# Seq2Parse: Neurosymbolic Parse Error Repair

ANONYMOUS AUTHOR(S)

We present Seq2Parse, a neurosymbolic approach to automatically repairing parse errors. Seq2Parse is based on the insight that *Symbolic* Error Correcting (EC) Parsers can, in principle, synthesize repairs, but, in practice, are overwhelmed by the many error-correction rules that are not *relevant* to the particular program that requires repair. In contrast, *Neural* approaches are fooled by the large space of possible sequence level edits, but can precisely pinpoint the set of EC-rules that *are* relevant to a particular program. We show how to combine their complementary strengths by using neural methods to train a sequence classifier that predicts the small set of relevant EC-rules for an ill-parsed program, after which, the symbolic EC-parsing algorithm can make short work of generating useful repairs. We train and evaluate our method on a dataset of 1,100,000 Python programs, and show that Seq2Parse is *accurate* and *efficient*: it can parse 94% of our tests within 2.1 seconds, while generating the exact user fix in 1 out 3 of the cases; and *useful*: humans perceive both Seq2Parse-generated error-locations and repairs to be almost as good as human-generated ones in a statistically-significant manner.

## 1 INTRODUCTION

Parse errors can vex novices who are learning to program. Traditional error messages only indicate the first error or produce messages that are either incomprehensibly verbose or not descriptive enough to help swiftly remedy the error [Qian and Lehman 2017; van der Spek et al. 2005]). When they occur in larger code bases, parse errors may even trouble experts, and can require a great deal of effort to fix [Ahadi et al. 2018; Denny et al. 2012; Kummerfeld and Kay 2003].

Owing to their ubiquity and importance, there are *two* established lines of work on automatically suggesting *repairs* for parse errors. In the first line, Programming Languages researchers have investigated *symbolic* approaches starting with classical parsing algorithms, *e.g.*, LR [Aho and Johnson 1974] or Earley [Earley 1970]. These algorithms can accurately locate syntax errors, but do not provide *actionable* feedback on how to fix the error. Aho and Peterson [1972] extends these ideas to implement *error correcting parsers* (EC-Parsers) that use special error production rules to handle programs with syntax errors and synthesize minimal-edit parse error repairs. Sadly, EC-parsers have remained mostly of theoretical interest, as their running time is cubic in the program size, and quadratic in the size of the language's grammar, which has rendered them impractical for real-world languages [McLean and Horspool 1996; Rajasekaran and Nicolae 2014].

In the second line, Machine Learning and NLP researchers have pursued *Deep Neural Network* (DNN) approaches using advanced sequence-to-sequence models [Hardalov et al. 2018; Sutskever et al. 2014] that use a large corpus of code to predict the next token (*e.g.*, at a parse error location). Unfortunately, these methods ignore the high-level structure of the language (or must learn it from vast amounts of data) and hence, lack accuracy in real-world contexts. For example, state-of-the-art methods such as Ahmed et al. [2021] parse and repair only *32%* of real student code with up to 3 syntax errors while Wu et al. [2020] repair only *58%* of syntax errors in a real-world dataset.

In this paper, we present Seq2Parse, a new approach to automatically repairing parse errors based on the following key insight. Symbolic EC-Parsers [Aho and Peterson 1972] can, in principle, synthesize repairs, but, in practice, are overwhelmed by the many error-correction rules that are not *relevant* to the particular program that requires repair. In contrast, Neural approaches are fooled by the large space of possible sequence level edits, but can precisely pinpoint the set of EC-rules that are *relevant* to a particular program. Thus, Seq2Parse addresses the problem of parse error repair by a neurosymbolic approach that combines the complementary strengths of the two lines of work via the following concrete contributions.

***1. Motivation.*** Our first contribution is an empirical analysis of a real-world dataset of more than a million novice Python programs that shows that parse errors constitute the majority of novice

errors, take a long time to fix, and that the fixes themselves can often require multiple edits. This analysis clearly demonstrates that an automated tool that suggests parse error repairs in a few seconds could greatly benefit many novices (§ 2).

***2. Implementation.*** Our second contribution is the design and implementation of Seq2Parse, which exploits the insight above to efficiently and accurately suggest repairs in a neurosymbolic fashion: (1) train sequence classifiers to predict the *relevant* EC-rules for a given program (§ 5), and then (2) use the predicted rules to synthesize repairs via EC-Parsing (§ 6).

***3. Abstraction.*** Predicting the rules is challenging. Standard NLP token-sequence based methods are confused by long trailing contexts that are independent of the parse error. This confusion yields to inaccurate classifiers that predict irrelevant rules yielding woefully low repair rates. Our second key insight eliminates neural confusion via a symbolic intervention: we show how to use Probabilistic Context Free Grammars (PCFGs) to *abstract* long low-level token sequences so that the irrelevant trailing context is compressed into single non-terminals, yielding compressed abstract token sequences that can be accurately understood by DNNs (§ 4).

***4. Evaluation.*** Our final contribution is an evaluation of Seq2Parse using a dataset of more than 1,100,000 Python programs that demonstrates its benefits in three ways. First, we show its *accuracy*: Seq2Parse correctly predicts the right set of error rules 81% of the time when considering the top 20 rules and can parse 94% of our tests within 2.1 seconds with these predictions, a significant improvement over prior methods which were stuck below a 60% repair rate. Second, we demonstrate its *efficiency*: Seq2Parse parses and repairs erroneous programs within 20 seconds 83% of the time, while also generating *the user fix in almost 1 out 3 of the cases*. Finally, we measure the *quality* of the generated repairs via a human study with 39 participants and show that humans perceive both Seq2Parse's edit locations and final repair quality to be useful and helpful, in a statistically-significant manner, even when not equivalent to the user's fix (§ 7).

## 2 A CASE FOR PARSE ERROR REPAIR

We motivate Seq2Parse by analyzing a dataset comprising *1,100,000 erroneous Python programs* and their respective fixes. This dataset was gathered from PythonTutor.com [Guo 2013] between the years 2017 and 2018, previously used in related work [Cosman et al. 2020; Endres et al. 2019]. Each program which throws an uncaught Python exception is paired with the next program by the same user that does not crash, under the assumption that the latter is the fixed version of the former. We discard pairs that are too different between buggy and fixed versions, since these are usually unrelated submissions or complete refactorings. We also discard submissions that violate PythonTutor's policies (*e.g.*, those using forbidden libraries). The resulting dataset contains usable program pairs, representing students from dozens of universities (PythonTutor has been used in many introductory courses [Guo 2013]) as well as non-traditional novices.

One might imagine that parse (or *syntax*) errors are usually easier to locate and repair than other algorithmic or runtime errors [Denny et al. 2012]. For example, the Python parser will immediately inform the programmer about missing parentheses in function argument lists or improper indentation. However, as has also been shown in previous work [Ahadi et al. 2018; Kummerfeld and Kay 2003], our data confirms that programmers (especially novices) deal with these kinds of errors regularly and spend a considerable amount of time fixing them.

***Observation 1: Parse errors are very common.*** Figure 1 presents the statistics of the different types of errors that users encountered in this dataset. We observe that 77.4% of all faulty programs failed with a syntax error, accounting for the vast majority of the errors that (novice) programmers face with their programs. The second category is merely 13.6% of the dataset and represents Python type errors. This is a strong indication that parse errors are a very common category of error.

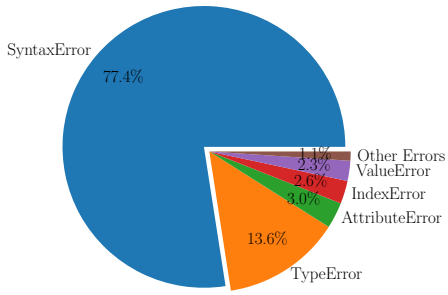



Fig. 1. The Python error type distribution.

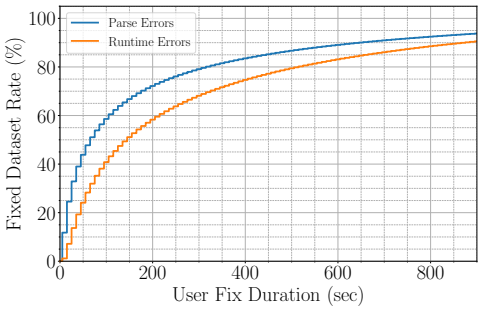



Fig. 2. The repair rates of the Python dataset.


Fig. 3. The Python dataset ratio that is fixed under the given number of token changes.

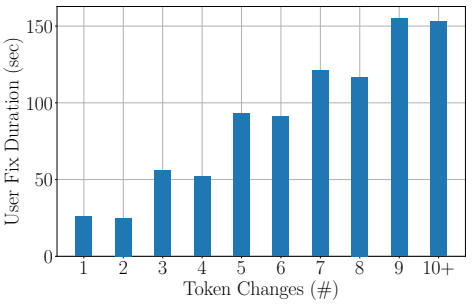



Fig. 4. The average time the user needed to fix the erroneous program for the needed token changes.

***Observation 2: Parse errors take time to fix.*** The web-based compiler used to obtain this dataset provides *server timestamps*. The timestamp is associated with each program attempt submission, erroneous or not. The *repair time* is the difference between the erroneous and fixed program timestamps. This timing can be imprecise, as there are various reasons these timings may be exaggerated, (*e.g.*, users stepping away from the computer, internet lag *etc.*). However, in aggregate, due to the large dataset size, these timings can still be viewed as an approximate metric of the time it took novice programmers to repair their errors.

Figure 2 shows the *programmer repair rate*, *i.e.* the dataset percentage that is repaired under a given amount of time. It presents the repair rate for parse errors and the rest of the error types, grouped together here as *runtime* errors. As expected, parse errors are fixed faster than the rest, but *not by a large difference*. For example, we observe that within 2 minutes, 46% of the runtime errors are repaired, while 63% of the syntax errors are. Although this is a non-trivial difference, we observe that there are still many "simpler" parse errors that required more than 2 minutes to fix.

***Observation 3: Parse errors may need multiple edits to fix.*** The average *token-level changes* needed to fix a program with syntax errors, *i.e.* the number of changes in the lexed program token sequence, is *10.7 token changes*, while the *median is 4*. (This does not count lexeme content changes, such as variable renamings, and thus underapproximates the work required.) As shown in Figure 3, 14.2% of errors need only 1 token change, 23.2% need 2 token changes, 7.0% need 3 and 9.0% need 4. Ultimately, 46.6% of these errors require more than 4 token changes.

```
1  def foo(a):
2    return a + 42
3
4  def bar(a):
5    b = foo(a) + 17
6    return b +
```

(a) A Python program with two functions that manipulate an integer. The second one has a parse error.

```
1  def foo(a):
2    return a + 42
3
4  def bar(a):
5    b = foo(a) + 17
6    return b
```

(b) A fixed version for the previous example that has no parse errors.

Fig. 5. A Python program example with syntax errors (left) and its fix (right).

