PyDex: Repairing Bugs in Introductory Python Assignments using LLMs

JIALU ZHANG*, University of Waterloo, Canada
JOSÉ PABLO CAMBRONERO†, Microsoft, USA
SUMIT GULWANI†, Microsoft, USA
VU LE†, Microsoft, USA
RUZICA PISKAC†, Yale University, USA
GUSTAVO SOARES†, Microsoft, USA
GUST VERBRUGGEN†, Microsoft, Belgium

Students often make mistakes in their introductory programming assignments as part of their learning process. Unfortunately, providing custom repairs for these mistakes can require a substantial amount of time and effort from class instructors. Automated program repair (APR) techniques can be used to synthesize such fixes. Prior work has explored the use of symbolic and neural techniques for APR in the education domain. Both types of approaches require either substantial engineering efforts or large amounts of data and training. We propose to use a large language model trained on code, such as Codex (a version of GPT), to build an APR system – PyDex – for introductory Python programming assignments. Our system can fix *both* syntactic and semantic mistakes by combining multi-modal prompts, iterative querying, test-case-based selection of few-shots, and program chunking. We evaluate PyDex on 286 real student programs and compare to three baselines, including one that combines a state-of-the-art Python syntax repair engine, BIFI, and a state-of-the-art Python semantic repair engine for student assignments, Refactory. We find that PyDex can fix more programs and produce smaller patches on average.

CCS Concepts: • **Social and professional topics** → **Computer science education**; • **Computing methodologies** → **Artificial intelligence**.

Additional Key Words and Phrases: AI for programming education, large language models, automated program repair

ACM Reference Format:
Jialu Zhang, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2024. PyDex: Repairing Bugs in Introductory Python Assignments using LLMs. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 133 (April 2024), 25 pages. https://doi.org/10.1145/3649850

*Substantial portion of work was performed when Jialu Zhang was an intern at Microsoft.
†Ordered alphabetically.

Authors' addresses: Jialu Zhang, University of Waterloo, Waterloo, Canada, jialu.zhang@uwaterloo.ca; José Pablo Cambronero, Microsoft, Redmond, USA, jcambronero@microsoft.com; Sumit Gulwani, Microsoft, Redmond, USA, sumitg@microsoft.com; Vu Le, Microsoft, Redmond, USA, levu@microsoft.com; Ruzica Piskac, Yale University, New Haven, USA, ruzica.piskac@yale.edu; Gustavo Soares, Microsoft, Redmond, USA, gsoares@microsoft.com; Gust Verbruggen, Microsoft, Keerbergen, Belgium, gverbruggen@microsoft.com.

ACM 2475-1421/2024/4-ART133
https://doi.org/10.1145/3649850

1 INTRODUCTION

Programming education has grown substantially in popularity in the past decade [Singer 2019]. A key challenge associated with this growth is the need to provide novice students with effective and efficient learning support. In an ideal world, teaching assistants would monitor students' learning process, and when students' code is not correct, they would then help them to derive a correct solution. However, this approach does not scale and educational institutions struggle to find teaching assistants. As a result, there is an interest in developing automated tools that students can use for feedback instead. These tools provide custom repairs for their programming mistakes. The field of automated program repair (APR), which has a long history in the software engineering community [Ahmed et al. 2022a; Le Goues et al. 2012, 2019; Long et al. 2017; Long and Rinard 2016; Mechtaev et al. 2016], has introduced different approaches [Gulwani et al. 2018; Hu et al. 2019; Pu et al. 2016; Rolim et al. 2017] to produce such automated repairs for student mistakes in introductory assignments. Given a buggy student program, the APR system aims to produce a patch that satisfies a specification (typically the instructor-provided test cases). The patch must also minimize the number of changes made, with the goal of facilitating student learning [Hu et al. 2019].

Prior automated program repair systems for student programming assignments have generally been implemented using purely symbolic [Gulwani et al. 2018; Hu et al. 2019; Rolim et al. 2017; Wang et al. 2018a] or purely neural [Ahmed et al. 2018; Pu et al. 2016] techniques. Symbolic approaches require substantial engineering efforts to develop, typically requiring significant program analysis/repair experience, as well as custom repair strategies tailored to the language domain in which students implement their assignments. Neural approaches mitigate some of the engineering challenges but typically require substantial amounts of data, often leading to specialized use cases for Massive Open Online Courses (MOOCs). Furthermore, these systems are typically tailored to focus exclusively on syntax repair or exclusively on semantic repair. For the latter, the assumption is the code to be repaired contains no syntactic errors.

In this paper, we introduce `PyDex`, a **Py**thon repair tool built on top of Co**dex**, a version of the popular LLM GPT-3 [Brown et al. 2020b] that was further trained on code. `PyDex` is a *unified* syntactic and semantic repair engine for introductory Python programming assignments. Using a large language model trained on code (LLMC) removes the need for custom symbolic repair logic or retraining of a new neural model, and it allows us to handle *both* syntactic and semantic mistakes. While LLMCs have been successfully applied to tasks such as code generation [cop 2024], their impact in the education domain remains controversial [Berger 2022]. Using an LLMC for repair provides an opportunity to produce a positive impact in this domain.

We follow the approach of recent work [Joshi et al. 2022; Xia and Zhang 2022] in framing program repair as a code generation task that can be tackled with an LLMC. However, using LLMCs to produce student repairs requires addressing three challenges. First, the system must be able to handle multi-modality: the instructor may provide test cases, a description of the task in natural language, and language tooling (e.g. a compiler) may provide further information. Second, APR patches in the education domain need to reduce the number of changes to support learning – this requires that we limit the extent to which the LLM can generate more code than necessary or make changes to parts of the program that are not incorrect. Third, incorporating the LLMC as a core (but black box) component in our design requires that we adapt traditional prompt engineering techniques to our setting.

`PyDex` *ensembles* multi-modal prompts to generate complementary repair candidates. It employs prompts in an iterative querying strategy that first uses syntax-targeted prompts and *then* semantics-targeted prompts. To reduce the number of changes induced by syntax errors that should have

relatively simple fixes, PyDex uses the program's structure to extract a subprogram to give as input to the LLMC. By reducing the code surface exposed to the LLMC, PyDex biases repairs towards fewer edits. When fixing semantics, PyDex takes inspiration from existing symbolic repair literature [Gulwani et al. 2018; Ke et al. 2015; Wang et al. 2018a] and leverages few-shot learning, which adds task-related examples to the prompt, by retrieving other students' programs that have similar mistakes (and eventual corrections). To identify these programs, PyDex computes a similarity metric over test-suite outcomes.

We evaluated PyDex on student programs from an introductory Python programming course at a major university in India. Our evaluation has 15 programming tasks, totalling 286 student programs. These student programs contain both syntactic and semantic mistakes. As there is currently no tool that can solve both errors simultaneously, we compare PyDex to three baselines built by composing: BIFI [Yasunaga and Liang 2021], a state-of-the-art syntax repair tool for Python; Refactory [Hu et al. 2019], a state-of-the-art semantic repair tool for education Python programs; and GenProg, a canonical semantic repair tool based on genetic programming. Specifically, we compare PyDex to BIFI+Refactory, PyDex+Refactory, and PyDex+GenProg, where for the latter two baselines we use PyDex to produce syntactic fixes before applying the corresponding semantic repair tool.

Our results show that PyDex can effectively repair student programs in our benchmark set. PyDex without few-shot learning can repair 86.71% of the student programs. This repair rate climbs to 96.5% with few-shots. Meanwhile, BIFI+Refactory, PyDex+Refactory, and PyDex+GenProg repair 67.13%, 83.57%, and 49.30%, respectively. Our statistical analysis shows that the improvement over BIFI+Refactory and PyDex+GenProg is statistically significant.

The average token edit distance associated with PyDex patches is smaller (28.59 without few-shots and 29.68 with few-shots) compared to the patches produced by the baselines BIFI+Refactory (70.39) and PyDex+Refactory (73.53). We found that PyDex+GenProg (22.82) produces slightly smaller patches, but the difference is not statistically significant. Our statistical analysis shows that the improvement over BIFI+Refactory and PyDex+Refactory is statistically significant.

We carried out an ablation study to understand the impact of our design decisions. Our results indicate that by performing iterative querying the repair rate rises from 82.87% to 86.71%. Furthermore, adding few-shots raises the repair success rate to 96.5%. The evaluation also shows that our techniques are important for maintaining the repaired program similar to the buggy input program. For example, removing the program chunker, which selects subprograms in the syntax repair phase, raises the average token edit distance from 5.46 to 9.38 in the syntax phase. We also show that different multi-modal prompts have varying performance, but if we combine their candidates as we do in PyDex, we obtain the best performance.

