zilles. 2015. PrairieLearn: 2015 122nd ASEE Annual Conference and Exposition. *ASEE Annual Conference and Exposition, Conference Proceedings* 122nd ASEE Annual Conference and Exposition: Making Value for Society, 122nd ASEE Annual Conference and Exposition: Making Value for... (2015). http://www.scopus.com/inward/record.url?scp=84941994154&partnerID=8YFLogxK

[11] Laura Zavala and Benito Mendoza. 2017. Precursor Skills to Writing Code. *J. Comput. Sci. Coll.* 32, 3 (Jan. 2017), 149–156.