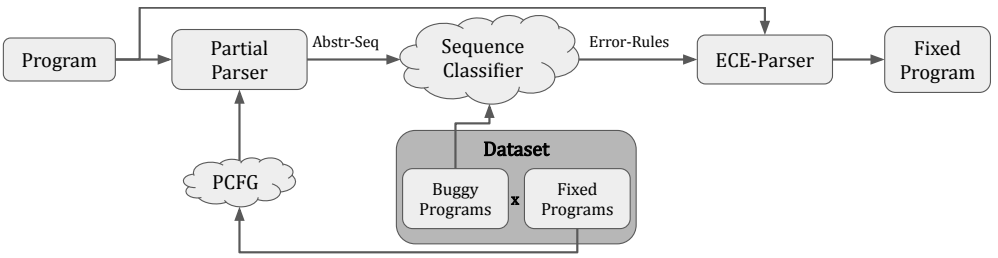



Fig. 6. SEQ2PARSE's overall approach.

***Observation 4: Parse errors with more edits take longer to fix.*** Figure 4 shows the average time for users to fix syntax errors as a function of the number of token changes needed. As expected, with an increasing number of token changes needed, programmers need more time to implement those changes. Most importantly, even for 1 or 2 token changes the average user spends *25 sec*, which is still a considerable amount of time for such simple and short fixes. The repair time jumps to *56 sec* for three token changes.

These four observations indicate that, while some errors can be easily and quickly fixed by programmers using existing error messages, there are many cases where novices struggle with fixing syntax errors. Therefore, we can conclude that an automated tool that parses and repairs such programs in only a few seconds could benefit many novices.

## 3 OVERVIEW

We begin with an overview of SEQ2PARSE's neurosymbolic approach to repairing parse errors, that uses two components. *(Neural)* Given a dataset of ill-parsed programs and their fixes, we partially parse the programs into *abstract sequences* of tokens (§ 3.2), that can be used to train *sequence classifiers* (§ 3.3), that predict program-relevant error rules for new erroneous programs (§ 3.4). *(Symbolic)* Next, given an erroneous program, and a (small) set of predicted *program relevant* error rules, the ECE-parser can exploit the high-level grammatical structure of the language to make short work of synthesizing the best repair (§ 3.1). We now give an overview of SEQ2PARSE, describing these elements in turn, using as a running example, the program in Figure 5a where the programmer has introduced an extra + operator after the `return b` on line 6. This extra + should be deleted, as shown in the developer-fixed program in Figure 5b.

### 3.1 Error Correcting Parsing

***Earley Parsers for Python.*** An Earley parser accepts programs that belong to a language that is defined by a given *grammar G* by using dynamic programming, to store top-down partial

```
S         → Stmts end_marker
Stmts     → Stmt \n | Stmt \n Stmts
Stmt      → FuncDef | ExprStmt
           | RetStmt | PassStmt | ...
FuncDef   → def name Params : Block
Block     → \n indent Stmts dedent
RetStmt   → return | return Args
Args      → ExprStmt | ExprStmt , Args
ExprStmt  → ArExpr | ...
ArExpr    → Literal
           | ArExpr BinOp Literal
Literal   → name | number | ...
```

Fig. 7. Simplified Python production rules.

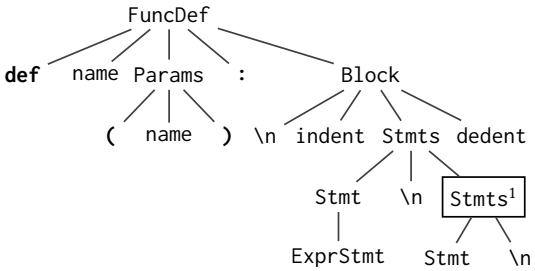



Fig. 8. The partial parse tree generated for `bar` in the example at Figure 5a

parses in a data structure called a *chart* [Earley 1970]. The grammar $G$ has a starting symbol `S` and a set of *production rules*. Figure 7 presents some simplified production rules for the Python programming language that will help parse the program in Figure 5a. *Terminal* symbols (or *tokens*) are syntactical symbols and are here presented in lowercase letters. Uppercase letters denote *non-terminal* symbols, which are rewritten using production rules during a parse. For example, the non-terminal `Stmt` defines all possible Python statements, including expressions (`ExprStmt`), return statements (`RetStmt`), *etc.* Figure 8 shows the top levels of the parse tree for the `bar` function in Figure 5a using these productions rules.

***Error Correcting Parsers for Python.*** An *Error Correcting Earley* (ECE) Parser extends the original algorithm's operations, to find a *minimum-edit* parse for a program with parse errors [Aho and Peterson 1972]. An ECE-Parser extends the original grammar $G$ with a set of *error production rules* to create a new *error grammar* $G'$ which has rules to handle *insertion*, *deletion*, and *replacement* errors. Let's see how to adapt Python's production rules for an ECE-Parser. First, the ECE-Parser adds to $G'$ a new start symbol `New_S`, the helper symbol `Replace` that is used for replacement errors and the symbols `Insert` and `Token` that introduce insertion errors. Additionally, for each terminal `t` in $G$ it adds the new non-terminal `E_t` that introduces errors relevant to the `t` symbol.

Next, in addition to the existing production rules, the error grammar $G'$ has the following error rules. The new start symbol uses the old one with the option of an insertion error at the end:

- `New_S → S | S Insert`

Additionally, for each production rule of a non-terminal `T` in $G$, another non-terminal error rule is added that introduces the terminal symbols `E_t`, for each original terminal `t` it has. For example, the `Stmts`, `Block` and `RetStmt` rules are updated as:

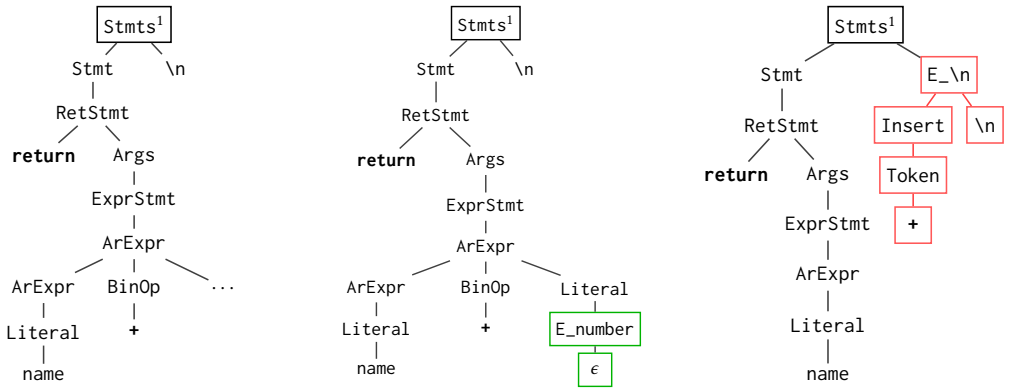
- `Stmts → ... | Stmt E_\n | Stmt E_\n Stmts`
- `Block → ... | E_\n E_indent Stmts E_dedent`
- `RetStmt → ... | E_return | E_return Args`

Next, for each terminal `t` in $G$, we add four error rules of the type:

- `E_t → t | ϵ | Replace | Insert t`

These four new error rules have the following usage for each terminal `t`:

(1) The `E_t → t` rule will match the original terminal `t` without any errors. This error rule is used in cases that the *non-error* version of the rule is needed. For example, in `Block → E_\n E_indent Stmts E_dedent` it can be the case that only `E_dedent` is needed to match the error and `E_\n` and `E_indent` can match their respective symbols.




(a) The partial parse tree for the example at Figure 5a.

(b) Adding a number with the green `E_number` error rule.

(c) Deleting the + with red `E_\n` error rule.

Fig. 9. The rest of the problematic function in Figure 8 and two possible error correcting parses

(2) Using `E_t` $\rightarrow \epsilon$ a *deletion* error is considered. The error rule will match *nothing*, or the *empty token* $\epsilon$, in the program, meaning the terminal is missing.
(3) Using `E_t` $\rightarrow$ `Replace` a *replacement* error is considered. `Replace` will match any terminal token that is *different* than `t`, making a replacement possible.
(4) The rules `E_t` $\rightarrow$ `Insert t` will introduce a *insertion* error, *i.e.* `Insert` will match any *sequence* of `Tokens` that are not supposed to precede `t` in order to make the program parse.

For example, for the terminal tokens **`return`**, `number` and `\n` (a new line) the relevant error production rules are:

- `E_return` $\rightarrow$ **`return`** | $\epsilon$ | `Replace` | `Insert` **`return`**
- `E_number` $\rightarrow$ `number` | $\epsilon$ | `Replace` | `Insert number`
- `E_\n` $\rightarrow$ `\n` | $\epsilon$ | `Replace` | `Insert \n`

Finally, the `Replace` non-terminal can match any possible terminal in $G$ to introduce replacement errors, the `Insert` non-terminal will introduce a sequence of insertion errors by using `Token` which also matches every terminal and we just differentiate the name in order be able to distinguish the different types of errors.

- `Replace` $\rightarrow$ **`return`** | **`pass`** | `\n` | `+` | `...` [**`all`** `terminals`]
- `Insert` $\rightarrow$ `Token` | `Insert Token`
- `Token` $\rightarrow$ **`return`** | **`pass`** | `\n` | `+` | `...` [**`all`** `terminals`]

***ECE Parsing Considerations for Python.*** Unfortunately, we run into various problems if we try to directly use an ECE-Parser for large, real-world languages like Python. Figure 9a presents a *partial parse* of the problematic statements `Stmts`$^1$ of Figure 8. Considering a deletion error (Figure 9b), the `E_number` $\rightarrow \epsilon$ error rule is used to match the empty symbol and generate a parse that suggests that a *number* is missing after the `+` operator. On the other hand, the `E_\n` $\rightarrow$ `Insert \n` error rule can be used to consider a insertion error (Figure 9c) before the new line character, basically *deleting* the `+` operator. In this case, `ArExpr` $\rightarrow$ `Literal` is used to parse the program instead of `ArExpr` $\rightarrow$ `ArExpr BinOp Literal`.

The ECE-Parser is an effective approach on finding minimum distance parses for programs than do not belong in the given programming language, *i.e.* have parse errors. However, this parsing algorithm has limited use in large real-world programming languages, *e.g.* Python or Java, and more time- and memory-efficient parsing algorithms are often used, *e.g.* LR parsing

```
def name(name): \n
indent return name + number \n
dedent \n

def name(name): \n
indent name = name(name) + number \n
return name + \n
dedent end_marker
```

(a) The token sequence generated by the lexer for the program.

```
Stmt \n

def name Params: \n
indent Stmt \n
return Expr BinOp \n
dedent end_marker
```

(b) The abstracted token sequence for the same program.

Fig. 10. The token sequences for the Python program example in Figure 5.