To summarize, we make the following contributions:

- We propose an approach to automatically repair mistakes in students' Python programming assignments using a large language model trained on code (LLMC). Our approach uses multimodal prompts, iterative querying, test-case-based few-shot selection, and structure-based program chunking to repair student mistakes. In contrast to prior work, our approach uses the *same* underlying LLMC to repair *both* syntactic and semantic mistakes.
- We implement this approach in PyDex, which uses OpenAI's popular Codex as the LLMC. We evaluate PyDex on a dataset of 286 real student Python programs drawn from an introductory Python programming course in India. We compare performance to three baselines, that leverage popular repair systems such as BIFI, Refactory, and GenProg. Our results show that PyDex yields a statistically significant higher repair rate than 2 of our 3 baselines, and a statistically significant smaller average token edit distance (i.e. smaller patches) than 2 of our 3 baselines.

The remainder of the paper is structured as follows. Section 2 walks through multiple examples of real student mistakes, as well as associated PyDex patches. Section 3 provides a brief background on concepts related to large language models. Section 4 describes our approach in detail. Section 5 provides experimental results on our dataset of student Python programs. Section 6 details further discussions and limitations. We discuss related work in Section 7. Finally, we conclude with takeaways in Section 8.

2 MOTIVATING EXAMPLE

2.1 Understanding Challenges in Repairing Introductory-level Programs

Consider Figure 1, which shows a student's incorrect program, along with a solution generated by PyDex. The student is solving the task of reading two numbers from stdin and printing different results depending on whether both, either, or neither are prime.

The student has made both syntactic and semantic mistakes. Lines 1 and 2 call `input` twice to read from stdin, and parse these values as integers using **`int`**. However, this constitutes a semantic mistake, as the assignment input format consists of two values *on the same line* separated by a comma. Furthermore, a traditional semantic repair engine would fail to fix this student's assignment as there is *also* a syntactic mistake at line 30. The student used a single = for comparison in the `elif` clause (the correct syntax would be a double equals).

The PyDex solution, shown alongside it, fixes the input processing (semantic mistake) by reading from stdin, splitting on the comma, and applying **`int`** (to parse as integer) using the `map` combinator. Line 23 fixes the syntax error by replacing single equals with double equals (for comparison). Interestingly, the underlying LLMC (Codex) *also* refactored the student's program. In this case, lines 8 through 17 correspond to a function to check if a number is prime. This function is called twice, at lines 18 and 19. This replaces the repeated code in the original program, which spanned lines 9-17 and lines 18-26.

The edit distance between the PyDex repair and the original student program is 95, while the distance between the instructor's reference solution and the original student program is 188. A smaller edit distance is a key goal for APR in the educational domain, as this can help the student understand the repair with respect to their own mistakes.

Figure 2 presents another example of an incorrect student program and a solution generated by PyDex. In this assignment, the students need to check whether a string, read from stdin, is a palindrome or not, and print out a message accordingly to stdout. For this student's program, PyDex has to generate a complex repair that fixes four syntax mistakes and multiple semantic bugs.

The student has made syntax errors on lines 4, 8, 10, and 12, where they have left off the colon symbol necessary for control flow statements in Python. On line 2, the student called a non-existent function `lower`. The student has used standard division on lines 5, 6, 13, and 14 when they should have used integer division. The student has included two spurious print statements, at lines 7 and 15, which will interfere with the instructor's test-suite execution, as the suite checks values printed to stdout for correctness. Finally, the student has omitted the expected print statements (along with the equality check) for the case where the input string is of even length.

While the student's program has many mistakes, the overall structure and key concepts are there. Looking at the PyDex solution shown alongside, it resolves these mistakes but preserves the student's overall structure. In particular, PyDex replaces the non-existent `lower` function with a call to the string method with the same name. It replaces the division operator (/) throughout the program with the intended floor division operator (//), comments out the extra print statements, and adds the missing equality check and print statements in the case of even-length inputs.

```
a=int(input())
b=int(input())
n=str(a)
m=str(b)
n1=n[::-1]
m1=m[::-1]
x=int(n1)
y=int(m1)
if x>1:
  for i in range(2, x):
    if (x%i)==0:
      p=0
        break
        else:
          p=1
else:
  p=0
if y>1:
  for j in range(2, y):
    if (y%j)==0:
      q=0
        break
      else:
        q=1
else:
  q=0
if p==1 and q==1:
  g=x+y
  print(g)
elif p=1 or q=1:
  h=a+b
  print(h)
else:
  print(a*b)
```

(a) Incorrect Program

```
a,b = map(int, input().split(','))
n=str(a)
m=str(b)
n1=n[::-1]
m1=m[::-1]
x=int(n1)
y=int(m1)
def prime(z):
  if z>1:
    for i in range(2,z):
      if (z%i)==0:
        return 0
          break
      else:
        return 1
    else:
      return 0
p=prime(x)
q=prime(y)
if p==1 and q==1:
  g=x+y
  print(g)
elif p==1 or q==1:
  h=a+b
  print(h)
else:
  print(a*b)
```

(b) PyDex's Repair

Fig. 1. A student's submission contains both syntactic and semantic mistakes (red). PyDex's fixes (blue) the original issues and also refactors part of the student's code into a function (lines 8 - 17 in (b)) that avoids code duplication (lines 9-17, 18-26 in (a)).

The edit distance between the PyDex repair and the original student program is 52, while the distance between the instructor's reference solution and the original student program is 97. The reference solution is a standard one-line program for palindrome. Once again, the PyDex repair is closer to the student submission than the instructor's reference solution.

2.2 Insights

Based on our observation of errors in student's introductory-level programs, we extract the following insights guiding our solution design. First, incorrect introductory-level programs often contain both syntactic and semantics errors at the same time, and this is an extremely challenging

```
i = input()
S = lower(i)
l = len(S)
if(l%2!=0)
  B = S[:(l+1)/2]
  E = S[:(l+1)/2:-1]
  print(B,E)
  if(B==E)
    print(i,'is a palindrome.')
  else
    print(i,'is NOT a
        palindrome.')
else
  B = S[:l/2]
  E = S[:l/2:-1]
  print(B,E)
```

(a) Incorrect Program

```
i = input()
S = i.lower()
l = len(S)
if(l%2!=0):
  B = S[:(l+1)//2]
  E = S[:l//2-1:-1]
  #print(B,E)
  if(B==E):
    print(i,'is a palindrome.')
  else:
    print(i,'is NOT a
        palindrome.')
else:
  B = S[:l//2]
  E = S[:l//2-1:-1]
  #print(B,E)
  if(B==E):
    print(i,'is a palindrome.')
  else:
    print(i,'is NOT a palindrome.')
```

(b) PyDex's Repair

Fig. 2. A complex repair (blue) that fixes multiple syntactic and semantic mistakes (red). The repair produced by PyDex, which preserves the overall structure of the student's program, makes fewer changes to the student's program than a patch with respect to the instructor's (one-liner) reference solution.

scenario for existing APR tools to handle alone as they are tailored to focus exclusively on syntax repair [Rolim et al. 2017; Yasunaga and Liang 2021] or exclusively on semantic repair [Mechtaev et al. 2018, 2016]. While combining a state-of-the-art syntactic fixer and semantic fixer to repair programs is possible, we detail the (lower) performance and challenges in Section 5 and Section 6.1. Second, an introductory-level program can have many mistakes, which require complex repairs. Such cases are difficult to address by traditional existing APR techniques [Le Goues et al. 2012; Long and Rinard 2015; Mechtaev et al. 2018, 2016; Qi et al. 2014a; Xuan et al. 2017], as they often focus on specific error types, are limited to a small number of edits, and target specific types of statements (such as conditionals). For example, the repairs (e.g., control-flow changes, in-lined function addition) shown in this section are out-of-scope for traditional APR tools. Third, because the eventual consumer of the generated patches are introductory-level programmers, we should minimize the cognitive load associated with many changes where possible. Finally, because students themselves may want to run the repair tool (enabling them to learn independently), the engineering efforts associated with running the APR tool should be minimized as much as possible.

3 BACKGROUND

We now provide a short background on concepts related to large language models.

Large language model. A *large language model* (LLM) can be viewed as a probability distribution over sequences of words. This distribution is learned using a deep neural network with a large number of parameters. These networks are typically trained on large amounts of text (or code) with objectives such as predicting particular masked-out tokens or *autoregressive* objectives such as predicting the next token given the preceding tokens. When the LLM has been trained on significant

amounts of code, we refer to it as a large language model trained on code (LLMC). In practice, most LLMs are now trained on code as well, so the functional difference between the two categories has become increasingly less relevant.

Often, LLMs are pre-trained and then *fine-tuned*, meaning trained further on more specialized data or tasks. A particularly popular LLMC is OpenAI's Codex [Chen et al. 2021], a variant of GPT-3 [Brown et al. 2020a] that is fine-tuned on code from more than 50 million GitHub repositories.

Few- (or zero-)shot learning. In contrast to traditional supervised machine learning, LLMs have shown to be effective for *few-* and even *zero-*shot learning. This means that the LLM can perform tasks it was not explicitly trained for just by giving it a few examples of the task or even no examples, respectively, at inference time.

