Neurosymbolic Repair for Low-Code Formula Languages

ROHAN BAVISHI*†, UC Berkeley, USA
HARSHIT JOSHI†, Microsoft, India
JOSÉ CAMBRONERO‡, Microsoft, USA
ANNA FARIHA‡, Microsoft, USA
SUMIT GULWANI‡, Microsoft, USA
VU LE‡, Microsoft, USA
IVAN RADIČEK‡, Microsoft, Croatia
ASHISH TIWARI‡, Microsoft, USA

Most users of low-code platforms, such as Excel and PowerApps, write programs in domain-specific formula languages to carry out nontrivial tasks. Often users can write most of the program they want, but introduce small mistakes that yield broken formulas. These mistakes, which can be both syntactic and semantic, are hard for low-code users to identify and fix, even though they can be resolved with just a few edits. We formalize the problem of producing such edits as the *last-mile repair* problem. To address this problem, we developed LaMirage, a LAst-MIle RepAir-engine GEnerator that combines symbolic and neural techniques to perform last-mile repair in low-code formula languages. LaMirage takes a grammar and a set of domain-specific constraints/rules, which jointly approximate the target language, and uses these to generate a repair engine that can fix formulas in that language. To tackle the challenges of localizing errors and ranking candidate repairs, LaMirage leverages neural techniques, whereas it relies on symbolic methods to generate candidate edits. This combination allows LaMirage to find repairs that satisfy the provided grammar and constraints, and then pick the most natural repair. We compare LaMirage to state-of-the-art neural and symbolic approaches on 400 real Excel and Power Fx formulas, where LaMirage outperforms all baselines. We release these benchmarks to encourage subsequent work in low-code domains.

164

CCS Concepts: • **Software and its engineering → Error handling and recovery**.

Additional Key Words and Phrases: Program Repair, Neurosymbolic, Low-Code

ACM Reference Format:
Rohan Bavishi, Harshit Joshi, José Cambronero, Anna Fariha, Sumit Gulwani, Vu Le, Ivan Radiček, and Ashish Tiwari. 2022. Neurosymbolic Repair for Low-Code Formula Languages. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 164 (October 2022), 30 pages. https://doi.org/10.1145/3563327

*Work done during an internship with the PROSE team at Microsoft
†Equal contribution
‡Authors in alphabetic order

Authors' addresses: Rohan Bavishi, UC Berkeley, USA, rbavishi@cs.berkeley.edu; Harshit Joshi, Microsoft, India, t-hjoshi@microsoft.com; José Cambronero, Microsoft, USA, jcambronero@microsoft.com; Anna Fariha, Microsoft, USA, annafariha@microsoft.com; Sumit Gulwani, Microsoft, USA, sumitg@microsoft.com; Vu Le, Microsoft, USA, levu@microsoft.com; Ivan Radiček, Microsoft, Croatia, ivradice@microsoft.com; Ashish Tiwari, Microsoft, USA, astiwar@microsoft.com.

2475-1421/2022/10-ART164
https://doi.org/10.1145/3563327

1 INTRODUCTION

Low-code (LC) platforms allow end users to build applications or carry out complex calculations with little-to-no programming experience. These platforms promise to democratize the access to computational tools and skills for end-users across a wide range of domains. LC domains include traditional end-user applications, such as spreadsheets (Excel [Microsoft Excel 2021] and Google Sheets [Google Sheets 2019]), but are increasingly expanding to more diverse areas, such as robotic process automation frameworks (Power Automate [Microsoft Power Automate 2019], UIPath [UiPath 2019]), and enterprise apps (Power Apps [Microsoft Power Apps 2019], Appian [Appian 2022]). While these tools usually offer a graphical user interface with basic functionality, most nontrivial applications built in this domain require the end user to write "small programs" or *formulas*. These formula-like languages are specifically designed for LC users [Microsoft Power Fx overview 2022] and are tailored to mirror spreadsheet formula languages, such as that in Excel, with which many users are already familiar. However, low-code languages can support functionalities that go beyond traditional spreadsheet-like languages. For example, formulas written in PowerApp's Power Fx language can interact with the user interface.