*etc.* [Chapman 1987; Knuth 1965]. For example, Python has 91 *terminal symbols* (including the program's `end_marker`) which means that for all the cases of the error rules `E_t` (excluding the non-error base case `E_t` → `t`), `Replace` and `Token`, 455 *terminal error rules* have to be added to the grammar $G'$. The Python grammar that we used has also 283 *production rules*, from which 182 *rules* have terminals in them, meaning another 182 *error rules* need to be added. Accounting the four helper production rules, *e.g.* for the new start symbol, the new grammar $G'$ has 641 *new error production rules*. This large amount of error rules renders the ECE-Parser not scalable for large programs or programs with a lot of parse errors when using real-world programming languages.

One of our insights, as seen in our running example in Figure 9, is that only a handful of error rules are relevant to each parse error. Therefore, we propose to improve ECE-Parsing's scalability by only adding *a small set* of error production rules, *i.e.* keeping the size of $G'$ limited and only slightly larger than the original grammar $G$. We propose to do so by *training classifiers* to select a small set of error rules only relevant to the parse error. However, the program token sequences that we can use may have irrelevant information, *e.g.* the `foo` function in our example in Figure 5 that does not contribute to the parse error. To address this problem, we propose to further *abstract* our program token sequences.

### 3.2 Abstracting Program Token Sequences

As shown in Figure 6, our neural component has the task of training a classifier to predict the relevant error rules for a given ill-parsed program.

***Problem: Representing Ill-parsed Programs.*** As the inputs are ill-parsed, the training and classification cannot use any form of analysis that requires a syntax tree as input [Gulwani et al. 2018; Martinez et al. 2013; Sakkas et al. 2020; Wang et al. 2018]. One option is to view the ill-parsed program as a plain *sequence of tokens* eliding variable names and such, as shown in Figure 10a. Unfortunately, we found such token sequences yielded inaccurate classifiers that were confused by irrelevant trailing context and predicted rules that were not relevant to repair the error at hand.

***Solution: Abstract with Partial Parses.*** Seq2Parse solves the problem of irrelevant context by *abstracting* the token sequences using *partial parses* to abstract away the irrelevant context. That is, we can use partial parse trees to represent ill-parsed programs as an *abstracted token sequence* shown in Figure 10b, where any *completed* production rules can be used to abstract the relevant *token sub-sequences* with the high-level *non-terminal*.

Figure 8 shows how partial parses can be used to abstract long low-level sequence of tokens into short sequences of non-terminals. (1) The function `foo` is completely parsed, since it had no parse errors and the highest level rule that can be used to abstract it is `Stmt` → `FuncDef`. (2) Similarly, note that `Params` → `( name )` is another completed production rule, therefore the low-level sequence of

```
S         → Stmts end_marker (p = 100.0%)
Stmts     → Stmt \n (p = 38.77%) | Stmt \n Stmts (p = 61.23%)
Stmt      → ExprStmt (p = 62.64%) | RetStmt (p = 7.59%) | ...
RetStmt   → return (p = 1.61%) | return Args (p = 98.39%)
Args      → ExprStmt (p = 99.20%) | ...
ExprStmt  → ArExpr (p = 29.40%) | ...
ArExpr    → Literal (p = 86.89%) | ArExpr BinOp Literal (p = 13.11%)
Literal   → name (p = 64.89%) | number (p = 20.17%) | ...
```

Fig. 11. The production rules shown in Figure 7 with their learned probabilities.

parameter tokens in the `bar` function can be abstracted to just the non-terminal `Params`. (3) However, the production rule for `FuncDef` is *incomplete* since the last statement `Stmt` (under $\texttt{Stmts}^1$) has a parse error as shown in Figure 9a.

***Problem: Ambiguity.*** The generation of this abstraction, however, poses another difficulty. Earley parsing collects a large amount of partial parses (via dynamic programming) until the program is fully parsed. That means at each program location, multiple partial parses can be chosen to abstract our programs. This *ambiguity* can be seen even in the two suggested repairs in Figure 9: if we delete the colored nodes in Figure 9b and Figure 9c we obtain two possible partial parses for our program, the first one matching Figure 9a and the second one not shown here.

***Solution: Probabilistic Context-Free Grammars.*** Seq2Parse solves the ambiguity problem of choosing between multiple possible partial parses via a data-driven approach based on *Probabilistic Context-Free Grammars* which have been used in previous work to select *complete* parses for ambiguous grammars [Collins 2013; Jelinek et al. 1992]. A PCFG associates each of its production rules with a *weight* or *probability*. These weights can be learned [Collins 2013] by using the data set to count the production rules used to parse a number of programs belonging to that language. Seq2Parse uses PCFGs to resolve the ambiguity of partial parses by associating each partial tree (in the Earley table) with a probability which is the *product* of the used rules' probabilities. The tree with the highest probability is selected as a final parse tree which can then be used to generate an abstracted token sequence, as described above.

Figure 11 shows the learned probabilities for the example Python grammar. We observe, for example, that `ReturnStmt` has two possible production rules and almost 98.4% of the times a **`return`** is followed by an argument list. Additionally, 62.6% of the times a `Stmt` is an `ExprStmt` and only 7.6% of the times it is a `RetStmt`. Thus, in our example, the probability that would be assigned to the partial parse for $\texttt{Stmts}^1$ in Figure 9b (only the sub-tree without the colored error nodes) is the product of the probabilities of the production rules `Stmts` → `Stmt \n`, `Stmt` → `RetStmt`, `RetStmt` → **`return`** `Args`, `Args` → `ExprStmt` *etc.* which is $38.77\% \cdot 7.59\% \cdot 98.39\% \cdot 99.20\% \cdot \dots = 4.57‰$, while the partial parse for $\texttt{Stmts}^1$ in Figure 9c would similarly be calculated as 47.61‰, making it the proper choice for the abstraction of the program.

### 3.3 Training Sequence Classifiers

The abstracted token sequences we extracted from the partial parses present us with short abstracted sequences that abstract irrelevant details of the context into non-terminals. Next, Seq2Parse uses the NLP approach of *sequence models* [Hardalov et al. 2018; Sutskever et al. 2014] to (use the abstract token sequences) to train a classifier that can predict the relevant error rules.

***Seq2Seq Architectures.*** Sequence-to-sequence (seq2seq) architectures transform an input sequence of tokens into a new sequence [Sutskever et al. 2014] and consist of an *encoder* and a *decoder*. The

encoder transforms the input sequence into an *abstract vector* that captures all the essence and context of the input. This vector does not necessarily have a physical meaning and is instead an internal representation of the input sequence into a higher dimensional space. The abstract vector is given as an input to the decoder, which in turn transforms it into an output sequence.

Seq2Parse uses a *sequence classifier* that can correctly predict a small set of relevant error production rules for a given abstracted token sequence. We use a *transformer encoder* [Vaswani et al. 2017] to encode the input sequences into abstract vectors that we then feed into a *DNN* classifier to train and make accurate predictions [Schmidhuber 2015].

***Training From a Dataset.*** Given a dataset of fixed parse errors, such as Figure 5, we extract the small set of relevant error rules needed for each program to make it parse with an ECE-Parser. Running the ECE-Parser on every program in the dataset with the *full set* of error production rules is prohibitively slow. Therefore, we extract the erroneous and fixed program token-level differences or *token diffs* and map them to *terminal error production rules*. The non-terminal error rules can be inferred using the grammar and the terminal rules. Next, we run the ECE-Parser with the extracted error rules to confirm which ones would make the program parse and assign them as *labels*.

For example, the diff for the program pair in Figure 5 would show the deleted + operator, thus extracting the error rules `Token` $\rightarrow$ `+` and `E_\n` $\rightarrow$ `Insert \n`, since the extra + precedes a newline character `\n`. Similarly, if a token `t` is added in the fixed program, the error rule `E_t` $\rightarrow \epsilon$ is added and if a token `t` replaces a token `a`, the error rules `E_t` $\rightarrow$ `Replace` and `Replace` $\rightarrow$ `a` are added.

### 3.4 Predicting Error Rules with Sequence Classifiers

The learned sequence classifier model, which has be trained on the updated error-rule-labeled data set can now be used to predict the relevant rules for new erroneous programs. Additionally, neural networks have the advantage of associating each class with a *confidence score* that can be used to rank error rule predictions for new programs, letting us select the top-$N$ ones that will yield accurate repairs when used with the ECE-Parser.

For our running example, in Figure 5a, we abstract the program as shown in Figure 10b and then we predict the error production rules for it with the trained sequence classifier. We rank the full set of *terminal* error rules based on their predicted confidence score from the classifier and return the *top 10* predictions for our example. Therefore, the predicted set of error rules is the following: `E_number` $\rightarrow \epsilon$, `E_number` $\rightarrow$ `Insert number`, `E_\n` $\rightarrow$ `Insert \n`, `E_(` $\rightarrow \epsilon$, `E_return` $\rightarrow \epsilon$, `Token` $\rightarrow$ `)`, `Token` $\rightarrow$ `+`, `Token` $\rightarrow$ `:`, `Token` $\rightarrow$ `name`, `Token` $\rightarrow$ `number`.

The classifier predicts mostly relevant error rules such as the ones that use `E_number`, `E_\n` and `E_return` for example, as we showed previously. There are also rules that are not very relevant to this parse error but the classifier predicts probably due to them being common parse errors, *e.g.* `Token` $\rightarrow$ `)`, `Token` $\rightarrow$ `:`. Finally, we added the *non-terminal* error rules needed to introduce these errors, which can be inferred by them. For example, we can infer `Stmts` $\rightarrow$ `Stmt E_\n`, `Stmts` $\rightarrow$ `Stmt E_\n Stmts` and `Block` $\rightarrow$ `E_\n indent Stmts dedent` from `E_\n` (we don't need `E_indent` or `E_dedent` here since no such terminal error rules were predicted).

We then parse the program in Figure 5a with the ECE-Parser and these specific error rules to generate a valid parse. We observe that it takes our implementation (as we show later in depth) less than *2 seconds* to generate a valid parse, which is also the one that leads to the user repair in Figure 5b! On the other hand, when we use a baseline ECE-Parser with the full set of error rules it takes *2 minutes and 55 seconds* to generate a valid parse, which is, however, not the expected user parse but the one shown in Figure 9b. These examples demonstrate the effectiveness of accurately *predicting error rules using sequence classifiers, which are trained on abstracted token sequences.*

| | | |
|---|---|---|
| $G$ | $\doteq$ | $(N, \Sigma, P, S)$ |
| PCFG | $\doteq$ | $(N, \Sigma, P, S, W)$ |
| $G'$ | $\doteq$ | $(N', \Sigma, P', S')$ |
| $P'$ | $\doteq$ | $P \cup$ ErrorRules |
| $N'$ | $\doteq$ | $N \cup \{S', H, I\} \cup \{E_a \mid a \in \Sigma\}$ |
| $e$ | $\in$ | $L(G)$ |
| $e_\perp$ | $\notin$ | $L(G)$ |
| Pair | $\doteq$ | $e_\perp \times e$ |
| DataSet | $\doteq$ | [Pair] |

| | | | |
|---|---|---|---|
| § 4 | **learnPCFG** | : | $G \to [e] \to$ PCFG |
| | **partialParse** | : | PCFG $\to e_\perp \to t^a$ |
| § 5 | **trainDL** | : | $[t^a \times$ ErrorRules$] \to$ DLModel |
| | **predictDL** | : | DLModel $\to t^a \to$ ErrorRules |
| § 6 | **diffRules** | : | Pair $\to$ ErrorRules |
| | **ECEParse** | : | ErrorRules $\to e_\perp \to e$ |
| | **train** | : | DataSet $\to$ DLModel |
| | **predict** | : | DLModel $\to G \to e_\perp \to$ ErrorRules |
| | **Seq2Parse** | : | $G \to$ DataSet $\to (e_\perp \to e)$ |

Fig. 12. A high-level API of the Seq2Parse system that learns to repair syntax errors.