In this setting of few- (or zero-)shot learning, the LLM is typically employed using what is termed *prompt-based learning* [Liu et al. 2021]. A *prompt* is a textual template that can be given as input to the LLM to obtain a sequence of iteratively predicted next tokens, called a *generation*. A prompt typically consists of a query and possibly zero or more examples of the task, called shots. For example, the prompt below includes a specific query to fix a syntax error. One valid generation, that fixes the syntax error, would be `print()`.

```
# Fix the syntax error of the program #

# Buggy program #
print(
```

In practice, a prompt can incorporate anything that can be captured in textual format. In particular, *multi-modal* prompts are those that incorporate different *modalities* of inputs, such as natural language, code, and data.[1]

Different prompts may result in different LLM completions. Other factors may also affect the completions produced, such as the sampling strategy or hyperparameters for the sampling strategy. One important hyperparameter is *temperature*, which controls the extent to which we sample less likely completions.

LLM selection. While we use OpenAI's Codex in this work, other LLMs could be used such as Salesforce's CodeGen [Nijkamp et al. 2022] or OpenScience's BLOOM [Laurençon et al. [n. d.]]. Even within OpenAI's Codex there are different underlying models offered, including Codex-Edit [Open AI 2022]. We found performance to be better with the standard Codex completion model. We now leverage these concepts to describe our approach.

4 METHODOLOGY

Figure 3 provides an overview of `PyDex`'s architecture. The student's buggy program first enters a syntax repair phase. In this phase, we extract subprograms from the original program that have a syntax error. Each such subprogram is fed to a syntax prompt generator that produces multiple syntax-oriented prompts. The LLMC then generates repair candidates, which are validated by the syntax oracle. This process is repeated until all syntax errors are removed. Any candidate that has no syntax errors moves on to the semantic phase. In this phase, `PyDex` uses a semantic prompt generator to produce semantics-oriented prompts. If it has access to other student's assignment history, `PyDex` can also add few-shots to these prompts. These prompts are fed to the LLMC, which generates new program candidates. These are validated by the test-suite-based semantic oracle. If multiple candidates satisfy all tests, `PyDex` returns the one with the smallest token edit distance with respect to the student's original program. We now describe each step in detail.

[1]The term multi-modality, in the context of LLMs, is also used for combinations of image/text/audio. In our setting, all inputs are text but they come from different distributions, such as code versus natural language.

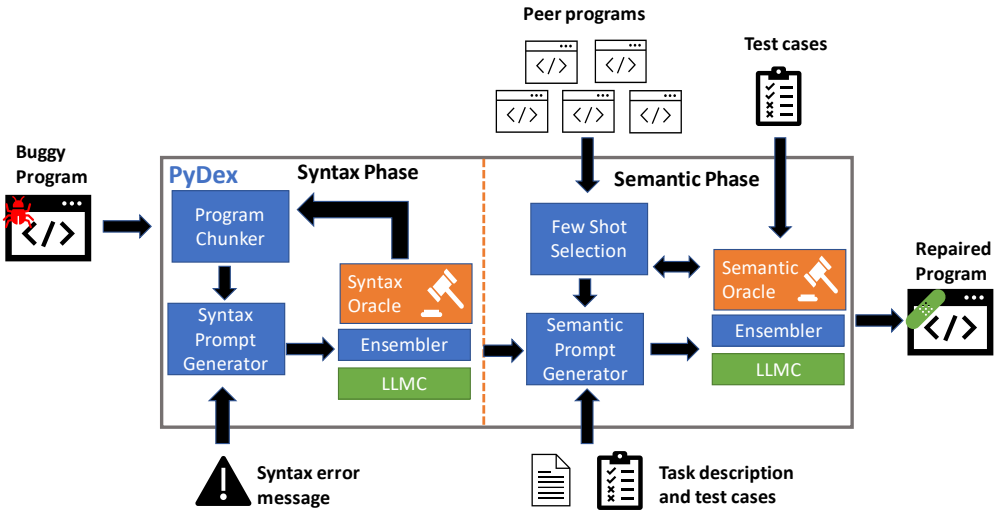

Fig. 3. PyDex architecture. A buggy program first enters a syntax repair phase. In this phase, PyDex transforms the program using a program chunker, which performs a structure-based subsetting of code lines to narrow the focus for the LLMC. Multiple syntax-oriented prompts are generated using this subprogram, fed to an LLMC, and any patches are integrated into the original program. If any candidate satisfies the syntax oracle, it can move on to the semantic phase. In the semantic phase, PyDex leverages both the natural language description of the assignment and the instructor-provided test cases to create various prompts. In addition, if available, PyDex can use other peers' solutions as few-shots by selecting them using test-case-based selection to identify failures that resemble the current student's program, along with eventually correct solutions. Prompts are fed to the LLMC to generate candidates. If multiple candidates satisfy the test suite, PyDex returns the one with the smallest edit distance with respect to the original student program.

4.1 Syntax Phase

Students typically first resolve syntax errors in their assignments, and then move on to resolve semantic errors (such as test case failures). PyDex takes inspiration from this approach and similarly splits its repair into syntax and semantic phases.

In the first phase, PyDex receives the student's buggy program. A syntax oracle, for example, the underlying Python parser[2], is used to determine if there is a syntactic mistake. If there is no such mistake, the program can move into the semantic phase. However, if there is a mistake, PyDex must produce a patch that resolves it, before moving to the semantic phase.

While our syntax prompt generator could directly include the original program in its entirety in the prompt, we have found that doing so can result in spurious edits that are not actually necessary to resolve the syntax error. Existing work has also observed similar phenomena in the related area of natural language to code generation [Poesia et al. 2022]. As a result, we introduced a component we call the *program chunker* to mitigate this challenge by reducing the amount of code included in the prompt.

[2]Some errors such as repeated function parameters throw a `SyntaxError` but are not checked until bytecode compilation

4.1.1 Program Chunking. For each syntax mistake in the original buggy program, the program chunker extracts a subset of lines that contains (1) the oracle-reported syntax error location and (2) the nearest encompassing control-flow statement. These *chunks* are a heuristic approximation of a basic block, and allow us to restrict the code input given to the LLMC. Note that we perform this heuristic approximation as a standard analysis to extract basic blocks typically requires a syntactically correct input program.

Algorithm 1 Chunker: extracting the code chunk that contains the error message

Input: *sC* : Program Source Code
Input: *msg* : Compiler Message
Output: *chunkedCode* : Chunked Program Source Code

```
1:  procedure CHUNKER(sC, msg)
2:      lineIndex, errorLine = locateError(sC, msg)
3:      errIndent = getIndentationLevel(sC, errorLine)
4:                                          ▷ move up until find a line with a smaller indentation level
5:      startIndex, startIndent = pointerUp(sC, lineIndex, errIndent)
6:                                  ▷ move down according to both indentation level and control-flow structure
7:      if sC[startIndex].startswith(cfKeyword) then          ▷ control-flow related keywords
8:          endIndex = pointerDown(sC, startIndex, startIndent, cfKeyword)
9:      else
10:         endIndex = pointerDown(sC, startIndex, startIndent)
11:     return chunkedCode = slice(startIndex, endIndex)
```

PyDex extracts the program chunk for the first (top-down) syntax error reported. Algorithm 1 outlines the procedure used to produce this program chunk. It takes advantage of both control-flow structure (based on Python keywords) and indentation, which are meaningful in the Python language. The program chunker first identifies the adjacent code that has the same or larger indentation level as the line with the syntax error. Then, if the code chunk contains control-flow related keywords, such as `if` and `elif`, PyDex makes sure the associated keywords (such as `elif` or `else`) for the same control flow statement are also in the chunk. This code chunk is then provided to the syntax prompt generator.

```
1  ...
2  if (condition1):
3      x = (0      # syntax error
4      x = 1
5  else
6    x = 2
7  if (condition2):
8      x = 3
9  ...
```

Fig. 4. An illustrated example of program chucking. Lines 3 and 4 have an indentation level of four, line 6 has an indentation level of two, and the rest of the lines have an indentation level of zero. Line 3 has the initial syntax error flagged by the interpreter. PyDex uses such indentation (along with control flow keywords) to heuristically extract program chunks for syntax repair.

For example, in Figure 4, the algorithm starts at line 3, setting the indentation level (`errIndent`) to 4. Subsequently, it moves up to traverse lower-indexed code lines, takes any line between

```
1  # Buggy Program #
2  while (n > 0):
3  a = n % 10
4  ...
```

(a) without error message

```
1  ### Error Msg ###
2  File "<unknown>",line 2
3    a = n % 10
4    ^
5  IndentationError: expected an
       indented block
6
7  # Buggy Program #
8  while (n > 0):
9  a = n % 10
10 ...
```

(b) with error message

Fig. 5. The syntax prompt generator produces prompts that can include the buggy program or the error message. We elide portions of the code fragments for brevity.

this error line and stops upon encountering the first line with an indentation level smaller than `errIndent`. The algorithm sets this as the starting line of the code chunk and then mark its indentation level as `startIndent`, which in this example is 0. At this starting line, if the line starts with a control-flow keyword (such as the `if` at line 2), the process moves down until reaching the first unmatched control-flow statement at an indentation level less than or equal to `startIndent`. Otherwise, if at the starting line, the code chunk does not start with a control-flow keyword, the algorithm simply moves down to higher-indexed code lines, including any consecutive line with an indentation level greater than or equal to `startIndent` until it finds a line with less indentation. In the provided example, the algorithm stops at line 7, resulting in a final code chunk spanning from line 2 to line 6. This example shows the algorithm's ability to selectively extract code chunk based on both indentation levels and control-flow structures, as depicted in Figure 4.

