Realtime syntax repair with resource constraints

Breandan Considine[1], Jin Guo[1], and Xujie Si[2]

[1] McGill University, Montréal, QC H2R 2Z4, Canada
{breandan.considine@mail, jguo@cs}.mcgill.ca
[2] University of Toronto, Toronto, ON, M5S 1A1 Canada
six@utoronto.ca

Abstract. We describe the implementation of a tool for real-time syntax correction in an IDE. Upon activation, our tool takes a syntactically invalid source code fragment around the caret position, and produces a small set of suggested repairs. We model the problem of syntax repair as a structured prediction task, whose goal is to generate the most likely valid repair in a small edit distance of the invalid code fragment.

Keywords: Error correction · CFL reachability · Langauge games.

1 Introduction

Syntax errors are a familiar nuisance for software developers. Whenever a syntax error is detected, the IDE typically flags the offending code fragment, but offers little guidance on how it should be fixed. The developer must inspect the code and manually apply the appropriate fix through a process of trial and error. This process can be distracting and time-consuming, especially for novice developers. In this paper, we describe a tool for automatic syntax repair in an IDE.

We propose a new approach to syntax repair and accompanying tool, called *Tidyparse* that suggests a small set of repairs to the user, which are guaranteed to be valid, minimal and natural. Our repair tool is a fusion of two widely available components: grammars and language models. At first glance, these two models are not obviously synergistic: the grammar is a deterministic, formal model of the language, while the language model is only an approximate generator of linguistic patterns. However, we show that by carefully integrating them, it is possible to generate repairs that are always correct and highly natural.

Language models are statistical models that generate natural sequences of text, however, these models make no guarantees about the validity of the generated text. Given a sequence of previous tokens, $\sigma_0, \ldots, \sigma_{n-1}$, an autoregressive language model outputs a distribution over the next most likely token, σ_n.

Almost every programming language ever developed is syntactically context-free, which means the syntax of the language can be expressed as a context-free grammar (CFG). This grammar can be used to recognize the validity of a given input sequence, or force an autoregressive language model to generate only syntactically valid sequences by blocking out invalid tokens during inference.

Likewise, this grammar can be also used to construct a synthetic grammar, recognizing all and only valid sequences within a certain edit distance of a broken source code fragment using the Bar-Hillel construction. Our approach uses a pretrained language model to sample repair candidates from this synthetic grammar. We rank the results by negative log likelihood under the language model, and present the top k candidates to the user. The user can then select the most appropriate repair from the list, or continue to edit the code manually.

Let us consider an example. Suppose the user has written the following code fragment: `v = df.iloc(5:, 2:)`. Assuming an alphabet of just a hundred lexical tokens, this tiny statement has millions of possible two-token edits, yet only six of those possibilities are accepted by the Python parser:

(1) `v = df.iloc(5:, 2,)` (3) `v = df.iloc(5[:, 2:])` (5) `v = df.iloc[5:, 2:]`

(2) `v = df.iloc(5), 2()` (4) `v = df.iloc(5:, 2:)` (6) `v = df.iloc(5[:, 2])`

To generate these repairs, we first lexicalize the input as follows:

```
v = df.iloc(5:, 2:)
v    = df   . iloc ( 5      : , 2      : )
NAME = NAME . NAME ( NUMBER : , NUMBER : )
```

Next, we will construct an automaton that recognizes every string within a certain edit distance of the input. We will depict the process for a simpler language, where the grammar is $S \rightarrow$ `( )` $\mid$ `(` S `)` $\mid SS$ and the broken code is `( ) )`.

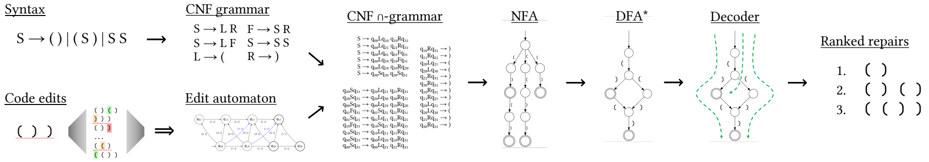


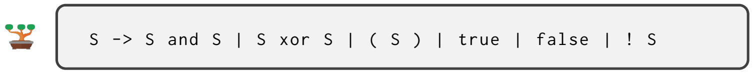
Fig. 1. Simplified dataflow. Given a grammar and broken code fragment, we create an automaton generating the language of small edits, then intersect it with the grammar to produce an intersection grammar, which can be simplified to a DFA and decoded.