In the next three sections, we describe in depth the specifics of our approach by defining all the methods in Figure 12. We start by presenting the program abstraction (section 4) using partial parses and a learnt PCFG, we then explain how we train sequence models for making error rule predictions (section 5) and, finally, we demonstrate our algorithms for building Seq2Parse (section 6), an approach for efficiently parsing erroneous programs.

## 4 ABSTRACTING PROGRAMS WITH PARSE ERRORS

We introduce here our approach for abstracting programs with parse errors into a suitable sequence of tokens for training sequence classifiers. We explain how a traditional Earley parser can be used to extract partial parses with the aid of a Probabilistic Context-Free Grammar (PCFG), in order to get a higher level of abstraction with richer context information than the simple Lexer output.

***Lexical Analysis.*** Historically, *lexical analysis*, lexing or tokenization is the process of converting a sequence of characters *i.e.* a program into a sequence of tokens (strings with an assigned and thus identified meaning). The program that performs the lexical analysis is called a *lexer* and is usually combined with a parser, which together analyze the syntax of a programming language $L(G)$, defined by the grammar $G$. When a program has a syntax error, the output token sequence of the lexer is the highest level of abstraction that we can acquire, since the parser fails and returns an error, meaning that we have no available parse trees.

***Token Sequences.*** Our goal is to parse a *program token sequence* $t^i$, which is a lexed program with parse errors (*i.e.* $t^i \notin L(G)$), and repair it into a *fixed token sequence* $t^o \in L(G)$ that can be used to return a repaired program without syntax errors. Let $t^i$ be a sequence $t^i_1, t^i_2, \ldots, t^i_n$ and $t^o$ be the updated sequence $t^o_1, t^o_2, \ldots, t^o_i, \ldots, t^o_j, \ldots, t^o_k$. The subsequence $t^o_i, \ldots, t^o_j$ can either *replace* a subsequence in $t^i$, it can *be inserted* in $t^i$ or can be the empty subsequence $\epsilon$ and essentially *delete* a subsequence in $t^i$ to generate the $t^o$. It can be the whole program, part of it or multiple parts of it. $t^o$ will finally be a token sequence that can be parsed by the original language's $L(G)$ parser.

However, programs can be large and, thus, $n$ can take large values, which makes them unsuitable for training effectively sequence models. Therefore, our goal is to first generate an *abstracted token sequence* $t^a$ that removes all irrelevant information from $t^i$ and gives hints for the parse error fix by using the internal states of an *Earley* parser.

### 4.1 Earley Partial Parses

We propose using an *Earley parser* to generate the abstracted token sequence $t^a$ for an input program sequence $t^i$. An Earley parser holds internally a *chart* data structure, *i.e. a list of partial parses*. Given a production rule $X \to \alpha\beta$, the notation $X \to \alpha \cdot \beta$ represents a condition in which $\alpha$

has already been parsed and $\beta$ is expected and both are sequences of terminal and non-terminal symbols (tokens).

Each state is a tuple $(X \rightarrow \alpha \cdot \beta,\ j)$, consisting of

- the production rule currently being matched $(X \rightarrow \alpha\beta)$
- the current position in that production (represented by the dot $\cdot$)
- the origin position $j$ in the input at which the matching of this production began

We denote the state set at an input position $k$ as $S(k)$. The parser is seeded with $S(0)$ consisting of only the top-level rule $S \rightarrow \gamma$. It then repeatedly executes three operations: *prediction, scanning,* and *completion.* There exists a *complete parse* if the complete top-level rule $(S \rightarrow \gamma\cdot, 0)$ is found in $S(n)$, where $n$ the input length. We define a *partial parse* to be any partially completed rules, *i.e.* if there is $(X \rightarrow \alpha \cdot \beta, i)$ in some state $S(k)$, where $i < k \leq n$.

Let, again, $t^i_1, t^i_2, \ldots, t^i_j, \ldots, t^i_k, \ldots, t^i_n$ be the input token sequence $t^i$, where at location $k$ there is a parse error and the parser has exhausted all possibilities and can not add any more rules in state $S(k+1)$, *i.e.* $S(k+1) = \emptyset$. We want to abstract $t^i$ by getting the longest possible part of the program that has a partial parse, *i.e.* by finding the largest $j$ for which there is a rule $(X \rightarrow \alpha \cdot \beta, 0) \in S(j)$. We use this rule for $X$ to replace $t^i_1, t^i_2, \ldots, t^i_j$ in $t^i$ with $\alpha$, thus getting an abstracted sequence $t^a$. In the same manner, we use the longest possible partial parses that we can extract from the chart to abstract $t^i_{j+1}, \ldots, t^i_k$, until we reach the parse error at location $k$.

However, each of the states $S(j),\ 0 \leq j \leq k$, holds a large number of partial parses and, thus, our heuristic to choose the longest possible partial parse to abstract programs may not be able to abstract the token sequence fully until the error location $k$. Additionally, there may be two or more partial parses in $S(k)$, with different lengths, *e.g.* $\{(X \rightarrow \alpha \cdot \beta,\ j),\ (X' \rightarrow \alpha' \cdot \beta',\ h)\} \in S(k),\ j \neq h$. We propose selecting the most *probable parse* with the aid of a PCFG.

### 4.2 Probabilistic Context-Free Grammars

We learn a PCFG from a large corpus of programs $[e], e \in L(G)$, that belong to a language $L(G)$, that a grammar $G$ defines, with the **learnPCFG** procedure as shown in Figure 12. We use the learned PCFG with an augmented Earley parser in **partialParse** to abstract a program $e_\perp$ into a abstract token sequence $t^a$.

A PCFG can be defined similarly to a *context-free grammar* $G \doteq (N,\ \Sigma,\ P,\ S)$ as a quintuple $(N,\ \Sigma,\ P,\ S,\ W)$, where:

- $N$ and $\Sigma$ are finite disjoint alphabets of non-terminals and terminals, respectively.
- $P$ is a finite set of production rules of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in (N \cup \Sigma)^*$.
- $S$ is a distinguished start symbol in $N$.
- $W$ is a finite set of probabilities $p(X \rightarrow \alpha)$ on production rules.

Given a dataset of programs $[e], e \in L(G)$ that can be parsed, let $count(X \rightarrow \alpha)$ be the number of times the production rule $X \rightarrow \alpha$ has been used to generate a final complete parse, while parsing $[e]$, and $count(X)$ be the number of times the non-terminal $X$ has been seen in the left side of a used production rule. The probability for a production rule $X \rightarrow \alpha$ is then defined as:

$$p(X \rightarrow \alpha) = \frac{count(X \rightarrow \alpha)}{count(X)}$$

**learnPCFG** invokes an instrumented Earley parser to calculate all the values $count(X \rightarrow \alpha), \forall X \rightarrow \alpha : P$ and $count(X), \forall X : N$. The *instrumented parser* keeps a *global record* of these values, while parsing the dataset $[e]$ of programs. Finally, **learnPCFG** outputs a PCFG that is based on the original grammar $G$ that was used to parse the dataset with the learned probabilities $W$.

### 4.3 Abstracted Token Sequences

Given a program $e_\perp$ with a parse error and a learned PCFG, **partialParse** will generate an abstracted token sequence $t^a$. The PCFG will be used with an *augmented Earley parser* to disambiguate partial parses and choose one, in order to produce an abstracted token sequence as described in § 4.1.

We augment Earley states $(X \rightarrow \alpha \cdot \beta,\ j)$ to $(X \rightarrow \alpha \cdot \beta,\ j,\ p)$, where $p$ is the probability that $X \rightarrow \alpha \cdot \beta$ is a correct partial parse. When there are two (or more) conflicting partial parses $\{(X \rightarrow \alpha \cdot \beta,\ j,\ p),\ (X' \rightarrow \alpha' \cdot \beta',\ h,\ p')\} \in S(k)$, the augmented parser selects the partial parse with the highest probability $max(p,\ p')$. The augmented parser calculates the probability $p$ for a partial parse $(X \rightarrow \alpha \cdot \beta,\ j,\ p)$ in the state $S(k)$, as the product $p_1 \cdot p_2 \cdot \cdots \cdot p_{k-1}$ of the probabilities $p_1,\ p_2,\ \ldots,\ p_{k-1}$ that are associated with the production rules $(X_1 \rightarrow \alpha_1 \cdot \beta_1,\ i_1,\ p_1), (X_2 \rightarrow \alpha_2 \cdot \beta_2,\ i_2,\ p_2), \ldots$ that have been used so far to parse the string of tokens $\alpha$.

## 5 TRAINING SEQUENCE CLASSIFIERS

Our next task is to *train* a model that can predict the error production rules that are needed to parse a given program $e_\perp$ (with syntax errors) according to a given grammar $G$, by using its (abstracted) program token sequence $t^a$. We define the function **predictDL** which takes as input a *pre-trained sequence classifier* DLModel and an abstracted token sequence $t^a$ and returns as output a *small subset* of ErrorRules. We train the DLModel offline with the **trainDL** method with a dataset $[t^a \times \text{ErrorRules}]$ of token sequences $t^a$ and the *exact small set* of error production rules ErrorRules that the ECE-Parser used to generate the *user parse*. We build our classifier DLModel using classic *Deep Neural Networks (DNNs)* and parts of state-of-the-art *Sequence-to-Sequence (seq2seq)* models. We leave the high level details of acquiring the dataset of labeled token sequences and using the predictor for new erroneous programs for section 6. In the next few paragraphs, we summarize the recent advances in machine learning that help as build the sequence classifier.

We encode the task of learning a function that will map token sequences of erroneous programs to a small set of error production rules as a *supervised multi-class classification (MCC)* problem. A *supervised* learning problem is one where, given a labeled training set, the task is to learn a function that accurately maps the inputs to output labels and generalizes to future inputs. In a *classification* problem, the function we are trying to learn maps inputs to a discrete set of two or more output labels, called *classes*. We use a *Transformer encoder* to encode the input sequences into abstract vectors that we then directly feed into a *DNN* classifier to build a *Transformer classifier*.

***Neural Networks.*** A neural network can be represented as a directed acyclic graph whose nodes are arranged in layers that are fully connected by weighted edges. The first layer corresponds to the input features, and the final to the output. The output of an internal node is the sum of the weighted outputs of the previous layer passed to a non-linear *activation function*, which in recent work is commonly chosen to be the rectified linear unit (ReLU) [Nair and Hinton 2010]. In this work, we use relatively *deep neural networks* (DNN) that have proven to make more accurate predictions in recent work [Schmidhuber 2015]. A thorough introduction to neural networks is beyond the scope of this work [Hastie et al. 2009; Nielsen 2015].