4.1.2 Syntax Prompt Generator. The syntax prompt generator produces two (multimodal) prompts, one with and one without the syntax error message reported by the syntax oracle. An example of both is shown in Figure 5. Because the syntax oracle is available, we do not need to choose a single prompt template for all programs, but instead we query the LLMC with both prompts, extract the code portion from each generation, merge it into the original program by replacing the lines corresponding to the current program chunk, and then rely on the syntax oracle to filter out invalid repairs.

If a program candidate has no syntax errors, it can move on to the semantic phase. If any syntax errors remain, the syntax phase is repeated. This iteration allows the repair of multiple, spatially-independent, syntax errors. For our evaluation, we allow this procedure to iterate at most two times to limit repair times.

4.2 Semantic Phase

After `PyDex` has generated syntactically valid candidate programs, the repair procedure moves to a semantic repair phase. Intuitively, this phase incorporates information that allows the LLMC to generate candidate programs that satisfy the programming assignment task, as determined by a semantic oracle. Following the approach of existing work in automated repair for programming assignments [Gulwani et al. 2018; Hu et al. 2019], we use the instructor's test suite (inputs and expected outputs) as the semantic oracle. We say a program is repaired if it produces the expected outputs for the given inputs.

```
[[Buggy Program]]
### Buggy Program ###
x=input()
y=int(x)
z = number % 10
y = 10 * y + z
number = number / 10
number = int(number)
print("Reverse:␣{}".format(x[::-1]))
print("Sum:␣{}".format(Sum))


[[Problem Description]]
#Write a program to read a number (int) from the user. Print the
    number in reverse. Also print the sum of the number and its
    reverse in a separate line. See the examples.
#NOTE: Do not print any prompt in the input().

[[Test Suite]]
#input:
43
#output:
Reverse: 34
Sum: 77

#input:
500
#output:
Reverse: 5
Sum: 505

### Correct Program ###
```

Fig. 6. An example multimodal prompt (in zero-shot setting for brevity) produced by the semantic prompt generator. This prompt includes code, natural language, and test cases. Lines starting with the double brackets are shown only for clarity, they are not part of the prompt itself.

4.2.1 Semantic Prompt Generator. The semantic prompt generator takes advantage of the rich set of signals available in the education domain. In particular, we exploit the fact that programming assignments typically have available: (1) a natural language description of the task, (2) a set of test cases, and (3) peers' programming solutions.

The semantic prompt generator takes as input a syntactically valid program, the task description in natural language, and the set of instructor-provided test cases. The generator then produces prompts with different combinations of this information. Figure 6 shows an example of such a multimodal prompt. This prompt includes the student's buggy code, the natural language description of the assignment, as well as the input-output-based test cases.

If PyDex has access to other student's assignment solution history, then it can also employ few-shot learning, described in the following Section 4.2.2, in each of these prompts.

Similarly to the syntax phase, rather than picking a single prompt template, we use all prompts generated and rely on the semantic oracle to identify viable repair candidates. Each prompt given

```
[[Shot Starts]]
# Incorrect Program #
print (m+n)
# Correct Program #
print (m*n)
[Shot Ends]

[[Buggy Program Starts]]
### Buggy Program ###
sum = m
i = 0
while i < n:
  sum += 1
  i += 1
print (sum)
[[Buggy Program Ends]]

[[Test Suite Starts]]
#input:
2 2
#output:
4

#input:
2 3
#output:
6
[[Test Suite Ends]]

### Correct Program ###
```

Fig. 7. An illustrative example of few-shot learning in PyDex. The incorrect program in the shot and the target buggy program have the same test suite execution [pass, fail].

to the LLMC can generate up to K candidates, where we heuristically set K to ten to balance the exploration of candidates with search space explosion. Each of these candidates is given to the semantic oracle, which executes that candidate on the test suite. We remove any candidate programs that result in a runtime exception or fail to satisfy any test cases.

If there are multiple valid candidate programs after the semantic phase, we return the one with the smallest token-based edit distance [Yasunaga and Liang 2021] to the student's submission as the repaired program.

4.2.2 Few-Shot Learning. If PyDex has access to other students' programs it can employ few-shot learning. In contrast to other repair systems, such as Refactory [Hu et al. 2019], that typically employ only correct programs, PyDex's few-shots consist of both correct *and* incorrect programs.

In particular, PyDex's few-shot learning example bank consists of pairs of program versions (p, p') where both p and p' satisfy the syntax oracle, p' satisfies the semantic oracle but p does not, and p is a historical edit-version ancestor of p'. Given a candidate program produced by the syntax phase of PyDex, we retrieve the three most similar p and their associated correct versions p' to include as shots in the LLMC prompts produced by the semantic prompt generator.

We take inspiration from traditional automated program repair and say two programs are similar if they result in similar test suite executions [Perry et al. 2019]. We define a test suite execution vector for program p that captures test failures as $T_p \in \mathbb{B}^n = (t_0, \cdots, t_n)$ where n is the number of test cases, and t_i is the boolean failure status of the ith test. We define the similarity function between p_1 and p_2 as $1 - \textsc{Hamming}(T_{p_1}, T_{p_2})$, where Hamming is the normalized Hamming distance [Hamming 1950] between the two vectors.

Figure 7 is an illustrative example (note this is not an actual student problem, we have created a simplified example) of a prompt structure for our few-shot learning setting. In this prompt example, we lay out in few-shots as a prefix, followed by the target buggy program, the test suite information, and then a prefix to prompt the model to return a corrected version of the buggy program. Note that if PyDex does not have access to peer programs, then it can still query the LLMC using a zero-shot approach. In our evaluation (Section 5) we show that this ablated strategy still performs competitively.

5 EVALUATION

We explore the following two research questions in our evaluation of PyDex:

- **(RQ1)** How does PyDex's overall performance compare to different baselines, which combine state-of-the-art syntactic and semantic repair approaches?
- **(RQ2)** What is the impact of the underlying design decisions in PyDex? Specifically, what is the impact of the structure-based program chunking, iterative querying, test-case-based few-shot selection, and multi-modal ensembled prompts?

Implementation. We have built a PyDex prototype using a mix of Python and open-source software libraries. The core of PyDex's implementation consists of approximately 600 lines of Python code, which is 5 to 10 times less than a typical symbolic repair system in the education domain [Gulwani et al. 2018; Hu et al. 2019; Rolim et al. 2017]. In addition to the reduced engineering efforts, PyDex can handle both syntactic and semantic bugs in one system, while most systems address one type.

We selected the top 10 program candidates in each syntax and semantics phase based on the average token log probabilities produced by the LLMC. We used OpenAI's Codex as our LLMC. Specifically, we used the completion model. We found that other models, such as Codex Edit [Open AI 2022], did not perform as well. We set the temperature to 0.8 based on preliminary experiments. We ran experiments on a Windows VM (Intel i7 CPU, 32GB RAM).

Benchmarks. We derived a benchmark set by selecting programs from a collection of introductory Python assignments collected by third-party authors in a large Indian university [H. Padmanabha et al. 2023]. This dataset is a Python-version of the dataset described in [Chhatbar et al. 2020].

The dataset contains 18 assignments, each with a problem description, the test suite, and students' authoring history. A student's history consists of an ordered collection of program versions, where each version can be an explicit submission to the testing server, or a periodic (passive) snapshot – the dataset does not have a way to distinguish between these. For each assignment, we selected the students that had an eventually correct program. For each such student, we followed the standard practice [Rolim et al. 2017] of collecting the latest (closest to the correct version in time) version that had a syntactic mistake as our repair target. This results in a total of 286 *program pairs*, each consisting of a buggy and a ground-truth correct program version. We make available our filtered evaluation dataset here: https://github.com/microsoft/prose-benchmarks/tree/main/PyDex.

We removed three assignments that required reading files that are not reported in the dataset or that asked students to generate a PDF plot, which makes assessing correctness difficult without extra manual inspection. We manually checked the students' submissions and we found their errors

Table 1. Statistics of 286 syntax errors reported in the datasets.

Error Type	Appearance
Misspelling, missing, or misusing parentheses and brackets	68
Incorrect indentation	52
Misspelling, missing, or misusing comma, colon and semicolon	41
Misspelling, missing, or misusing Python keywords	11
Misspelling, missing, or misusing quotation marks	10
Misspelling, missing, or misusing assignment operator (=)	5
Errors on defining and calling functions	5
Using Python 2 syntax	3
Others	91
In total	286

were diverse. The repaired syntax errors in PyDex benchmarks include incorrect indentation, illegal usage of an empty block, misspelling a keyword, undefined symbols, and unmatched delimiters such as parentheses, among others. Table 1 shows a summary of these errors.