We first construct an automaton that recognizes every string within a certain edit distance of the input. We then reduce the grammar to Chomsky normal form (CNF) and construct an intersection grammar, which recognizes all and only valid sequences recognized by the grammar and edit automaton. This grammar is known to be non-recursive, and can be simplified to a deterministic finite automaton (DFA) using standard techniques. Finally, we decode the DFA to produce a list of repair candidates, which we rank by negative log likelihood under the language model.

Now that we have a high-level overview of our approach, we will demonstrate a few of the capabilities of our tool by means of some usage examples.

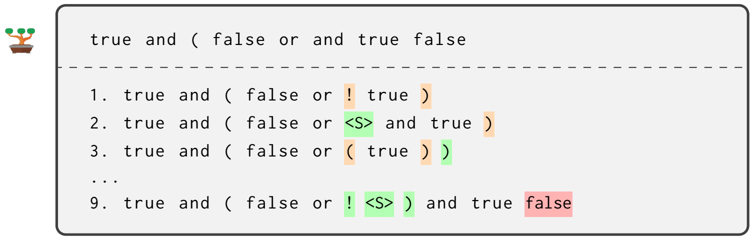
2 Usage examples

Tidyparse offers a convenient user interface featuring a text editor, a grammar editor and a parse tree viewer for interactive prototyping. All tokens are delimited by whitespace. For example, suppose we have the following grammar:



```
S -> S and S | S xor S | ( S ) | true | false | ! S
```

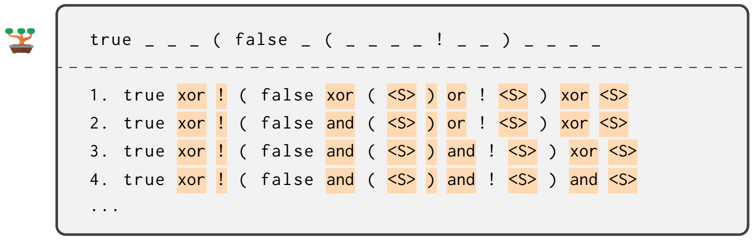
Syntax repair is the primary intended use case of our tool. If given an unparsable string, it will return an ordered set of suggestions how to fix it, highlighted with colors, where green is insertion, orange is substitution and red is deletion.



```
true and ( false or and true false

1. true and ( false or ! true )
2. true and ( false or <S> and true )
3. true and ( false or ( true ) )
...
9. true and ( false or ! <S> ) and true false
```

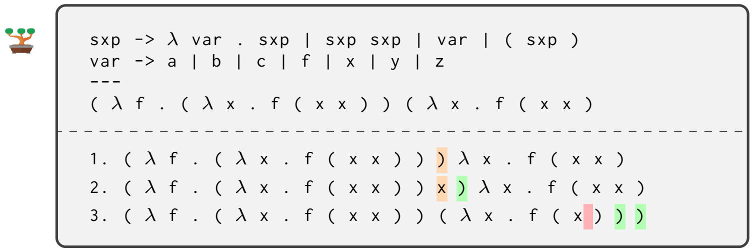
Code completion is the secondary intended use case. Given a string containing holes, our tool will return several possible completions:



```
true _ _ _ ( false _ ( _ _ _ _ ! _ _ ) _ _ _ _

1. true xor ! ( false xor ( <S> ) or ! <S> ) xor <S>
2. true xor ! ( false and ( <S> ) or ! <S> ) xor <S>
3. true xor ! ( false and ( <S> ) and ! <S> ) xor <S>
4. true xor ! ( false and ( <S> ) and ! <S> ) and <S>
...
```

For simplicity, it is also possible to define a grammar and string side-by-side, as shown in the untyped λ-calculus example below:



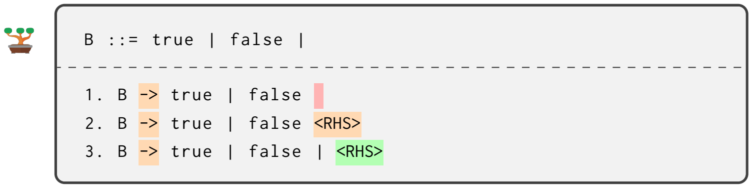
```
sxp -> λ var . sxp | sxp sxp | var | ( sxp )
var -> a | b | c | f | x | y | z
---
( λ f . ( λ x . f ( x x ) ) ( λ x . f ( x x )

1. ( λ f . ( λ x . f ( x x ) ) ) λ x . f ( x x )
2. ( λ f . ( λ x . f ( x x ) ) x ) λ x . f ( x x )
3. ( λ f . ( λ x . f ( x x ) ) ( λ x . f ( x ) ) )
```

By default, Tidyparse samples the finite intersection language uniformly without replacement, then sorts the results by Levenshtein distance. Customizing the ranking order is possible using a programmatic interface.

2.1 Grammar assistance

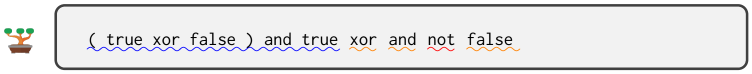
Tidyparse uses a CFG to parse the CFG, so it can provide editing assistance while the user is designing the CFG. For example, if the CFG is missing a term or uses the wrong delimiters, it will suggest a list of possible fixes.



```
B ::= true | false |
---------------------------------------------------
1. B -> true | false
2. B -> true | false <RHS>
3. B -> true | false | <RHS>
```

2.2 Syntax highlighting

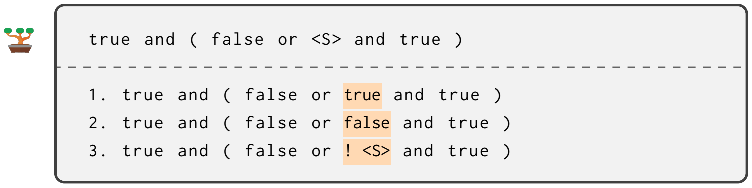
Subsequences which are partly parseable are underlined in blue. Unparsable alphabetic tokens are marked orange. All other tokens are marked red.



```
( true xor false ) and true xor and not false
```

2.3 Interactive nonterminal expansion

Users can interactively build up a complex expression by placing the caret over a nonterminal they wish to expand, then pressing ctrl+Space to receive a list of possible substitutions.



```
true and ( false or <S> and true )
---------------------------------------------------
1. true and ( false or true and true )
2. true and ( false or false and true )
3. true and ( false or ! <S> and true )
```

2.4 Nonterminal stubs

Tidyparse augments CFGs with two additional rules, which are desugared into a vanilla CFG before parsing. The first rule, α-SUB, allows the user to define a nonterminal parameterized by α, a non-recursive nonterminal in the same the CFG representing some finite type and its inhabitants. α-SUB replaces all productions containing $\langle\alpha\rangle$ with the terminals in their transitive closure, $\alpha \rightarrow^* \beta$. The second rule, α-INT, introduces bracketed terminals for each user-defined nonterminal, representing unexpanded subexpressions.

Tidyparse can also perform a limited form of type checking. Typed expressions are automatically expanded into ordinary nonterminals using the α-SUB

$$\frac{\mathcal{G} \vdash (w\langle\alpha\rangle \to xz) \in P \qquad \alpha^* : \{\beta \mid (\alpha \to^* \beta) \in P\}}{\mathcal{G} \vdash \forall \beta \in \alpha^*.(w\langle\alpha\rangle \to xz)[\beta/\alpha] \in P'}\ \alpha\text{-SUB}$$

$$\frac{\mathcal{G} \vdash v \in V}{\mathcal{G} \vdash (v \to \langle v \rangle) \in P}\ \langle\cdot\rangle\text{-INT}$$

rule, for example when parsing an expression of the form $x + y$, the grammar will recognize `true + false` and `1 + 2`, but not `1 + true`.

```
E<X> -> E<X> + E<X> | E<X> * E<X> | ( E<X> )
X -> Int | Bool

# The above grammar is equivalent to writing:

E<Int> -> E<Int> + E<Int> | E<Int> * E<Int>
E<Bool> -> E<Bool> + E<Bool> | E<Bool> * E<Bool>
```

3 Related work

Many methods to sample from language models have been proposed. Some of these guarantee that all samples are grammatically valid. Others guarantee that all grammatically valid samples are generable. The trick is not just synthesizing valid functions, but doing so in a parallel communication-free manner, without compromising soundness or completeness. The goal is to massively scale up a discrete sampler without replacement.