***Sequence Models.*** *Seq2seq* models aim to transform input sequences of one domain into sequences of another domain [Sutskever et al. 2014]. In the general case, these models consist of two major layers, an *encoder* and a *decoder*. The encoder transforms an input token sequence $x_1, x_2, \ldots, x_n$ into a *abstract vector* $V \in \mathbb{R}^k$ that captures all the essence and context of the input sequence. This vector does not necessarily have some physical meaning and is just an internal representation of the input sequence into a higher dimensional space. The abstract vector is then given as an input to the decoder, which in turn transforms it into an output sequence $y_1, y_2, \ldots, y_n$.

The simplest approach historically uses a Recurrent Neural Network (RNN) [Rumelhart et al. 1986; Werbos 1990], which is a natural next step from the classic neural networks. Each RNN unit operates on each input token $x_t$ separately. It keeps an internal *hidden state* $h_t$ that is calculated as a function of the input token $x_t$ and the previous hidden state $h_{t-1}$. The output $y_t$ is calculated as the product of the current hidden state $h_t$ and an output weight matrix. The activation function is usually chosen as the standard *softmax* function [Bishop 2006; Goodfellow et al. 2016]. Softmax assigns probabilities to each output that must add up to 1. Finally, the loss function at all steps of the RNN is typically calculated as the sum of the cross-entropy loss of each step.

***Transformers.*** The Transformer is an DNN architecture that deviates from the recurrent pattern (*e.g.*, RNNs) and is solely relying on *attention mechanisms*. Attention has been of interest lately [Bahdanau et al. 2015; Kim et al. 2017; Vaswani et al. 2017] mainly due to its ability to detect dependencies in the input or output sequences regardless the distance of the tokens. The nature of this architecture makes the Transformer significantly easier to parallelize and thus has a higher quality of predictions and sequence translations after a shorter training period.

The novel architecture of a Transformer [Vaswani et al. 2017] is structured as a *stack of N identical layers*. Each layer has two main sub-layers. The first is a *multi-head self-attention mechanism*, and the second is a position-wise fully connected neural network. The output of each sub-layer is $LayerNorm(x + SubLayer(x))$, where $SubLayer(x)$ is the function implemented by each sub-layer, followed by a residual connection around each of the two sub-layers and by layer normalization $LayerNorm(x)$. To facilitate these residual connections, all sub-layers in the model, as well as the input *embedding layers*, produce outputs of the same dimension $d_{model}$.

***Transformer Classifier.*** For our task, we choose to structure DLModel as a *Transformer Classifier*. We use a novel Transformer encoder to represent an abstracted token sequence $t^a$ into an abstract vector $V \in \mathbb{R}^k$. The abstract vector $V$ is then fed as input into a multi-class DNN. We use **trainDL** to train the DLModel given the training set $[t^a \times \text{ErrorRules}]$. The binary cross-entropy loss function is used per class to assign the loss per training cycle. DLModel predicts error production rules for a new input program $t^a$. Critically, we require that the classifier outputs *confidence scores* $C$ that measure how sure the classifier is that a rule can be used to parse the associated input program $e_\perp$. The **predictDL** function uses the trained DLModel to predict the confidence scores $[\text{ErrorRules} \times C]$ for all error production rules ErrorRules for a new unknown program $e_\perp$ with syntax errors. The ErrorRules are then sorted based on their predicted confidence score $C$ and finally the *top-N* rules are returned for error-correcting parsing. $N$ is a small number in the 10s that will give accurate predictions without making the ECE-Parser too slow, as we discuss in section 7.

## 6 BUILDING A FAST ERROR CORRECTING PARSER

We show how **Seq2Parse** uses the abstracted token sequences from section 4 and the trained sequence models from section 5 to generate an *error-correcting parser* $(e_\perp \rightarrow e)$, that will parse an input program $e_\perp$ with syntax errors and produce a correct program $e$. We first describe how we extract a machine-learning-amenable training set from a corpus of fixed programs and finally how we structure everything to train our model.

### 6.1 Learning Error Production Rules

The **trainDL** method requires a dataset of token sequences $t^a$ that is annotated with an *exact and small set* of error production rules, *i.e.* $[t^a \times \text{ErrorRules}]$. These ErrorRules are just a subset of all the possible error rules that are needed to parse and fix $t^a$. The straight-forward approach is to use **ECEParse** with all possible error production rules for each program $e_\perp$ in the dataset. Then, when **ECEParse** returns with a successful parse, we extract the rules that where used to parse the

**Algorithm 1** Training **Seq2Parse**'s model DLModel

**Input:** Probabilistic Grammar $G$, DataSet $Ds$
**Output:** Classifier $Model$

**procedure** TRAIN($G$, $Ds$)
  $D_{ML} \leftarrow \emptyset$
  **for all** $p_{err} \times p_{fix} \in Ds$ **do**
    $t^a \leftarrow$ PARTIALPARSE($G$, $p_{err}$)
    $rules \leftarrow$ DIFFRULES($p_{err} \times p_{fix}$)
    $D_{ML} \leftarrow D_{ML} \cup (t_a \times rules)$
  $Model \leftarrow$ TRAINDL($D_{ML}$)
  **return** $Model$

**Algorithm 2** Predicting error production rules with **Seq2Parse**'s model DLModel

**Input:** Classifier $Model$, Probabilistic Grammar $G$, Program $P$
**Output:** Error Production Rules $Rls$

**procedure** PREDICT($Model$, $G$, $P$)
  $t^a \leftarrow$ PARTIALPARSE($G$, $P$)
  $Rls \leftarrow$ PREDICTDL($Model$, $t^a$)
  **return** $Rls$

program $e_\perp$. This approach generates a dataset with the smallest possible set of error rules as labels per program, since the original ECE-Parser returns the minimum-distance edit parse. However, this approach completely ignores the programmer's fix and takes an unreasonable amount of time to parse a dataset with millions of programs, due to the inefficient nature of the ECE-Parser.

We suggest using an $O(ND)$ difference algorithm [Myers 1986] to get a small but still representative set of error production rules for each program $e_\perp$. We employ this algorithm to find the differences between the input *program token sequence* $t^i$, which is the lexed program $e_\perp$ and the *fixed token sequence* $t^o$, which is the lexed program $e$. This algorithm returns changes between token sequences in the form of *inserted or deleted tokens*. It is possible that this algorithm returns a sequence of deletions followed by a sequence of insertions, which can in turn be interpreted as a *replacement* of tokens. We map these three types of changes to the respective error production rules. Let $t^i$ be a sequence $t^i_1, t^i_2, \ldots, t^i_n$ and $t^o$ be the updated sequence $t^o_1, t^o_2, \ldots, t^o_m$. We map:

- an inserted output token $t^o_j$ to a *deletion* error $E_{t^o_j} \longrightarrow \epsilon$.
- a deleted input token $t^i_k$ to an *insertion* error $Tok \longrightarrow t^i_k$ and the helper rule $E_{t^i_{k+1}} \longrightarrow Ins\ t^i_{k+1}$.
- a replaced token $t^i_k$ with $t^o_j$ to a *replacement* error $Repl \longrightarrow t^i_k$ and the helper rule $E_{t^o_j} \longrightarrow Repl$.

In the case of an insertion error, we also include the helper rules $Ins \longrightarrow Tok$ and $Ins \longrightarrow Ins\ Tok$, that can derive any nonempty sequence of insertions. To introduce (possible) insertion errors at the end of a program, we include the starting production rules $S' \longrightarrow S$ and $S' \longrightarrow S\ Ins$.

The above algorithm, so far, adds only the *terminal error productions*. We have to include the *non-terminal error productions* that will invoke the terminal ones. If $X \longrightarrow a_0 b_0 a_1 b_1 \ldots a_m b_m,\ m \geq 0$, is a production in $P$ such that $a_i$ is in $N^*$ and $b_i$ is in $\Sigma$, then we add the error production $X \longrightarrow a_0 X_{b_0} a_1 X_{b_1} \ldots a_m X_{b_m},\ m \geq 0$ to $P'$, where each $X_{b_i}$, is either a new non-terminal $E_{b_i}$ that was added with the previous algorithm, or just $b_i$ again if it was not added.

Finally, we further refine the new small set of error productions for each program $e_\perp$ with ECE-Parser, in order to create the final annotated dataset $[t^a \times \text{ErrorRules}]$. The changes that we extracted from the programmers' fixes might include irrelevant changes to the parse error fix, *e.g.* code clean-up. Therefore, filtering with the ECE-Parser is still essential to annotate each program with the appropriate error production rules. We implement this error-rule-extracting approach in the function **diffRules**, which extracts the token differences between an erroneous program $e_\perp$ and a fixed program $e$ and returns the appropriate error production rules.

**Algorithm 3** Generating the final ECEP

**Input:** Grammar $G$, DataSet $Ds$
**Output:** Error Correcting Parser $Prs$

```
1: procedure Seq2Parse(G, Ds)
2:     ps ← Map(λ.p → Snd(p), Ds)
3:     PCFG ← LearnPCFG(G, ps)
4:     Model ← Train(PCFG, Ds)
5:     ERulePredictor ← Predict(Model, PCFG)
6:     Prs ← (λ.p_err → ECEParse(ERulePredictor(p_err), p_err))
7:     return Prs
```

### 6.2 Training and Using a Transformer Classifier

Given a (probabilistic) grammar $G$ and a dataset $Ds$, Algorithm 1 extracts a machine-learning appropriate dataset $D_{ML}$ in order to **train** a Transformer classifier $Model$ with **trainDL**. The classifier $Model$ can then be used to predict error rules for new erroneous programs $p_{err}$.

The dataset $D_{ML}$ starts as an empty set. For each program pair $p_{err} \times p_{fix}$, we, first, employ **partialParse** with the PCFG $G$ and an erroneous program $p_{err}$ to extract the abstracted token sequence $t^a$. Second, we use the token difference algorithm **diffRules** to extract the specific error *rules* that fix $p_{err}$ based on $p_{fix}$. The abstracted sequence $t^a$ is annotated with the label *rules* and is added to $D_{ML}$. The Transformer classifier $Model$ is trained with **trainDL** and the newly extracted dataset $D_{ML}$, which is finally returned by the algorithm. Finally, the **train**ing procedure can be performed offline and thus won't affect the performance of the final program repair.

Having trained the Transformer classifier $Model$, we can now predict error rules $Rls$, that will be used by an ECE-Parser, by using the **predict** procedure defined in Algorithm 2. **predict** uses the same input grammar $G$ to generate an abstracted token sequence $t^a$ for the program $P$ with the **partialParse** procedure. Finally, the **predictDL** procedure predicts a small set of error production rules $Rls$ for the sequence $t^a$ given the pre-trained $Model$.