Baselines. Most repair systems focus on either syntactic or semantic repairs.[3] To create a state-of-the-art baseline that performs *both*, we combined BIFI, a state-of-the-art transformer-based Python syntax repair tool, and Refactory, a state-of-the-art semantics repair tool designed for introductory Python assignments.

To run this baseline, we gave BIFI the original student program with syntax errors and generated 50 candidate programs for each buggy program. For each candidate, we ran the syntax oracle and checked for syntactic correctness. For each candidate that passed the syntax check, we called Refactory along with the instructor's reference solution.[4] If Refactory can repair *any* of the candidates, we say it has repaired the student's program. If there are multiple candidate programs that passed the test suite, we choose the one with the smallest token edit distance from the original.

We also consider two additional baselines. We use PyDex to produce syntax repairs and then apply Refactory to solve any semantic repairs, as described previously. We refer to this baseline as PyDex+Refactory.

Finally, we consider a baseline that uses a version of GenProg [Le Goues et al. 2012] for semantic repairs. Because there is no official implementation of GenProg for Python programs, we took a publicly available implementation [Zeller 2023] that adapts portions of the algorithm to better match Python syntax. This approach evolves a student's buggy submission and can also incorporate statements from the instructor's reference solution. Like Refactory, GenProg assumes the input program does not contain syntax errors. So to run our comparison, we use PyDex to produce syntax repairs and then apply GenProg. We use GenProg to generate up to 10 candidates with a 30-second timeout for each repair attempt. We refer to this baseline as PyDex+GenProg.

5.1 RQ1: Overall Repair Performance

Table 2 shows that without few-shot learning PyDex can repair 86.71% of student programs. This repair rate climbs to 96.5% if we incorporate few shots. In contrast, BIFI+Refactory, PyDex+Refactory, and PyDex+GenProg repair 67.13%, 83.57%, and 49.30% of student programs, respectively.

[3] A notable exception in the education domain is sk_p[Pu et al. 2016], however, this tool is not publicly available and the repair rate (29%) described in the paper is low compared to our baselines.

[4] The original Refactory paper shows that there is little-to-no performance difference between providing one and multiple correct reference programs.

Table 2. PyDex (without few shots) repairs a larger fraction of programs (86.71%) compared to our baselines (67.13%, 83.57%, 49.3%). On average, PyDex repairs are closer in terms of token edit distance (TED) to the original student program compared to two of the three baselines. Adding few-shots based on other peers' programs raises PyDex's repair rate to 96.50% while keeping a comparable average token edit distance (29.68). To save space in the table, "ID" represents the problem ID in the dataset, "# Sub" means the number of submissions of this problem, and "RR" is short for repair rate.

Method		PyDex (without few-shot)		PyDex (with few-shot)		BIFI + Refactory		PyDex(syntax) + Refactory		PyDex(syntax) + GenProg	
ID	# Sub	RR (%)	Mean TED (SD)	RR (%)	Mean TED (SD)	RR (%)	Mean TED (SD)	RR (%)	Mean TED (SD)	RR (%)	Mean TED (SD)
2865	11	100.00	6.45 (4.74)	100.00	6.45 (4.74)	100.00	16.45 (7.00)	100.00	20.55 (6.08)	90.91	16.10 (6.08)
2868	28	85.71	8.79 (8.94)	100.00	8.64 (8.49)	82.14	36.35 (19.26)	96.43	35.15 (19.24)	96.43	26.00 (9.06)
2869	23	95.65	16.68 (18.47)	100.00	10.30 (10.99)	69.57	47.75 (20.27)	100.00	42.35 (19.77)	30.43	20.29 (12.85)
2870	27	74.07	10.00 (13.33)	100.00	15.00 (19.35)	85.19	39.48 (31.38)	92.59	35.72 (31.78)	33.33	20.22 (21.35)
2872	18	100.00	8.33 (15.15)	100.00	7.39 (13.01)	72.22	105.08 (34.58)	100.00	103.06 (38.65)	88.89	15.94 (6.56)
2873	32	78.13	12.00 (16.18)	90.63	12.93 (15.47)	84.38	75.00 (19.75)	100.00	71.41 (20.37)	25.00	18.63 (5.48)
2874	16	100.00	9.56 (12.50)	100.00	8.50 (11.76)	87.50	35.79 (18.63)	100.00	38.94 (31.43)	75.00	15.83 (5.48)
2875	23	86.96	14.75 (19.97)	100.00	11.52 (12.52)	78.26	63.22 (28.97)	100.00	58.65 (28.55)	47.83	17.09 (7.94)
2877	21	100.00	9.71 (16.82)	100.00	9.14 (16.79)	80.95	67.47 (27.87)	100.00	57.95 (32.19)	85.71	19.44 (11.49)
2878	25	100.00	37.00 (60.16)	100.00	36.32 (59.53)	68.00	138.18 (44.17)	88.00	167.50 (66.11)	52.00	21.46 (15.49)
2879	21	76.19	131.19 (51.62)	85.71	132.78 (52.61)	52.38	183.45 (40.90)	71.43	195.33 (55.24)	4.76	229.00 (N/A)
2882	23	60.87	90.64 (71.76)	91.30	106.57 (77.57)	0.00	N/A	0.0	N/A	17.39	42.00 (18.30)
2883	5	100.00	17.40 (14.67)	100.00	17.40 (14.67)	40.00	141.00 (8.49)	100.00	103.60 (39.37)	60.00	46.00 (19.47)
2920	10	80.00	84.38 (67.62)	80.00	53.50 (66.05)	0.00	N/A	10.00	69.00 (N/A)	20.00	42.00 (5.66)
2921	3	100.00	28.00 (3.61)	100.00	28.00 (3.61)	0.00	N/A	0.0	N/A	0.0	N/A
Overall		86.71	28.59	96.50	29.68	67.13	70.39	83.57	73.53	49.30	22.82

The mean token edit distance between the buggy program and our repaired program is 28.59 (no few-shot) and 29.68 (with few shots) compared to 70.39 for BIFI+Refactory, 73.53 for PyDex+Refactory, and 22.82 for PyDex+GenProg.

We carry out a statistical analysis to compare performance across these systems. We exclude PyDex without few-shots as this is effectively an ablation. We compare the repair rate and mean token edit distance across assignments and systems by using paired t-tests. We use paired tests as performance is paired at the assignment level. We carry out the paired t-tests using pairwise comparisons with a Bonferroni adjustment for repeated comparisons. For the repair rate, we consider a 1-sided test with an alternative hypothesis of performance being greater for PyDex. For the mean token edit distance (TED), we consider a 1-sided test with an alternative hypothesis of PyDex's TED being smaller. Because TED can be undefined if a system fails to repair any programs, we exclude assignments where any baseline has a repair rate of zero (i.e. assignments 2882, 2920, 2921).

For repair rates, we find that the comparison between PyDex and BIFI+Refactory (and similarly between PyDex+Refactory and BIFI+Refactory) is statistically significant (at 0.01), and so we reject the null hypothesis. We find that the comparison between PyDex and PyDex+Refactory results in a p-value of 0.057 (after Bonferroni adjustment), so we do not reject the null hypothesis in this case (though if we reduce the number of pair-wise comparisons it is significant). Finally, the comparison between PyDex and PyDex+GenProg is statistically significant at 0.01.

For mean TED, we find that the comparison between PyDex and BIFI+Refactory (as well as between PyDex and PyDex+Refactory) is significant at p=0.01. We also find that the comparison between PyDex and PyDex+GenProg is not statistically significant.

From this analysis, we conclude[5] that PyDex outperforms the baseline BIFI+Refactory on both repair rate (higher) and size of repair (smaller), PyDex+Refactory on the size of repair but not necessarily on repair rate, and PyDex+GenProg on repair rate but not on the size of repair.

[5]We arrived at similar conclusions with two-sided tests.

Table 3. The first stage in the repair process is to fix syntax errors. PyDex can produce a syntactically valid candidate for all programs in our benchmark, compared to 80.07% for BIFI. On average, PyDex's repairs are also closer to the original program (edit distance of 5.46 versus 25.07).

Method		PyDex	BIFI	
ID	# Sub	Mean TED (SD)	Repair rate (%)	Mean TED (SD)
2865	11	2.18 (1.25)	100.00	1.82 (0.75)
2868	28	2.75 (2.17)	82.14	1.83 (1.11)
2869	23	2.91 (2.41)	73.91	1.47 (0.80)
2870	27	2.33 (2.18)	85.19	2.04 (1.89)
2872	18	2.39 (1.2)	72.22	1.62 (0.87)
2873	32	2.84 (2.58)	84.38	2.56 (2.04)
2874	16	2.06 (1.84)	87.50	2.07 (2.02)
2875	23	2.78 (2.71)	78.26	1.78 (1.56)
2877	21	2.19 (1.29)	80.95	3.18 (7.47)
2878	25	4.84 (8.58)	0.00	40.2 (60.65)
2879	21	18.86 (21.24)	66.67	117.00 (58.15)
2882	23	17.39 (23.23)	86.96	127.65 (74.76)
2883	5	5.60 (9.74)	80.00	36.25 (37.98)
2920	10	10.30 (18.68)	50.00	51.75 (54.90)
2921	3	1.67 (0.58)	100.00	1.33 (0.58)
Overall		5.46	80.07	25.07

Repairing semantic errors typically depends on first resolving any syntactic errors. Indeed, students often focus on resolving mistakes reported by the parser/compiler before they move on to debugging test cases. PyDex's architecture reflects this approach. As a result, we also want to understand syntax repair performance by comparing just PyDex and BIFI.