Many of these platforms have significant user bases, resulting in a huge amount of code written as formulas. For example, Excel has 1 billion users [Morgan Stanley 2015]. Power Apps, an LC platform for building enterprise apps, is one of Microsoft's fastest-growing product offerings (based on a recent earnings call). Economically, the LC sector is expected to continue to grow substantially. For example, Gartner (a market consulting firm) predicts that up to 65% of application development will be done in such platforms by 2024 [VentureBeat 2022].

Traditional software engineers have benefited from decades of academic research at the intersection of programming languages, software engineering, and artificial intelligence. For example, program synthesis research has enabled engineers to quickly extract information from complex logs [Raza and Gulwani 2017], fix network policies [Hallahan et al. 2017], and wrangle dataframes [Bavishi et al. 2019]. Code search techniques, often integrated into version control platforms such as GitHub, allow developers to quickly search for related code snippets, often overcoming syntax-level differences [Premtoon et al. 2020]. Automated refactoring tools [Miltner et al. 2019] facilitate tasks such as updating APIs [Gao et al. 2021]. Techniques can automatically produce patches by leveraging static analyzers, test suites, or code examples [Goues et al. 2019a].

However, many of these advances have been focused on general purpose programming languages used by traditional programmers, such as Java, C#, C++, or C, but not on LC platforms. This lack of LC-developer assistance can limit the accessibility of LC platforms, despite their intended goal of democratizing computational power. Our goal is to provide this new class of programmers with first-class feedback and tooling comparable to that available for traditional programmers. To lead this effort, we first identify: *where is the first place LC users tend to get stuck?* Based on user forums and discussions with both the Excel and PowerApps teams at Microsoft, we identified that small mistakes (first focused on syntax, and then on semantics) paired with lack of error assistance are often the first stumbling block for LC programmers. Figure 1 shows some user errors taken from help forums [MrExcel Message Board 2021; Power Apps Community 2021] and the corresponding unhelpful compile-error messages. This problem is compounded in the end-user domain as the formula authors often lack the experience to identify the correct repair even if the application were to provide a more detailed error message, which can lead to substantial frustration. Prior work studying programming novices has similarly found that syntax errors can contribute substantially to user frustration [Drosos et al. 2017].

Last-mile repair problem. We studied faulty formulas reported in LC help forums [MrExcel Message Board 2021; Power Apps Community 2021] and shared by the Excel and PowerApps teams

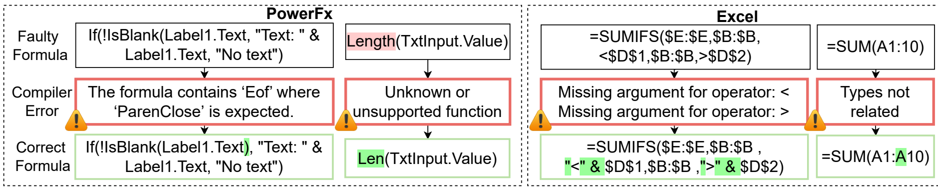

Fig. 1. Real faulty formulas taken from help forums, their compile-error messages, and the correct formulas

at Microsoft. We observed that: (1) the formulas are often almost correct, (2) most of the essential components of the formulas are present in the correct order, (3) the errors are usually in terminal symbols, typically *punctuation*, such as unbalanced parentheses, missing commas, missing quotes, etc., and (4) the repaired formulas are usually within a small *token edit distance*[1] of the intended correct formulas. Repair of such faulty formulas, which are close to the correct formula, involve edits that an experienced programmer can identify without additional information, albeit with additional time and effort. We focus on the problem of repairing such almost-correct formulas and call it the *last-mile repair problem* (Section 3).

Neural techniques are not enough: Recently, the machine-learning (ML) community has taken up the problem of repairing program errors in general-purpose languages by using deep neural networks [Gupta et al. 2017; Yasunaga and Liang 2020, 2021]. Neural techniques, however, suffer from multiple shortcomings in our setting. First, these techniques are data-hungry and require availability of huge program corpora from which to learn. However, public corpora are not readily available in LC domains. Second, even large pre-trained models often struggle to guarantee *correctness* of the generated code and end up generating code that contains mistakes (including syntax errors), as shown by a recent work [Austin et al. 2021; Poesia et al. 2022]. In the context of repair, it is critical that we do not introduce additional mistakes or ignore the existing ones.