This problem is also closely related to model counting in the CSP literature, so a practical speedup could lead to improvements on a lot of interesting downstream benchmarks.

In general, the problem of program induction from input-output examples is not well-posed, so specialized solvers that can make stronger assumptions will usually have an advantage on domain-specific benchmarks. Most existing program synthesizers do not satisfy all of these desiderata, e.g., soundness (**S**), completeness (**S**), naturalness (**N**), and parallelism (||). It depends on how you define ||, but basically, we want to decode in parallel. So an LLM that uses a GPU we do not consider to be "parallel" in the sense we mean here.

	S	**C**	**N**	**Theory**	**||**	**Tool**
Tidyparse [4]	✓	✓	✓	$\text{CFG}_\cap$	✓	IDE-ready
Seq2Parse [8]	✓†	✗	✓	CFG	✗	Python
BIFI [11]	✗	✗	✓	Σ^*	✗	Python
OrdinalFix [12]	✓	✗	✗	CFG+	✗	Rust
Aho/Peterson [1]	✓	✗	✗	CFG	✗	None

Now, we consider just LLM-based SyGuS synthesizers.

	S	C	N	Theory	||	Tool
Outlines [10]	✓†	✓†	✓	CFG	✗	Python
SynCode [9]	✓	?	✓	CFG	✗	Python
GAD [6]	✓	?	✓	CFG	✗	Python
CodeGuard+ [5]	✓	?	✓	CFG	✗	Python
FLAP [7]	✓	?	✓	CFG	✗	Python
DOMINO [3]	✓	✗	✓	CFG	✗	Python

Also, we consider discrete program search and enumerative search techniques that do not use an LLM, but allow some semantic constraints on the generated program.

	S	C	N	Theory	||	Tool
Boltzmann Brain [2]	✓	✓	✓	UTλC	✓	Python
OrdinalFix [12]	✓	?	✗	N/A	✓	Rust
Bend/DPS	✓	?	?	IntCalc	✓	CUDA

4 Conclusion

References

1. Aho, A.V., Peterson, T.G.: A minimum distance error-correcting parser for context-free languages. SIAM Journal on Computing **1**(4), 305–312 (1972)
2. Bendkowski, M.: Automatic compile-time synthesis of entropy-optimal boltzmann samplers. arXiv preprint arXiv:2206.06668 (2022)
3. Beurer-Kellner, L., Fischer, M., Vechev, M.: Guiding LLMs the right way: Fast, non-invasive constrained generation. arXiv preprint arXiv:2403.06988 (2024)
4. Considine, B.: A pragmatic approach to syntax repair. In: Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. pp. 19–21 (2023)
5. Fu, Y., Baker, E., Chen, Y.: Constrained decoding for secure code generation. arXiv preprint arXiv:2405.00218 (2024)
6. Park, K., Wang, J., Berg-Kirkpatrick, T., Polikarpova, N., D'Antoni, L.: Grammar-aligned decoding. arXiv preprint arXiv:2405.21047 (2024)
7. Roy, S., Sengupta, S., Bonadiman, D., Mansour, S., Gupta, A.: Flap: Flow adhering planning with constrained decoding in LLMs. arXiv preprint arXiv:2403.05766 (2024)
8. Sakkas, G., Endres, M., Guo, P.J., Weimer, W., Jhala, R.: Seq2parse: neurosymbolic parse error repair. Proceedings of the ACM on Programming Languages **6**(OOPSLA2), 1180–1206 (2022)
9. Ugare, S., Suresh, T., Kang, H., Misailovic, S., Singh, G.: Improving LLM code generation with grammar augmentation. arXiv preprint arXiv:2403.01632 (2024)

10. Willard, B.T., Louf, R.: Efficient guided generation for LLMs. arXiv preprint arXiv:2307.09702 (2023)
11. Yasunaga, M., Liang, P.: Break-it-fix-it: Unsupervised learning for program repair. In: International Conference on Machine Learning. pp. 11941–11952. PMLR (2021)
12. Zhang, W., Wang, G., Chen, J., Xiong, Y., Liu, Y., Zhang, L.: Ordinalfix: Fixing compilation errors via shortest-path cfl reachability. arXiv preprint arXiv:2309.06771 (2023)