### 6.3 Generating an Efficient Error Correcting Parser

Algorithm 3 presents our *neurosymbolic* approach, **Seq2Parse**. This is the high-level algorithm that combines everything that we described so far in the last three sections. **Seq2Parse** first extracts the fixed programs $ps$ from the dataset $Ds$ to learn a probabilistic context-free grammar $PCFG$ for the input grammar $G$ with **learnPCFG**. It then **train**s the Transformer classifier $Model$ to predict error production rules. We define an error rule predictor, ERulePredictor, using the **predict** procedure with the pre-trained $Model$ and grammar $PCFG$. Finally, the algorithm returns the ECE-Parser $Prs$, which we define as a function that takes as input an erroneous program $p_{err}$ that uses the ERulePredictor to get the set of error rules needed by **ECEParse** to parse and repair it.

## 7 EVALUATION

We have implemented our approach in Seq2Parse: a system for repairing parse errors for Python at its entirety. Next, we describe our implementation and an evaluation that addresses four questions:

- **RQ1**: How *accurate* are Seq2Parse's predicted error production rules? (§ 7.1)
- **RQ2**: How *precisely* can Seq2Parse repair parse errors? (§ 7.2)
- **RQ3**: How *efficiently* can Seq2Parse repair parse errors? (§ 7.3)
- **RQ4**: How *useful* are Seq2Parse's suggested repairs? (§ 7.4)

***Training Dataset.*** For our evaluation, we use the same Python dataset that we used in our error data analysis in section 2 gathered from PythonTutor.com [Guo 2013] between the years 2017 and

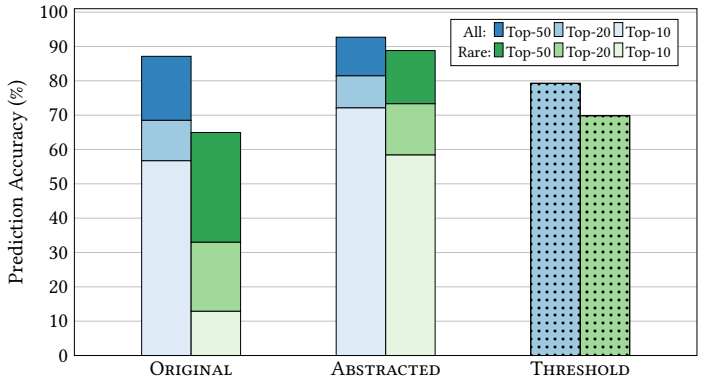



Fig. 13. Results of our error production rule prediction classifiers for the simple original token sequences and their abstracted versions using the PCFG.

2018. The dataset has more than 1,100,000 usable erroneous Python programs and their respective fixes. The programs have an average length of *87 tokens*, while the abstracted token sequences have a much shorter average of *43 tokens*. We choose 15,000 random programs from the dataset for all our tests, and the rest we use as our training set.

We first learn a PCFG on the training set of fixed programs to learn the probabilities for each production rule in the *full Python grammar*. Seq2Parse then extracts the abstracted token sequences for all programs in the training set. Next, while the full Python grammar has *455 possible terminal error production rules*, in reality, only *340 error rules* are ever used in our dataset and are assigned as *labels*. We arrive at this set of error rules by parsing all the erroneous programs in the training set with the ECE-Parser and the "diff" error rules, as described in subsection 6.1.

***Transformer Classifier.*** Seq2Parse's error rule prediction uses a Transformer classifier with *six* transformer blocks, that each has a fully-connected hidden layer of 256 neurons and 12 attention heads. The output of the transformer blocks is then connected to a DNN based classifier with *two* fully-connected hidden layers of 256 and 128 neurons respectively. The neurons use rectified linear units (ReLU) as their activation function, while the output layer uses the sigmoid function for each class. Additionally, there are *two input embedding layers* of a length of 128 units, one for input tokens and one for their positions in the sequence. We also limit the input abstracted token sequences to a length of 128 tokens, which covers 95.7% of the training set, without the need of pruning them. Finally, the Transformer classifier was trained using an Adam optimizer [Kingma and Ba 2014], a variant of stochastic gradient descent, for a total of 50 epochs.

### 7.1 RQ1: Accuracy

Figure 13 shows the accuracy results of our error production rule prediction experiments. The y-axis describes the *prediction accuracy*, *i.e.* the fraction of test programs for which the correct *full set* of error rules to repair the program was predicted in the top-K sorted rules. The Original version of our transformer classifier does not consider the abstracted token sequences and used the full Original token sequences, whose results are presented in the first two bars of Figure 13. The next two bars show our final results using the Abstracted token sequences to train the classifier. Finally, the last two dotted bars show the results for when a probability Threshold is set in order to select the predicted error rules (instead of picking the static top-K ones) but using again the Abstracted sequences as input. The predicted error rule set ranges between 1–20 elements.

The blue bars show the accuracy on the full test set of All 15,000 test programs, while the green bars show the results on a subset of Rare programs, *i.e.* programs that did not include any of the 50 most popular error rules. The Rare programs amount only for 4% of our test set.

The Original predictor, even with the Top-50 predicted error rules, is less accurate than the Top-20 predictions of the Abstracted, with an accuracy of 87.13%, which drops to 68.48% and 56.71% respectively for the Top-20 and Top-10 predictions. The Abstracted predictor significantly outperforms the Original predictor with a 72.11% Top-10 accuracy, 81.45% Top-20 accuracy and 92.70% Top-50 accuracy.

The Threshold predictions are almost as accurate as the Abstracted Top-20 predictions with an accuracy of 79.28% and a median number of selected error rules of 14 (average 14.1). This could potentially mean that this predictor is a valid alternative for the static Top-20 predictions.

Finally, we observe that our Abstracted classifiers generalize efficiently for our dataset of erroneous Python programs and is almost as accurate for the Rare programs as the rest of the dataset with a 73.32% Top-20 accuracy (88.81% Top-50 accuracy). The same holds for the Threshold predictions with a 69.83% Rare accuracy.

> Seq2Parse's transformer classifier learns to encode programs with syntax errors and select candidate error production rules for them effectively, yielding *high accuracies*. By abstracting the tokens sequences, Seq2Parse is able to *generalize* better and make more accurate predictions with a *81.45% Top-20 accuracy*.

### 7.2 RQ2: Repaired Program Preciseness

Next we evaluate Seq2Parse's end-to-end accuracy and preciseness in generating Seq2Parse's parsing time to *5 mins* and run our experiments on the 15,000-program test set. Additionally, we use here the highest-performing transformer classifiers, *i.e.* the Abstracted and Threshold classifiers.

We compare *three versions* of our Seq2Parse implementation (AllParses, MinimumCost and Threshold) against two versions of the ECE-Parser with a static selection of the 20 and 50 most popular error production rules in our training set. We make this choice because we observe that the 50 most popular error rules are used as labels for as much as *86%* of the training set.

Our AllParses and MinimumCost ECE-Parsers both use the *Top-20 predictions* from our Abstracted classifier to parse and repair buggy programs. The AllParses ECE-Parser keeps internally *all possible states* that arise from using the predicted error rules similarly to the original ECE-Parser described by Aho and Peterson [1972]. We use a maximum repair cost of 2 edits (*i.e.* a maximum of 2 insertions, deletions or replacements) to limit the search space. The MinimumCost version however keeps always the minimum-edit repair and discards all other states that may lead to a higher cost. This allows for a higher maximum cost of 10 edits. We use the same ECE-Parser and cost as in MinimumCost for our Threshold parser. Finally, while AllParses can generate a large number of repairs, we keep only the top 5 repairs after filtering with a static code checker (Pylint, https://www.pylint.org/) as most developers will consider only a few suggestions before falling back to manual debugging [Kochhar et al. 2016; Parnin and Orso 2011].

Figure 14 shows the percentage of test programs that each of these four versions can parse successfully (*i.e.* the *parse accuracy*), the rare programs parse accuracy, and the *user equivalent parse accuracy*, *i.e.* the amount of parses that match the one that the user compiled. We observe that the MinimumCost parser *outperforms* every other option with 94.25% parse accuracy and 94.01% rare parse accuracy. It also generates the intended user parse for 20.55% of the set, *i.e.* over 1 out 5 of the cases. The 20 most popular parser with 79.87% parse accuracy and 65.01% rare parse accuracy is much less accurate, and is 4.24% less likely to generate the user parse, while the 50 most

| Error Rule Approach | *Parse Accuracy* | *Rare Parse Accuracy* | *User Fix Accuracy* |
|---|---|---|---|
| 20 Most Popular | 79.87% | 65.01% | 16.31% |
| 50 Most Popular | 90.89% | 81.26% | 18.56% |
| AllParses | 61.82% | 72.22% | **33.92%** |
| MinimumCost | **94.25%** | **94.01%** | 20.55% |
| Threshold | 94.19% | 93.42% | 21.19% |

Fig. 14. Experimental results of Seq2Parse's repair approaches.

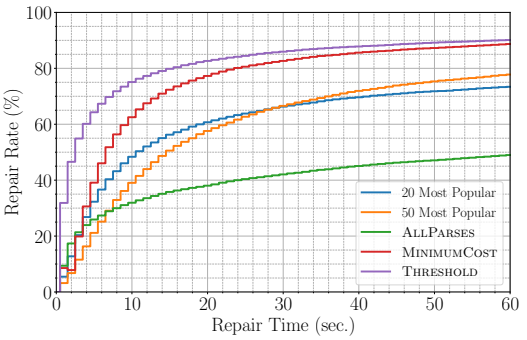



Fig. 15. The repair rate for all the approaches in Figure 14

popular is slightly less accurate with 90.89% and 81.26% accuracy, as expected from the usage of a large number of popular error rules. The 50 most popular parser has also a high user fix accuracy of 18.56%. The AllParses parser has the lowest parse accuracy of 61.82%, however it manages to generate the user fix 33.92% of the time and achieve a high 72.22% rare accuracy. Finally, the Threshold parser is almost as accurate as the efficient MinimumCost parser with 94.19% and 93.42% parse and rare accuracy, while achieving a slightly higher user fix accuracy of 21.19%.

> Seq2Parse can *parse and repair 94.25%* of programs with syntax errors. In addition, it generates *the exact user fix over 20% of the time.*

### 7.3 RQ3: Efficiency

Next we evaluate Seq2Parse's efficiency by measuring how many programs it is able to parse. We limit each ECE-Parser to 5 minutes. (In general, the procedure is undecidable, and we conjecture that a longer timeout will diminish the practical usability for developers.) We compare the efficiency of Seq2Parse for all the versions of Figure 14.

Figure 15 shows the cumulative distribution function of all Seq2Parse approaches' repair rates over their repair time. We observe that using Threshold predictions with the MinimumCost ECE-Parser is the most efficient and it maintains the highest parse accuracy at all times, with a repair rate of 83.04% within 20 seconds and a median parse time of 2.1 seconds. The MinimumCost with the top 20 error rule predictions is still very efficient with a repair rate of 78.10% within 20 seconds and a median parse time of 5.3 seconds.