Symbolic techniques are not enough: Symbolic techniques, such as error recovery in parsers [Aho and Peterson 1972; Fischer et al. 1979; Rajasekaran and Nicolae 2014], can provide some guarantees (e.g., syntactic validity), but their design is typically constrained by the tradeoffs of their ultimate use-case: compilation toolchains. Compilers solve a well-defined problem, and, thus, should be deterministic and preserve semantics. For example, compilers typically have a limited lookahead and implement limited error-recovery, e.g. panic mode [Aho et al. 1986], if any. In contrast, repair engines need to "guess" the meaning of an ill-formed program, as there may be multiple possible valid repair candidates (i.e., error-free variants) of a faulty program, but not all may match the user's intent. Furthermore, these repairs may constitute additions and deletions of user input, often faraway from the location where the error is detected. Purely symbolic techniques often fail to distinguish between, or even generate, such viable candidate repairs. For example, the state-of-the-art error-recovery tool GRMTOOLS [Diekmann and Tratt 2020] non-deterministically picks from edit operation sequences within the same edit distance. It also focuses on the first error location found, and ignores repairs that involve edits preceding that error location. Our use-case requires generation of exhaustive repairs and a finer-grained ranking among multiple repairs.

Neurosymbolic technique: Increasingly, systems combine neural and symbolic approaches to leverage the advantages of both. For example, Synchromesh [Poesia et al. 2022] enforces symbolic constraints on a large pre-trained language model (LM) during decoding for code generation from natural language descriptions. These symbolic constraints remove common LM mistakes such as referencing unavailable names. Another effective approach is to use neural models that can guide

[1]Similar to string edit distance, token edit distance involves adding, removing, or replacing program tokens.

otherwise symbolic approaches. For example, Kalyan et al [Kalyan et al. 2018] showed that a model can help speed up search substantially by guiding branching decisions in deductive search.

Neurosymbolic repair with LaMirage. Based on the above observations, we design LaMirage, a framework that can generate a last-mile repair engine tailored to a particular target LC formula language using neurosymbolic techniques for proposed repair candidate enumeration and ranking. To motivate our design, we take inspiration from program synthesis [Gulwani et al. 2017]. The feasibility of program synthesis rests on two pillars: (1) controlling the search space through effective candidate enumeration and (2) ranking to pick one from many candidates. To address these requirements, LaMirage has the following key properties:

- *Effective enumeration.* A naïve approach that enumerates and explores all possible options fails to scale because the search space is large. LaMirage exploits the fact that most errors are in certain *unreliable language symbols*. This observation allows us to (1) produce language-agnostic edit actions based on insertion/deletion/update of such unreliable symbols, and (2) bias enumeration to these symbols, allowing the engine to search deeper in the search tree, and, thus, examine repair candidates that involve fixes far from the reported error locations.
- *Complementary domain specific knowledge.* The language-agnostic edit actions are syntax driven, editing the buggy formula to satisfy the grammar provided. However, non-context-free properties may also need to be enforced to produce a valid repair candidate. To enforce such properties, LaMirage uses additional rules to generate repairs that eliminate certain classes of semantic errors, such as adjusting the number of parameters in a call to remove arity errors.
- *Neural error localization and candidate selection.* In cases where the actual error location may lie outside the range considered by the symbolic engine, LaMirage uses a *pointer network* [Vinyals et al. 2015] to predict additional error location ranges. To enumerate candidates at these locations, we consider a (location-specific) *edit range* surrounding each predicted location. This symbolic relaxation allows us to take advantage of the predictions even if they are imperfect. The next challenge we address with a neural technique is the selection of the intended correct formula among the set of generated repair candidates. Ranking only based on distance from the faulty formula is not sufficient, as this often leads to ties. We break such ties using *neural selection*: we use a fine-tuned, pre-trained, deep-neural-network-based language model to guide us to the most natural formula. Our experiments (Section 5) show the effectiveness of our neural localization and selection approaches.