Table 3 summarizes the *syntax repair rates* across assignments and approaches. Our results show that PyDex repairs the syntax bugs in all of the 286 programs, with a 100% syntax repair rate. This outperforms the state-of-the-art BIFI, which has a syntax repair rate of 80.07%. In addition, PyDex's syntax repairs have a substantially lower mean token edit distance (5.46 versus 25.07), meaning our repairs on average introduce fewer changes to the original programs, which may facilitate understanding of the fixes.

We also observed that in 17 out of 286 cases, BIFI fails to handle the input program, potentially due to lexer issues. This highlights another advantage of using PyDex to repair programs because PyDex does not have any constraints over the input as a result of its prompt-based learning strategy.

BIFI is very effective at repairing small syntax mistakes in assignments of lower difficulty. For example, in assignment 2865, BIFI repairs all syntax errors and does so with a smaller average token edit distance (1.82 versus 2.18) compared to PyDex. One interesting direction for future work is to combine BIFI with PyDex, as the repairs can be complementary. In this case, PyDex could focus on generating more complex repairs and BIFI could focus on small edits for simpler tasks such as missing a quote in a string.

5.2 RQ2: Ablation Study

We now present the results of experiments to analyze different design choices in PyDex. PyDex uses multimodal prompts, iterative querying, test-case-based few-shot selection, and structure-based program chunking to repair student mistakes. The power of few-shot selection was already shown in Table 2. We will now present the results of the other three design choices.

Table 4. Chunking reduces the average token edit distance across all assignments. PG is short for performance gain.

Method	PyDex (no chunking)	PyDex (with chunking)	
ID	Mean TED (SD)	Mean TED (SD)	PG (%)
2865	2.45 (1.21)	2.18 (1.25)	11.11
2868	2.82 (2.14)	2.75 (2.17)	2.53
2869	2.91 (2.41)	2.91 (2.41)	0.00
2870	2.33 (2.18)	2.33 (2.18)	0.00
2872	2.44 (1.2)	2.39 (1.2)	2.27
2873	3.09 (2.61)	2.84 (2.58)	8.08
2874	2.25 (2.08)	2.06 (1.84)	8.33
2875	3.52 (4.13)	2.78 (2.71)	20.99
2877	2.29 (1.27)	2.19 (1.29)	4.17
2878	11.08 (20.3)	4.84 (8.58)	56.32
2879	33.14 (24.91)	18.86 (21.24)	43.10
2882	42.57 (41.54)	17.39 (23.23)	59.14
2883	6.20 (11.08)	5.60 (9.74)	9.68
2920	15.20 (19.45)	10.30 (18.68)	32.24
2921	1.67 (0.58)	1.67 (0.58)	0.00
Overall	9.38	5.46	41.79

5.2.1 Program Chunking. In the syntax stage, PyDex first extracts program chunks from the original buggy program as detailed in Section 4. The intuition is that these chunks contain the syntax error we want to fix, along with the surrounding context, while excluding code lines that are not relevant to the fix. Our goal is to reduce the number of (spurious) edits produced by the LLMC by reducing the code surface in the prompt.

To evaluate the impact of program chunking on the syntax repair stage, we removed it from PyDex and compared syntax repair performance to the original approach. Table 4 shows the average token edit distance produced in the syntax phase with and without program chunking. We found that program chunking can reduce the average token edit distance up to 56.32% (problem assignment 2878). Overall, the average token edit distance is reduced from 9.38 to 5.46 (41.79%) by adding program chunking.

5.2.2 Iterative Querying. Students typically resolve syntax errors first and then move on to resolving semantic mistakes. PyDex's architecture follows this same intuition. To compare the effectiveness of this iterative approach, we ran a variant of PyDex that addresses both syntax and semantic bugs in a *single* round. Table 5 shows the results of this ablated variant and full PyDex (without few-shots). We find that splitting concerns into two phases results in an increase in the overall repair rate from 82.87% to 86.71%. Using two phases increases the average TED slightly (26.79 to 28.59). However, for the majority of the problems (10 out of 15), PyDex (with iterative) has a smaller or equal mean TED than PyDex (no iterative). In the remaining 5 problems, we found PyDex with iterative querying has a larger mean TED because it successfully generates repairs for challenging buggy submissions where PyDex (no iterative) is unable to repair.

5.2.3 Multimodal Prompts. PyDex combines different types of input (code, natural language, test cases) into its prompts. This richness of inputs is a particular advantage of the educational setting. PyDex *ensembles* these various prompts by querying the LLMC and *then* relying on the (syntax or

Table 5. PyDex performs iterative querying, splitting the repair procedure into a syntactic and a semantic phase. We find that this iterative approach raises the overall repair rate (RR) from 82.87% to 86.71% (without few-shots).

Method	PyDex (no iterative)		PyDex (with iterative)	
ID	RR (%)	Mean TED (SD)	RR (%)	Mean TED (SD)
2865	100.00	6.45 (4.74)	100.00	6.45 (4.74)
2868	85.71	8.92 (8.88)	85.71	8.79 (8.94)
2869	86.96	13.35 (12.36)	95.65	16.68 (18.47)
2870	70.37	11.42 (13.87)	74.07	10.00 (13.33)
2872	100.00	8.50 (15.22)	100.00	8.33 (15.15)
2873	71.88	9.48 (11.63)	78.13	12.00 (16.18)
2874	100.00	9.75 (12.51)	100.00	9.56 (12.50)
2875	82.61	13.16 (18.69)	86.96	14.75 (19.97)
2877	100.00	9.71 (16.82)	100.00	9.71 (16.82)
2878	100.00	38.16 (62.24)	100.00	37.00 (60.16)
2879	71.43	130.07 (53.23)	76.19	131.19 (51.62)
2882	56.52	97.85 (72.64)	60.87	90.64 (71.76)
2883	100.00	17.40 (14.67)	100.00	17.40 (14.67)
2920	50.00	50.20 (48.9)	80.00	84.38 (67.62)
2921	100.00	28.00 (3.61)	100.00	28.00 (3.61)
Overall	82.87	26.79	86.71	28.59

semantics) oracle to filter out candidates. This approach is based on the idea that different prompts may produce complementary candidates.

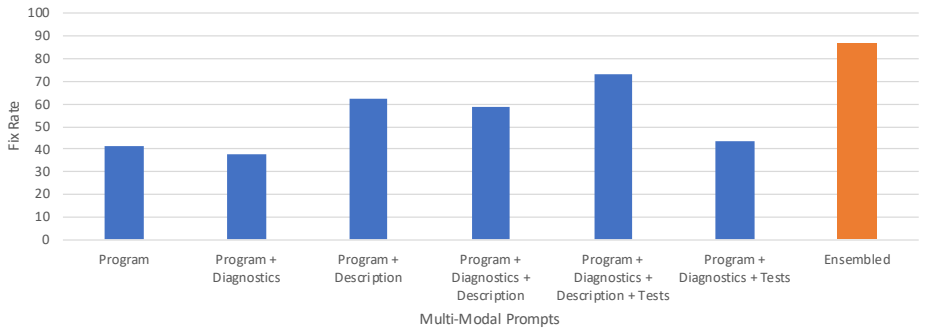

Fig. 8. PyDex *ensembles* multiple prompts, by querying and then relying on the (syntax and semantic) oracles to rule out invalid candidates. Ensembling complementary prompts outperforms any particular prompt.

Figure 8 shows that different prompt structures result in different overall performances in terms of fix rate. If a single prompt structure needs to be chosen, `Program + Diagnostics + Description + Tests` structure is most effective in this experiment. However, if we *ensemble* the candidates, these are complementary.

6 DISCUSSION

We now discuss two important points. First, we provide details on why simply combining a state-of-the-art syntax repair tool and a separate semantics repair tool is not as effective as using PyDex. Second, we discuss important limitations.

6.1 Why Not Combine a State-of-the-Art Syntactic Fixer and Semantics Fixer to Repair Programs?

We investigated why BIFI+Refactory, which combines two state-of-the-art repair systems, produces repairs that (on average) have a larger token edit distance compared to PyDex. We found that in some cases, BIFI produces repairs by deleting a portion of the code snippet that contains the syntax errors. Although this is an effective way to deal with syntax errors, it makes repairing semantic errors harder by deleting parts that may capture crucial logic.