We observe that, using a fixed set of the 20 and 50 most popular rules, Seq2Parse (with the MinimumCost ECE-Parser) repairs 61.41% and 58.61% of the programs respectively within 20 seconds, and has median parse times of 7.0 and 13.6 seconds respectively. The 50 most popular rules admit parsing fewer programs quickly than the 20 most popular.

We also observe that Seq2Parse successfully parses around 38.50% of the programs with its AllParses approach in 20 seconds and has a median parse time of 23.2 seconds. While this approach is much less efficient that the others, it is also able to generate the exact human repair in 1 out of 3 cases, representing a valuable quality tradeoff (§ 7.2).

Seq2Parse can parse programs with syntax errors for the vast majority of the test set in under 20 seconds with a median parse time of 2.1 seconds.

### 7.4 RQ4: Usefulness

As Seq2Parse is intended as an aid for programmers (especially novices) faced with parse errors, we are also interested in subjective human judgments of the quality and helpfulness of our repairs. Around 34% of repairs produced by Seq2Parse using its AllParses approach are identical to the historical human repair and thus likely helpful for programmers. However, it may be that Seq2Parse's parses (and thus repairs) are still helpful for debugging even when they differ slightly from the human repair (*i.e. non-equivalent* repairs). To investigate this hypothesis, we conduct a human study of the quality and debugging helpfulness of Seq2Parse's non-equivalent repairs.

***Human Study Setup.*** We recruited participants from two large public research institutions (names omitted for blind review) and through Twitter. The study was online, took around 30 minutes, and participants could enter a drawing for one of two $50 awards. In the study, participants were each asked to rate 15 debugging hints randomly selected from a corpus of 50 stimuli.[1]

We created the stimuli by selecting 50 buggy programs from our test set for which Seq2Parse and the human produced different fixes. Other than ensuring a wide array of difficulty (as assessed by how long the human took to fix the error), programs were selected randomly. Each stimulus consisted of a buggy program, its associated syntax error message, and a potential program fix presented as a *debugging hint*. For each stimulus, we produced two versions: one where the debugging hint was generated by Seq2Parse and one where the debugging hint was the historical human fix. Note that, in practice, the historical human fix would *not* be available to a struggling novice in real situations: it represents future or oracular information. Informally, in our comparison, the historical human fixes can be viewed as an upper bound.

Participants rated the quality and helpfulness of each debugging hint using a 1–5 Likert scale. They also indicated if the debugging hint provided helpful information beyond that in the Python error message. Participants were unaware of whether any given hint was generated by a human or Seq2Parse, and participants were never shown multiple fixes to the same program. To be included in the analysis, participants had to assess at least four stimuli. Overall, we analyze 527 unique stimuli ratings from $n = 39$ valid participants (246 for human fixes and 281 for Seq2Parse).

***Overall Results.*** While humans in our study find that non-equivalent repairs produced by Seq2Parse are lower in both quality and debugging helpfulness than those produced manually (2.9/5 helpfulness for tool-produced repairs vs. 3.7/5 for human-produced repairs, $p < 0.001$), humans still often find Seq2Parse's fixes helpful for debugging. Participants found that Seq2Parse repairs contained helpful debugging information beyond that contained in the Python Error message 48% of the time (134/281). This additional debugging information was helpful in terms of both the content (73% of the time) and location (55% of the time). Additionally, Seq2Parse fixes are helpful for easy and hard Syntax Errors alike: we found no statistically-significant difference between the helpfulness or quality of Seq2Parse's repairs for easy (those repaired by the human in under 40 seconds) or hard parse errors (over 40 seconds). Overall, these results indicate that even when Seq2Parse repairs differ from historical human repairs, they can still be helpful for debugging.

***Individual Stimuli.*** Beyond an analysis of Seq2Parse's overall quality, we also analyze the helpfulness of each stimulus. Of the 48 programs for which we collected sufficient data to permit statistical comparison, the historical repair was statistically more helpful for debugging than

[1] All human study stimuli are included in our replication package at *link removed for blind review.*

Seq2Parse's repair for 33% of stimuli (16/48, $p < 0.05$). However, we found that Seq2Parse's repair was actually *more helpful* for debugging than the human's repair for 15% of stimuli (7/48, $p < 0.05$). For the remaining 52% of stimuli, we found no evidence of a statistical difference in the debugging helpfulness of the two repairs.

```
# Buggy
def gcdIter(a, b):
    for i in range(1, a+1):
        if a % i == 0:
        elif b % i == 0:
    return i
gcdIter(9, 12)
```

```
# Human
def gcdIter(a, b):
    for i in range(1, a+1):
        return a % i
gcdIter(9, 12)
```

```
# Seq2Parse
def gcdIter(a, b):
    for i in range(1, a+1):
        if a % i == 0: new_var
        elif b % i == 0: break
    return i
gcdIter(9, 12)
```

(a) Seq2Parse repair significantly more helpful: 4.3/5 vs 1.0/5, $p = 0.03$

```
aList = [12, 'yz', 'ab'];
aList.reverse();
print "List : ", aList
```

```
aList = [12, 'yz', 'ab']
aList.reverse()
```

```
aList = [12, 'yz', 'ab']
aList.reverse()
print("List : ", aList)
```

(b) Seq2Parse repair significantly more helpful: 4.75/5 vs 2.0/5, $p = 0.02$

```
a = int(input(enter a))
print(a***3)
```

```
a = int(input("enter a"))
print(a**3)
```

```
a = int(input(enter)(a))
print(a ** (* 3))
```

(c) Historical human repair significantly more helpful: 1.8/5 vs 4.75/5, $p = 0.01$

Fig. 16. Three example buggy programs followed by their historical human and Seq2Parse repairs. For (a) and (b), Seq2Parse's repair was rated more helpful by participants. For (c), the human repair was more helpful.

To better contextualize these results, we provide examples of stimuli with statistically significant differences in debugging helpfulness. In figure 16b, Seq2Parse's repair was significantly more helpful than the historical repair: Seq2Parse correctly adds parentheses to `print` while the human simply deletes the buggy line, perhaps out of confusion or frustration. Similarly, figure 16a's Seq2Parse repair was also better than the human repair. In this case, the user appears to try to implement a function to calculate the greatest common divisor of two integers, but has empty `if` and `elif` statements. To "fix" this bug, the user deletes the `if` and `elif` and modifies the return statement. However, this fix does not correctly calculate the greatest common divisor. Seq2Parse, on the other hand, adds a template variable to the `if` and `break` to the `elif`. While this also does not implement greatest common divisor, it is viewed as more helpful than the user repair. This example also demonstrates the beneficial ability of our approach to conduct multi-edit repairs.

Figure 16c, on the other hand, shows an example of a more helpful human repair. In this case, the human correctly deletes the extraneous `*` in the power operator while Seq2Parse adds parentheses to make a more complex expression, the result of favoring one insertion over one deletion.

34% of Seq2Parse's repairs are equivalent to historical repairs. Of the remainder, our human study found 15% to be more useful than historical repairs and 52% to be equally useful. In total, including both equivalent and non-equivalent cases, Seq2Parse repairs are at least as useful as historical human-written repairs 78% of the time.

## 8 RELATED WORK

There is a vast literature on automatically repairing or patching programs: we focus on the most closely related work on providing feedback for parse errors.

***Error-Correcting Parsers.*** As we have already demonstrated, error-correcting parses have been proposed for repairing syntax errors and we have extensively described ECE-Parsers [Aho and Peterson 1972]. The technique presented by Burke and Fisher [1987] describes another EC-Parser, which is applicable with LR and LL parsing. It uses three phases: first attempts to repair the parse error by symbol insertions, deletions, or substitutions. If that fails, it tries to close one or more open code blocks and if that fails, it removes code surrounding the erroneous symbol. Finally, it uses *deferred parsing* that may be viewed as double parsing, where one main parser moves forward as much as possible, whereas a second parser is $k$ steps behind, so that it can backtrack to a state $k$ steps before efficiently if a phase fails. van der Spek et al. [2005] have shown that the previous approach is not applicable in real-world languages for some specific cases (*e.g.* multiple function definitions) and has suggested an improvement that works with the JavaCC parser generator and a form of *follow-set error recovery*. Corchuelo et al. [2002] have suggested an error-correcting version of the popular LR parser. Rather than focusing on error production rules, this method adds *error-repair transitions* along with the regular shift/reduce operations. It employs a simple cost model and heuristics to limit the explosion of the repair search space. Finally, Thompson [1976] has suggested using *probabilistic parsing* to overcome the drawback of selecting the minimal-edit repair by using a PCFG to select the most *probable* repair parse. However, these approaches are impractical and inefficient for real-world applications, as they can only successfully parse small examples or use tiny grammars. In contrast, Seq2Parse relies on pre-trained sequence models to efficiently explore the repair search space for a minimal overhead in real-time parsing.

***Sequence Models in Software Engineering.*** Rahmani et al. [2021] and Verbruggen et al. [2021] have suggested using pre-trained auto-regressive transformer models, such as GPT-3 [Brown et al. 2020], to augment pre-existing program synthesis techniques. They use pretrained models to acquire semantic power over smaller subproblems that can't be solved with the syntactic power of classic program synthesis. Similar to Seq2Parse, their work uses established pre-existing algorithms from the NLP and PL research areas. However, Seq2Parse trains its own Transformer-based model to augment an error correcting parsing algorithm, providing more focused prior knowledge than an pretrained sequence model, thus making our model highly accurate.

***Sequence Models for Parsing.*** SynFix [Bhatia and Singh 2016] and *sk_p* [Pu et al. 2016] are two systems that use seq2seq models consisting of Long Short-Term Memory networks (LSTMs). They mostly focus on educational programming tasks in order to learn task-specific patterns for fixing erroneous task solutions. SynFix uses a model per task and uses as an input sequence the program prefix until the error locations that the language parser provides. *sk_p* (while it does not solely focus on syntax errors) makes sequence predictions per program line, by considering only the abstracted context lines (previous and next lines). The model is applied to every program line and the predictions with the highest probabilities are selected. Seq2Parse manages to parse and repair a large number of programs regardless the task they are trying to solve by encoding the full

erroneous programs with a state-of-the-art Transformer model and utilizing an EC-Parser to parse them accordingly, thus achieving a much higher accuracy. Additionally, it uses a real-world dataset of millions of Python programs to learn to effectively parse programs, while SynFix and *sk_p* are trained on smaller datasets of correct programs that have errors manually introduced on training, possibly skewing the predictions away from real-world fixes.

DeepFix [Gupta et al. 2017] is another seq2seq approach for repairing syntactical errors in C programs. It relies on stacked *gated recurrent units* (GRUs) with attention and applies some simple abstraction over the terminal tokens. The input programs are broken into subsequences for each line and the model gets as input all the line subsequences with their associated line numbers. DeepFix only predicts single line fixes and its predictions are applied iteratively multiple times, if multiple parse errors exist or until the parse error is fixed. DeepFix struggles with the same problems as previous work, as it solely relies on the sequence models' capability to learn the full grammar and repair programs with minimal abstraction and prior knowledge over the language.