In contrast to existing symbolic state of the art, LaMirage offers more expressive repairs. In particular, while being syntax-guided, LaMirage supports repairs to some semantic errors by allowing language developers to easily integrate domain-specific strategies for identifying errors beyond those captured by a CFG. Additionally, LaMirage supports backtracking (carried out both symbolically and by a neural localizer) to perform edits required for errors that have a root cause at a different location from where they are detected. Both of these contributions allow LaMirage to substantially outperform the symbolic state of the art in our evaluation.

From a neural perspective, LaMirage follows prior work in the area of program synthesis and program repair that use machine learning to improve their search process. The novelty of LaMirage lies in its application of this combination of techniques to the problem of fixing last-mile errors in the low-code domain, paired with the specific setup (and insights) required to make neural models, such as our localizer and ranker, work well in combination with an effective symbolic approach.

In summary, we make the following contributions:

- We define the problem of *last-mile repair* for buggy programs (Section 3). We present a tractable formulation that approximates the target execution engine with a grammar and a set of constraints.

Table 1. The user authors Power Fx formula $P1$, but the compiler reports an error at :. To automatically fix this, LaMirage first inserts a) to get $P2$, inducing the correct arity for the `IsBlank` function call. LaMirage then generates multiple repairs for the remaining errors, including $P3$ and $P4$, both of which are edit distance 3 from $P1$. The correct expression, $P4$, is ranked higher based on its naturalness by fine-tuned CodeBERT. This example is a real user's Power Fx formula adapted for ease of exposition.

	Faulty expression at various stages of repair	Description
P1:	`If (!IsBlank(LunchSeminar, UpdateContext(LunchSeminarVar : LunchSeminar)))`	Error location :
P2:	`If (!IsBlank(LunchSeminar), UpdateContext(LunchSeminarVar : LunchSeminar)))`	ADD)
P3:	`If (!IsBlank(LunchSeminar), UpdateContext(LunchSeminarVar.LunchSeminar()))`	REPLACE : by . and ADD (
P4:	`If (!IsBlank(LunchSeminar), UpdateContext({LunchSeminarVar : LunchSeminar}))`	ADD { and REPLACE) by }

We motivate the application of last-mile repair in the low-code domain (Section 2), where the grammar and constraints can successfully model the target engine.

- We present LaMirage, a neurosymbolic approach that combines the strengths of both symbolic techniques (effective enumeration) and deep learning (natural ranking and long-range error localization) to solve the last-mile repair problem in LC formula languages. We show that more expressive repairs, paired with effective use of neural models to complement the symbolic procedures, can improve the number of repairs in our low-code evaluation.
- We developed concrete grammar and constraint approximations that are empirically effective for the domains of Excel and Power Fx. We evaluate LaMirage on two benchmark sets of 200 faulty formulas from each domain collected from help forums and system telemetry at Microsoft. We compare LaMirage to the state-of-the-art neural and symbolic systems [Chen et al. 2021; Diekmann and Tratt 2020; Open AI 2022; Yasunaga and Liang 2021], and a commercially available alternative. In Excel, LaMirage's top repair candidate matched the ground truth repair in 174 out of 200 formulas, compared to 147 for the next best system, Codex-Edit. In Power Fx, LaMirage's top repair candidate matched the ground truth in 170 out of 200 formulas, compared to 106 for Codex-Edit. We also motivate our design with ablation studies.
- We release our benchmarks, gathered from real user forum posts and telemetry, along with manually annotated ground-truth repairs.[2]

The rest of this paper is structured as follows. Section 2 provides an example of last-mile repair in a PowerFx formula. Section 3 presents the formal problem definition, Section 4 details our approach, Section 5 presents our experimental results, Section 6 discusses the choices language developers can make in LaMirage, Section 7 summarizes related work and Section 8 provides closing remarks.