Below is one such example from our evaluation. The code snippet contains a syntax mistake in the last line. The parser complains that the *"Expression cannot contain an assignment ="*. In particular, the student has written an equal (highlighted below in red) when they should have used a plus operator (which corresponds to the repair produced by PyDex).

```
1  marksSum={}
2  for i in total:
3    if int(i[0])not in marksSum:
4      marksSum[int(i[0])]= int(i[2])
5    else:
6      k=int(i[2])
7      marksSum[int(i[0])]+=k
8  for i in sorted(marksSum):
9    print(str(i)=":"+str(marksSum[i]))
```

However, BIFI produced a different fix by removing the second **for** loop (lines 8-9) completely. This deletion introduced challenges for Refactory in the later semantic repair phase. Although Refactory in the end successfully repaired this program, the repair it generated is syntactically equivalent to the reference solution and is effectively completely rewritten with respect to the original incorrect program.

Overall, our comparison between PyDex and BIFI+Refactory highlights the challenges in combining state-of-the-art syntax and semantics tools to repair incorrect introductory programming assignments. BIFI and Refactory each focus on their targets, syntactic bug repair and semantic bug repair, respectively, and combining them may result in unexpected performance. Additionally, combining BIFI and Refactory required non-trivial engineering efforts (approximately 3 weeks of effort from one Python expert). This further motivates the need for a unified approach that can handle both types of bugs for introductory Python programmers.

6.2 Limitations

PyDex validates candidate repairs by comparing execution results on the test suite with the reference program given by instructors. Validating program correctness through tests is not as strong as formal verification. To the best of our knowledge, the use of tests as a proxy for correctness is standard in the educational domain [Gulwani et al. 2018; Singh et al. 2013].

We carried out our evaluation on one particular set of 286 student programs. The size of the dataset is on par with literature on state-of-the-art automated program repair [Ahmed et al. 2022a; Li et al. 2022b], but increasing the size of the evaluation dataset may provide additional insights and present an opportunity for future work.

PyDex relies on an LLM so it inherits its limitations. PyDex (like the LLM it uses) does not have a soundness or completeness guarantee. Also, we acknowledge randomness is another limitation in PyDex and we sampled and picked the top repair candidates to mitigate the effect caused by randomness. These limitations might be addressed by requesting further information from the students, and it remains future work.

Language requirement. We scoped PyDex to introductory Python assignments as that is the only domain where we have a suitable dataset and carry out an evaluation. Other education tools [Bhatia et al. 2018; Wang et al. 2018b] share this same limitation of focusing on one programming language. However, the principles behind the design of PyDex apply to programs written in other imperative languages, as conceptually none of our prompt engineering methods are language-specific. Applying PyDex to education assignments in these other languages would require reimplementation of the chunking procedure, test-case execution harness, and swapping the syntax oracle for the corresponding domain. For example, if we were to apply PyDex to Java programs, the chunker can rely on control-flow keywords (if, for, while), but indentation may no longer be meaningful; the execution harness could be replaced with JUnit, and the syntax oracle could be replaced with javac.

Data leakage. Data leakage is also a threat to validity in PyDex. PyDex is built on top of Codex, and Codex is trained on public internet data. To have a fair comparison, we only use a non-public dataset as our evaluation target to mitigate the data leakage problem (all our results are on this dataset). Using a public dataset could otherwise inflate performance. This limitation is unfortunately shared by all existing work that uses LLMs. Using a non-public dataset is our best effort, but we agree that data leakage cannot be completely avoided at this stage. For example, "determine if a string is a palindrome" is a question used in our evaluation, but we also found "determine if a string is a palindrome" is also one of the questions in the HumanEval dataset (the human-written dataset used to evaluate the original Codex model).

Moreover, for introductory programming assignments, PyDex provides unique value in that it can craft and customize the solution to the student's errors (i.e., smaller edit distance patches, as shown in our evaluation). Students can always search for reference solutions as repairs, but we observe that this is not a good option in practice because the differences between buggy program and reference program can be large, as we show in Section 2.

Why can students benefit pedagogically from a tool that automatically repairs their buggy programs? Automatically fixing students' submissions is not the same as providing an explanation for their mistakes. However, human feedback, in the form of a student-tailored corrected solution, represents a substantial time investment [Keuning et al. 2016; Singh et al. 2013]. Absent such time investment, students typically must rely on a reference solution. This is the starting motivation for employing automated repair in this context. In this way, PyDex provides a preferable alternative to comparing to a standard answer key. Furthermore, a repaired solution is often a starting point for more meaningful feedback. For example, Tung et al [Phung et al. 2023a] produced a syntax repair to then generate a natural language explanation of the error and needed changes.

Runtimes of the different tools. We use tools with substantially different environments. PyDex relies on an API (so network time plays a role), BIFI requires GPU-based computing for inference, and GenProg is done on a CPU. Therefore, we did not compare runtimes as they would be hard to interpret. More importantly, from our analysis, the lower repair rate of the baselines is due to the repair capability, not tool timeouts.

7 RELATED WORK

Automated Program Repair. The programming languages and software engineering community has a long history of developing tools for automatically repairing errors in buggy programs. Existing approaches have applied a variety of technical ideas, including program analysis [Mechtaev et al. 2018, 2016; Shariffdeen et al. 2021; Wan et al. 2023; Zhang et al. 2021], search-based techniques [Wong et al. 2021] like genetic programming [Kim et al. 2013; Le Goues et al. 2012; Qi et al. 2014b], machine learning [Ahmed et al. 2022b; Bhatia et al. 2018; Long et al. 2017; Long and Rinard 2016; Santolucito et al. 2022; Wang et al. 2018b; Zhang et al. 2020] and more recently LLM [Xia and Zhang 2023a,b].

A particularly popular approach to APR consists of generating many program candidates, typically derived by performing syntactic transformations of the original buggy program, and then validating these candidates using a test suite as an oracle [Le Goues et al. 2012]. Similarly, PyDex uses a syntax oracle (the Python parser) and semantic oracle (test cases) to validate candidate programs produced. However, state-of-the-art APR tools are limited to repairing either syntax [Joshi et al. 2022] or semantic errors [Fan et al. 2023], but not both. PyDex significantly differs from these existing tools by automatically repairing both syntax and semantic errors in buggy programs.

In addition, PyDex employs a large language model (Codex) as the main program transformation module and uses an ensemble of multi-modal prompts to improve its success rate. Therefore, PyDex is able to generate complex repairs, which are difficult to address by existing traditional APR techniques [Le Goues et al. 2012; Long and Rinard 2015; Mechtaev et al. 2018, 2016; Qi et al. 2014a; Xuan et al. 2017], which often focus on specific error types, are limited to a small number of edits, and repair specific statements (such as conditionals) exclusively. Moreover, PyDex targets students' incorrect submissions, rather than professional developers' production bugs or LLM-generated bugs [Chen et al. 2023; Fan et al. 2023]. As a result, PyDex has two additional requirements: 1) minimizing the size of the change made to allow students to better learn from the repaired program, and 2) reducing the engineering efforts to run the APR tool.

AI for Programming Education. AI has been extensively applied to the domain of education [Finnie-Ansley et al. 2022; Li et al. 2022a]. Past programming education research has explored topics including feedback generation [Gulwani et al. 2018; Hu et al. 2021, 2019; Phung et al. 2023b; Rolim et al. 2017; Singh et al. 2013; Song et al. 2021; Wang et al. 2018a; Zhang et al. 2023] and program repair [Dinella et al. 2020; Lu et al. 2021; Wang et al. 2018b; Xin and Reiss 2017; Yasunaga and Liang 2021; Yi et al. 2017]. PyDex is complementary to this work, showing that the task of program repair in this domain can be successfully tackled using an LLMC.

LLMs for Code Intelligence. Large pre-trained language models, such as OpenAI's Codex, Salesforce CodeGen [Nijkamp et al. 2022], and BigScience's BLOOM[Laurençon et al. [n. d.]], have been shown to be effective for a range of code intelligence tasks. For example, Microsoft's Copilot[cop 2024] builds on Codex to produce more effective single-line and multi-line code completion suggestions. Prior work has shown that such LLMs can also be used for repairing programs outside of the educational context [Dinella et al. 2022; Lian et al. 2023; Rahmani et al. 2021; Verbruggen et al. 2021; Xiang et al. 2023; Zhang et al. 2022]. Using these models to perform code generation from informal specifications, such as natural language, has also been a topic of active research [Li et al. 2022a]. Similarly to this work, PyDex uses an LLM but is designed to focus on student programming, and as such our design decisions (e.g., reducing token edit distance) may not apply to other domains such as professional developers.

8 CONCLUSION

We introduced an approach to repair syntactic and semantic mistakes in introductory Python assignments. At the core of our approach sits a large language model trained on code. We leverage multi-modal prompts, iterative querying, test-case-based few-shot selection, and program chunking to produce repairs. We implement our approach using Codex in a system called PyDex and evaluate it on real student programs. Our results show that our unified system PyDex can effectively repair real student programs, while producing smaller patches.

ACKNOWLEDGMENTS

We thank OOPSLA reviewers for their insightful comments. Jialu Zhang and Ruzica Piskac were supported in part by NSF grants CCF-2106845 and CCF-2131476.

REFERENCES

2024. Microsoft Copilot. https://github.com/features/copilot/.

Toufique Ahmed, Noah Rose Ledesma, and Premkumar Devanbu. 2022b. SYNFIX: Automatically Fixing Syntax Errors using Compiler Diagnostics. arXiv:2104.14671 [cs.SE]

Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. 2022a. Verifix: Verified Repair of Programming Assignments. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 74 (Jul 2022), 31 pages.