*Lenient parsing* [Ahmed et al. 2021] presents another sequence model approach. It uses *two seq2seq Transformer models* and trains them with a large corpus of code. One model is trained to repair and create proper nested blocks of code, called BlockFix, and the second one, called FragFix, repairs and parses fragments of code (*e.g.* program statements) within a repaired block. BlockFix tokenizes input program block in a similar manner to our abstracted token sequences, by abstracting identifiers, constants, expressions, etc., and is trained on pairs of valid and manually-corrupted blocks. On the other hand, FragFix repairs on a program-statement level within blocks (mostly focusing on missing semicolons and commas), by using serialized versions of ASTs and error hints manually injected on the ASTs. While this overall approach is mostly automatic, it relies on the manual corruption of a dataset to generate erroneous programs that may not correlate to the errors actual developers make and solely relies on the seq2seq models to learn the underlying language model and make repairs. In contrast, Seq2Parse mitigates this problem by learning how programmers fixed programs from a large corpus and by abstracting via partial parses. Additionally, our use of EC-Parsers and the language grammar significantly improves program repairs.

***Graph models for parsing.*** Graph-based Grammar Fix (GGF) [Wu et al. 2020] suggested using a *Gated Graph Neural Network* encoder for the partial parse trees that can be acquired from a LALR parser and a *GRU* encoder for the parts of the program sequence that are not parsed. This approach aims to better summarize the context of the program in order to train more accurate models. Its models then predict an error location in the program sequence and a token suggestion for the repair. This single-token repair approach is applied iteratively multiple times until the program correctly parses. While this approach is much more accurate than any previous work, as it repairs *58%* of the syntax errors of a real-world dataset, it still lacks the advantages of using a parser with the actual grammar as the final step of the repairing process that Seq2Parse takes benefit from and relies again on the model to learn the semantics of the language.

***Neural Machine Translation (NMT) for Program Repair.*** CoCoNuT [Lutellier et al. 2020] proposed a complex architecture that uses a new *context-aware NMT model* that has two separate *Convolutional Neural Network (CNN)* encoders, one for the buggy lines and one for their surrounding lines. It also uses *ensemble learning* to train NMT models of different hyper-parameters to capture different relations between erroneous and correct code. This approach uses a minimal level of abstraction over the input programs, with only a subword-level tokenization to minimize the vocabulary size and make training tractable. CURE [Jiang et al. 2021] suggested a similar *code-aware NMT model* that is pre-trained using unsupervised learning on correct programs. It also uses a programming language GPT [Brown et al. 2020] model that learns to predict the next token in program sequences and uses beam search to maintain a small set of accurate repairs.

## 9 CONCLUSION

We have presented *neurosymbolic parse program repair*, a new neurosymbolic approach to automatically repair parse errors. Our approach is to use a dataset of ill-parsed programs and their fixed versions to train a Transformer classifier (neural component) which allows us to accurately predict EC-rules for new programs with syntax errors. In order to make accurate predictions, we abstract the low-level program token sequences using partial parses and probabilistic grammars. A small set of predicted EC-rules is finally used with an ECE-Parser (symbolic component) to parse and repair new ill-parsed programs in a tractable and precise manner.

We have implemented our approach in Seq2Parse, and demonstrated, using a corpus of 1,100,000 ill-parsed Python programs drawn from two years of data from an online web-based educational compiler, that Seq2Parse makes accurate EC-rule predictions 81% of the time when considering the top 20 EC-rules, and that the predicted EC-rules let us parse and repair over 94% of the test set in 2.1 sec median parse time, while generating the user fix in almost 1 out of 3 cases. Finally, we conducted a user study with 39 participants which showed that Seq2Parse's edit locations and repairs are useful and helpful, even when they are not equivalent to the user's fix.

## REFERENCES


Alireza Ahadi, Raymond Lister, Shahil Lal, and Arto Hellas. 2018. Learning Programming, Syntax Errors and Institution-Specific Factors. In *Proceedings of the 20th Australasian Computing Education Conference* (Brisbane, Queensland, Australia) *(ACE '18)*. Association for Computing Machinery, New York, NY, USA, 90–96. https://doi.org/10.1145/3160489.3160490

Toufique Ahmed, Premkumar Devanbu, and Vincent J Hellendoorn. 2021. Learning lenient parsing & typing via indirect supervision. *Empirical Software Engineering* 26, 2 (mar 2021). https://doi.org/10.1007/s10664-021-09942-y

A. V. Aho and S. C. Johnson. 1974. LR Parsing. *ACM Comput. Surv.* 6, 2 (jun 1974), 99–124. https://doi.org/10.1145/356628.356629

Alfred V. Aho and Thomas G. Peterson. 1972. A Minimum Distance Error-Correcting Parser for Context-Free Languages. *SIAM J. Comput.* 1 (1972), 305–312.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR* abs/1409.0473 (2015).

Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. https://doi.org/10.48550/ARXIV.1603.06129

Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 209–210.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. https://doi.org/10.48550/ARXIV.2005.14165

Michael G. Burke and Gerald A. Fisher. 1987. A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery. *ACM Trans. Program. Lang. Syst.* 9, 2 (mar 1987), 164–197. https://doi.org/10.1145/22719.22720

Nigel P Chapman. 1987. *LR Parsing: Theory and Practice*. Cambridge University Press, New York, NY, USA.

Michael Collins. 2013. Probabilistic Context-Free Grammars (PCFGs). *Lecture Notes* (2013). https://u.cs.biu.ac.il/~89-680/collins-pcfgs.pdf

Rafael Corchuelo, José A. Pérez, Antonio Ruiz, and Miguel Toro. 2002. Repairing Syntax Errors in LR Parsers. *ACM Trans. Program. Lang. Syst.* 24, 6 (nov 2002), 698–710. https://doi.org/10.1145/586088.586092

Benjamin Cosman, Madeline Endres, Georgios Sakkas, Leon Medvinsky, Yao-Yuan Yang, Ranjit Jhala, Kamalika Chaudhuri, and Westley Weimer. 2020. *PABLO: Helping Novices Debug Python Code Through Data-Driven Fault Localization*. Association for Computing Machinery, New York, NY, USA, 1047–1053. https://doi.org/10.1145/3328778.3366860

Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All Syntax Errors Are Not Equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (Haifa, Israel) *(ITiCSE '12)*. Association for Computing Machinery, New York, NY, USA, 75–80. https://doi.org/10.1145/2325296.2325318

Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13, 2 (Feb. 1970), 94–102. https://doi.org/10.1145/362007.362035

Madeline Endres, Georgios Sakkas, Benjamin Cosman, Ranjit Jhala, and Westley Weimer. 2019. InFix: Automatically Repairing Novice Program Inputs. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 399–410. https://doi.org/10.1109/ASE.2019.00045

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press, 180–184. http://www.deeplearningbook.org.

Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. *Programming Language Design and Implementation* (2018). https://doi.org/10.1145/3192366.3192387

Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 31, 1 (Feb. 2017). https://ojs.aaai.org/index.php/AAAI/article/view/10742

Momchil Hardalov, Ivan Koychev, and Preslav Nakov. 2018. Towards Automated Customer Support. In *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. Springer, 48–59. https://doi.org/10.1007/978-3-319-99344-7_5

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer New York. https://doi.org/10.1007/978-0-387-84858-7

Frederick Jelinek, John D Lafferty, and Robert L Mercer. 1992. Basic Methods of Probabilistic Context Free Grammars. In *Speech Recognition and Understanding*. Springer, 345–360. https://doi.org/10.1007/978-3-642-76626-8_35

Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE.

Yoon Kim, Carl Denton, Luong Hoang, and Alexander M. Rush. 2017. Structured Attention Networks. *CoRR* abs/1702.00887 (2017). arXiv:1702.00887 http://arxiv.org/abs/1702.00887

Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. (22 Dec. 2014). arXiv:1412.6980 [cs.LG]

Donald E. Knuth. 1965. On the translation of languages from left to right. *Information and Control* 8, 6 (1965), 607–639. https://doi.org/10.1016/S0019-9958(65)90426-2

Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *International Symposium on Software Testing and Analysis*. ACM, 165–176. https://doi.org/10.1145/2931037.2931051

Sarah K. Kummerfeld and Judy Kay. 2003. The Neglected Battle Fields of Syntax Errors. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20* (Adelaide, Australia) *(ACE '03)*. Australian Computer Society, Inc., AUS, 105–111.

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

Matias Martinez, Laurence Duchien, and Martin Monperrus. 2013. Automatically extracting instances of code change patterns with AST analysis. In *2013 IEEE international conference on software maintenance*. IEEE, 388–391.

Philippe McLean and R. Nigel Horspool. 1996. A Faster Earley Parser. In *CC*.

Eugene W Myers. 1986. An O(ND) difference algorithm and its variations. *Algorithmica* 1, 1-4 (1986), 251–266.

Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *International Conference on Machine Learning*. 807–814.

Michael A Nielsen. 2015. *Neural Networks and Deep Learning*. Determination Press.

Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *International Symposium on Software Testing and Analysis*. ACM, 199–209. https://doi.org/10.1145/2001420.2001445

Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. sk_p: a neural program corrector for MOOCs. https://doi.org/10.48550/ARXIV.1607.02902

Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (oct 2017), 24 pages. https://doi.org/10.1145/3077618

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, Article 158 (oct 2021), 29 pages. https://doi.org/10.1145/3485535

Sanguthevar Rajasekaran and Marius Nicolae. 2014. An error correcting parser for context free grammars that takes less than cubic time. https://doi.org/10.48550/ARXIV.1406.3405

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning Representations by Back-propagating Errors. *Nature* 323, 6088 (1986), 533–536. https://doi.org/10.1038/323533a0

Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 2020. Type Error Feedback via Analytic Program Repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 16–30. https://doi.org/10.1145/3385412.3386005

Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (Jan 2015), 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. arXiv:1409.3215 [cs.CL]

Richard A. Thompson. 1976. Language Correction Using Probabilistic Grammars. *IEEE Trans. Comput.* C-25, 3 (1976), 275–286. https://doi.org/10.1109/TC.1976.5009254

P. van der Spek, N. Plat, and C. Pronk. 2005. Syntax Error Repair for a Java-Based Parser Generator. *SIGPLAN Not.* 40, 4 (April 2005), 47–50. https://doi.org/10.1145/1064165.1064173

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc., 5998–6008. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic Programming by Example with Pre-Trained Models. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 100 (oct 2021), 25 pages. https://doi.org/10.1145/3485477

Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, Align, and Repair: Data-driven Feedback Generation for Introductory Programming Exercises. In *Programming Language Design and Implementation*. 481–495. https://doi.org/10.1145/3192366.3192384

P.J. Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proc. IEEE* 78, 10 (1990), 1550–1560. https://doi.org/10.1109/5.58337

Liwei Wu, Fei Li, Youhua Wu, and Tao Zheng. 2020. *GGF: A Graph-Based Method for Programming Language Syntax Error Correction*. Association for Computing Machinery, New York, NY, USA, 139–148. https://doi.org/10.1145/3387904.3389252