2 MOTIVATING EXAMPLES AND OVERVIEW

Popular low-code (LC) platforms, such as Excel and Power Apps, often expose a subset of commonly used functionalities via drop-down menus in a graphical user interface. However, to accomplish more involved tasks, users must write formulas in the underlying LC formula language. This can pose significant challenges, as LC platform users have varying degrees of programming expertise and are typically attracted to such platforms mainly due to a relatively lower entry barrier. In particular, LC formula authors often struggle when identifying and manually fixing small errors in their formulas. In this *last-mile* setting, the formula is almost correct, but still requires a few tweaks to be fully correct. In Example 2.1, we illustrate this common experience by walking through the details of a Power Apps example, where users write Power Fx formulas.

Example 2.1. Consider an incorrect Power Fx formula $P1$ (Table 1). This example is a real user's Power Fx formula adapted for ease of exposition. The user is trying to update the context by

[2] Benchmarks will be available at https://github.com/microsoft/prose-benchmarks/tree/main/LaMirage after privacy review

assigning the value of `LunchSeminar` to `LunchSeminarVar` if `LunchSeminar` is not empty. The formula $P1$ has errors: the compiler points to the colon ':' as the culprit location. However, the root cause of the error is not at that location. □

Novices often struggle to fix such buggy formulas. This can result from multiple factors, including: lack of user experience writing formulas, large formulas that compose multiple functions, lack of LC editor support for simple features such as syntax highlighting, ambiguous or complex error messages from the underlying LC language toolchain. This combination of factors can make spotting and fixing even small bugs a daunting task.

Furthermore, not all errors in formulas are due to simple typographical mistakes. In $P1$, for example, one of the errors occurs because the user was not aware that a key-value pair needs to be enclosed within curly braces ('{' and '}'). Additionally, errors in LC formulas can also be semantic mistakes. For example, if we wrap the key-value pair in $P1$ with curly braces, the compiler will no longer report a syntax error, but type errors (and arity errors) remain: the `If` function requires two or three arguments of appropriate types.

Example 2.2. The incorrect Power Fx formula $P1$ (Table 1) is an instance of the last-mile repair problem: the correct formula ($P4$) is 3 edits away from $P1$ and can be obtained by: (1) inserting a parenthesis ')' after `LunchSeminar`, (2) inserting an opening brace '{' before the key-value pair, and (3) replacing the closing parenthesis ')' after the key-value pair by a closing brace '}'. □

We now describe our framework, which allows us to generate such repairs automatically.

LaMirage Overview. LaMirage is a last-mile repair-engine generator. A repair-engine generator is a meta procedure that takes an annotated grammar approximating a target LC language and optional language-specific transformation rules and checks (which we view as constraints), and generates a repair engine that can fix last-mile errors in formulas in the target LC language. Figure 2 presents the architecture of LaMirage. Developers create a new repair engine by providing: (I1) an annotated grammar that specifies the *unreliable* terminals in the specific LC language, (I2) domain-specific insights, such as repair rules, (I3) a collection of paired well-formed and buggy formulas to train a neural error localizer, and (I4) a collection of well-formed formulas to fine tune a ranker (CodeBERT [Feng et al. 2020]). Given a faulty formula, the engine predicts error locations using the neural error localizer. The locations complement those identified symbolically. The engine then enumerates repair candidates, ranked by the token edit distance from the faulty formula. These candidates are obtained by inserting and/or deleting tokens corresponding to *unreliable* terminals in the faulty formula at the error locations identified. Ties between candidate repairs are resolved using fine-tuned CodeBERT to select the most *natural* repair to return to the user.

The repair engines created by LaMirage are syntax-directed, working as modified LL parsers, but the repairs produced are not limited to fixing syntactic errors but also common semantic errors. A LaMirage-generated repair engine works like a normal LL parser with the following modifications:

- If the parser reaches a failure state, then instead of stopping, the repair engine backtracks and attempts to insert or delete the "unreliable" tokens defined in I1. These edit operations can fix many syntactic errors, and can also induce semantic corrections (e.g., fixing function arity mistakes by re-associating call arguments using a different parenthesization).

- Each time the parser's internal state is updated, an external call to an appropriate repair rule, defined in I2, is made, which can optionally change the parser's state. These calls allow us to implement transformations that can induce more complex semantic corrections that require domain knowledge about the underlying LC formula language beyond that captured by the grammar annotations provided in I1. For example, these rules can perform basic type casts in frequently misused function calls or correct naming errors in the formula.