Umair Z Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. 78–87.

Emery Berger. 2022. Coping with Copilot. https://www.sigarch.org/coping-with-copilot/.

Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-Symbolic Program Corrector for Introductory Programming Assignments. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 60–70. https://doi.org/10.1145/3180155.3180219

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020b. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

Tom B. Brown et al. 2020a. Language Models are Few-Shot Learners. In *NeurIPS 2020, December 6-12, 2020, virtual.*

Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. arXiv:2304.05128 [cs.CL]

Darshak Chhatbar, Umair Z. Ahmed, and Purushottam Kar. 2020. MACER: A Modular Framework for Accelerated Compilation Error Repair. In *Artificial Intelligence in Education: 21st International Conference, AIED 2020, Ifrane, Morocco, July 6–10, 2020, Proceedings, Part I* (Ifrane, Morocco). Springer-Verlag, Berlin, Heidelberg, 106–117.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2130–2141.

Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Australasian Computing Education Conference* (Virtual Event, Australia) *(ACE '22)*. Association for Computing Machinery, New York, NY, USA, 10–19.

Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. *SIGPLAN Not.* 53, 4 (June 2018), 465–480.

Sharath H. Padmanabha, Fahad Shaikh, Mayank Bansal, Debanjan Chatterjee, Preeti Singh, Amey Karkare, and Purushottam Kar. 2023. Advances in Automated Pedagogical Compile-time Error Repair. In *Proceedings of the 16th Innovations in Software Engineering Conference (ISEC '23)*. Association for Computing Machinery, New York, NY, USA, Article 11, 11 pages. https://doi.org/10.1145/3578527.3578535

R. W. Hamming. 1950. Error detecting and error correcting codes. *The Bell System Technical Journal* 29, 2 (1950), 147–160. https://doi.org/10.1002/j.1538-7305.1950.tb00463.x

Jingmei Hu, Priyan Vaithilingam, Stephen Chong, Margo Seltzer, and Elena L. Glassman. 2021. Assuage: Assembly Synthesis Using A Guided Exploration. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 134–148. https://doi.org/10.1145/3472749.3474740

Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 388–398. https://doi.org/10.1109/ASE.2019.00044

Harshit Joshi, José Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. 2022. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. *arXiv preprint arXiv:2208.11640* (2022).

Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 295–306.

Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2016, Arequipa, Peru, July 9-13, 2016*, Alison Clear, Ernesto Cuadros-Vargas, Janet Carter, and Yván

Túpac (Eds.). ACM, 41–46. https://doi.org/10.1145/2899415.2899422

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 802–811.

Hugo Laurençon, Lucile Saulnier, Thomas Wang, Christopher Akiki, Albert Villanova del Moral, Teven Le Scao, Leandro Von Werra, Chenghao Mou, Eduardo González Ponferrada, Huu Nguyen, et al. [n. d.]. The BigScience Corpus A 1.6 TB Composite Multilingual Dataset. ([n. d.]).

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (2019), 56–65.

Leping Li, Hui Liu, Kejun Li, Yanjie Jiang, and Rui Sun. 2022b. Generating Concise Patches for Newly Released Programming Assignments. *IEEE Transactions on Software Engineering* (2022), 1–1. https://doi.org/10.1109/TSE.2022.3153522

Yujia Li et al. 2022a. Competition-Level Code Generation with AlphaCode.

Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, and Tianyin Xu. 2023. Configuration Validation with Large Language Models. arXiv:2310.09690 [cs.SE]

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv preprint arXiv:2107.13586* (2021).

Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 727–739.

Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312.

Yunlong Lu, Na Meng, and Wenxin Li. 2021. FAPR: Fast and Accurate Program Repair for Introductory Programming Courses. *CoRR* abs/2107.06550 (2021). arXiv:2107.06550 https://arxiv.org/abs/2107.06550

Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 129–139.

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701. https://doi.org/10.1145/2884781.2884807

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. https://doi.org/10.48550/ARXIV.2203.13474

Open AI 2022. New GPT-3 Capabilities: Edit & Insert. https://openai.com/blog/gpt-3-edit-insert/

David M. Perry, Dohyeong Kim, Roopsha Samanta, and Xiangyu Zhang. 2019. SemCluster: Clustering of Imperative Programming Assignments Based on Quantitative Semantic Features. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 860–873. https://doi.org/10.1145/3314221.3314629

Tung Phung, JosÃ© Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023a. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. In *Proceedings of the 16th International Conference on Educational Data Mining*, Mingyu Feng, Tanja KÃ¤ser, and Partha Talukdar (Eds.). International Educational Data Mining Society, Bengaluru, India, 370–377. https://doi.org/10.5281/zenodo.8115653

Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023b. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. *arXiv preprint arXiv:2302.04662* (2023).

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227* (2022).

Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Sk_p: A Neural Program Corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity* (Amsterdam, Netherlands) *(SPLASH Companion 2016)*. Association for Computing Machinery, New York, NY, USA, 39–40.

Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014a. The strength of random search on automated program repair. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 254–265. https://doi.org/10.1145/2568225.2568254

Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014b. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. New York, NY, USA, 254–265.

Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29.

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *ICSE 2017* (icse 2017 ed.).

Mark Santolucito, Jialu Zhang, Ennan Zhai, Jürgen Cito, and Ruzica Piskac. 2022. Learning CI Configuration Correctness for Early Build Feedback. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 1006–1017. https://doi.org/10.1109/SANER53432.2022.00118

Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 390–405.

Natash Singer. 2019. The Hard Part of Computer Science? Getting Into Class. https://www.nytimes.com/2019/01/24/technology/computer-science-courses-college.html. *The New York Times* (Jan. 2019). Accessed: 2012-09-01.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI 13)*. Association for Computing Machinery, New York, NY, USA, 15–26.

Dowon Song, Woosuk Lee, and Hakjoo Oh. 2021. *Context-Aware and Data-Driven Feedback Generation for Programming Assignments*. Association for Computing Machinery, New York, NY, USA, 328–340.

Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic programming by example with pre-trained models. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–25.

Chengcheng Wan, Yuhan Liu, Kuntai Du, Henry Hoffmann, Junchen Jiang, Michael Maire, and Shan Lu. 2023. Run-Time Prevention of Software Integration Failures of Machine Learning APIs. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 264–291. https://doi.org/10.1145/3622806

Ke Wang, Rishabh Singh, and Zhendong Su. 2018a. Search, Align, and Repair: Data-Driven Feedback Generation for Introductory Programming Exercises *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 481–495.

Ke Wang, Zhendong Su, and Rishabh Singh. 2018b. Dynamic Neural Program Embeddings for Program Repair. In *International Conference on Learning Representations*.

Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: balancing edit expressiveness and search effectiveness in automated program repair. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 354–366. https://doi.org/10.1145/3468264.3468600

Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-shot Learning. *arXiv preprint arXiv:2207.08281* (2022).

Chunqiu Steven Xia and Lingming Zhang. 2023a. Conversational Automated Program Repair. arXiv:2301.13246 [cs.SE]

Chunqiu Steven Xia and Lingming Zhang. 2023b. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. arXiv:2304.00385 [cs.SE]

Qiao Xiang, Yuling Lin, Mingjun Fang, Bang Huang, Siyong Huang, Ridi Wen, Franck Le, Linghe Kong, and Jiwu Shu. 2023. Toward Reproducing Network Research Results Using Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks, HotNets 2023, Cambridge, MA, USA, November 28-29, 2023*. ACM, 56–62. https://doi.org/10.1145/3626111.3628189

Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-Related Code for Automated Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 660–670.

Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

Michihiro Yasunaga and Percy Liang. 2021. Break-It-Fix-It: Unsupervised Learning for Program Repair. In *ICML (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 11941–11952.

Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 740–751.

Andreas Zeller. 2023. Repairing Code Automatically. In *The Debugging Book*. CISPA Helmholtz Center for Information Security. https://www.debuggingbook.org/html/Repairer.html Retrieved 2023-02-11 13:21:11+01:00.

Jialu Zhang, De Li, John Charles Kolesar, Hanyuan Shi, and Ruzica Piskac. 2023. Automated Feedback Generation for Competition-Level Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 13, 13 pages. https://doi.org/10.1145/3551349.3560425

Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K. Lahiri. 2022. Using Pre-Trained Language Models to Resolve Textual and Semantic Merge Conflicts (Experience Paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/3533767.3534396

Jialu Zhang, Ruzica Piskac, Ennan Zhai, and Tianyin Xu. 2021. Static Detection of Silent Misconfigurations with Deep Interaction Analysis. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 140 (oct 2021), 30 pages. https://doi.org/10.1145/3485517

Jialu Zhang, Mark Santolucito, and Ruzica Piskac. 2020. Succinct Explanations With Cascading Decision Trees. *CoRR* abs/2010.06631 (2020). arXiv:2010.06631 https://arxiv.org/abs/2010.06631

Received 19-OCT-2023; accepted 2024-02-24