Example 2.3. Continuing from Example 2.2, we describe how we repair $P1$. The *unreliable* terminals in our Power Fx implementation correspond to punctuation tokens (parentheses, curly braces, brackets, dots, commas, and colons). During parsing $P1$, when the first comma is processed, a domain-specific rule is triggered that enforces the arity for the `IsBlank` call to be 1 by inserting a closing parenthesis. Thus, $P1$ is turned into $P2$. Arity analysis is one of the domain-specific insights incorporated for the Power Fx domain (I2). Other examples include rules that fix misspelled function or variable names.

The repair engine follows the steps of a regular parser until it hits the token colon ':'. At this point, the parser backtracks to the point where the d-th last reliable token was consumed. Here, d is a parameter whose value lies in $[1{-}4]$. Since `LunchSeminarVar` and `UpdateContext` (identifiers) are reliable, if $d = 2$, we backtrack to the point after `UpdateContext` is consumed. Our engine now enumerates several repairs that add and/or remove punctuation tokens. While in this example, symbolic backtracking yielded repair candidates, there may be formulas where the true error location lies in an earlier part of the formula outside the symbolic backtracking depth. In such cases, LaMirage makes use of neural error localization (described in Section 4.3.2) to predict error locations.

Following this methodology of backtracking and candidate enumeration, we generate candidates $P3$ and $P4$. Since both $P3$ and $P4$ are at a token-edit distance 3 from $P1$, we break the tie using fine-tuned CodeBERT to find $P4$ as the most natural repair. □

This example demonstrates a key insight behind LaMirage: successfully repairing last-mile errors in LC formulas benefits from a combination of symbolic and neural techniques. Without symbolic candidate enumeration, there is no *guarantee* that the repair candidates produced are valid programs in Power Fx. Without the neural re-ranking of candidate repairs, the correct result $P4$ would not be identified as the most natural repair. Finally, without neural error localization, locations far from the error state would not be considered during formula editing. As a result, all these steps play a role in producing the final correct repair.

Conceptually, this syntax-driven approach to enumerating candidate repairs and then ranking the resulting repaired formulas based on a score (in this case, the edit distance and neural ranker score) has parallels with traditional program synthesis.

3 LAST-MILE REPAIR PROBLEM

Let T be a target engine—e.g., a compiler, an interpreter, or a runtime engine—that accepts or rejects programs. Informally, given a string s, representing an ill-formed program, we seek to transform (repair) s to another nearby string $\hat{s}$ that is accepted by T. Several values of $\hat{s}$ may exist, but we want the one that the user most likely intended.

The first challenge to fixing s is to generate candidate strings that are near s and also accepted by T. In most real-world scenarios, it is impractical or expensive to query T repeatedly to identify valid repair candidates. Furthermore, we need to have a practical way to produce these candidates. Rather than considering all strings, we take inspiration from syntax-guided synthesis [Alur et al. 2013; Gulwani 2011; Polozov and Gulwani 2015] and use a context-free grammar G as a constructive approximation of T.

A context-free grammar is a quadruple $G := (V, \Sigma, R, S)$, where V is the the set of non-terminals, Σ is the set of terminals, $R \subset V \times (V \cup \Sigma)^*$ is the set of production rules, and $S \in V$ is a start symbol.

A string $s \in \Sigma^*$ is accepted by G, or in the language defined by G, denoted by $s \in L(G)$, if there exists a *derivation* of s in G. A derivation of s is a sequence of strings $S \to s_1 \to \ldots \to s_k \to s$,

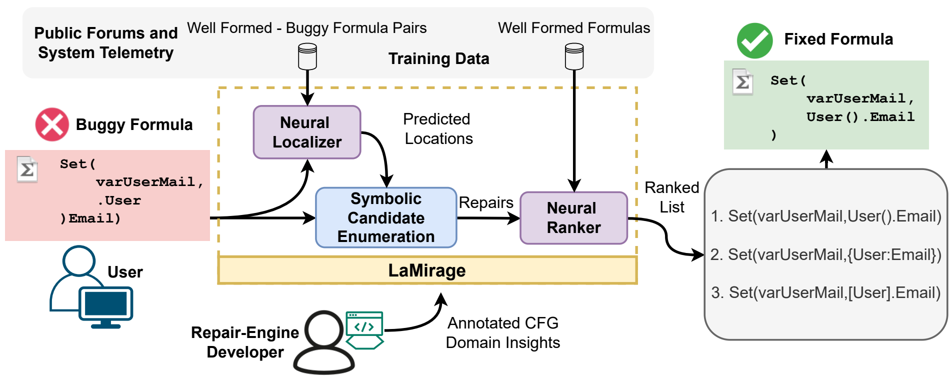

Fig. 2. LaMirage uses three key components: a neural error localizer, a symbolic candidate enumerator, and a neural ranker. The neural localizer predicts error locations that augment deterministic backtracking to perform edits far from the location where the error is identified – for efficiency, we only use predicted locations when our deterministic backtracking does not yield viable candidates. The symbolic enumerator produces candidate repairs that are guaranteed to satisfy the grammar and constraints provided by the repair engine developer. Finally, a neural ranker, trained on well-formed formulas from the domain, re-ranks candidate repairs. We train both neural models on formulas collected from public forums and system telemetry.

where each $s_i \in (V \cup \Sigma)^*$, and for each $s_i \longrightarrow s_j$, s_j is obtained by replacing a non-terminal X in s_i with $X_1, \ldots, X_n$ where each $X_i \in (V \cup \Sigma)$ and $(X \longrightarrow X_1 \ldots X_n) \in R$.

Let L_{full} be the language of all strings accepted by T. A smaller $L(G)$ implies a more efficient enumeration of candidate repairs (as the search space is smaller), but the reduction in overlap with L_{full} may place some candidate repairs out of scope. A larger $L(G)$ may increase the scope of fixes, but may also result in spurious candidates outside of L_{full}. Given a fixed size of $L(G)$, we want to maximize overlap with L_{full}, but one might allow spurious candidates if it makes writing G easier. Note that we can apply T on the final candidate repairs (after search) to filter out invalid candidates.

The second challenge is that G, by definition, can only cover context-free properties of L_{full}. To address this challenge, we introduce a set of context-sensitive constraints $\mathcal{C}$ that are satisfied by all programs in L_{full}. A constraint $C \in \mathcal{C}$ is a mapping: $\Sigma^* \mapsto \mathbb{B}$. Constraints capture requirements that (1) well-formed programs must satisfy, and (2) are not captured by G. Some examples are: (a) programs should be type correct, (b) each variable should be defined, and (c) every function or operator name should be supported by T. Together, G and the constraints $\mathcal{C}$ serve as a proxy for T.

Let $\mathcal{C}_{full}$ be the set of all constraints enforced by T. Like G, $\mathcal{C}$ is an approximation. However, unlike G, $\mathcal{C}$, in practice, is typically a sound over-approximation, meaning: $\forall s \in \Sigma^*\ \mathcal{C}_{full}(s) \implies \mathcal{C}(s) \wedge \neg\mathcal{C}(s) \implies \neg\mathcal{C}_{full}(s)$. Evaluating $\mathcal{C}$ is more efficient than checking acceptance with T (by construction), so we can use it during our candidate repair search. We present more discussion around the tradeoffs in choice of G and $\mathcal{C}$ in Section 6.

The third challenge is to model the user intent, i.e., quantifying the likelihood of a candidate repair being the one that the user intends. We want to maximize the probability $Pr(\hat{s} \mid s)$, which quantifies the probability that $\hat{s}$ is the user-intended program given they wrote s. Our assumption is that the user usually writes a program that is "close" to what they intend. To quantify "closeness", we can use any distance metric `dist` on strings. In this work, we use *token edit distance* and require a distance threshold δ that specifies the required closeness. Among the set of programs that are within a distance of δ from s, we assume that $Pr(\hat{s} \mid s)$ is proportional to the prior probability, namely $Pr(\hat{s})$, of observing $\hat{s}$. Intuitively, we want to find a string $\hat{s}$ as a repair that is "close" (according to distance function `dist` and the distance threshold δ) to the buggy program s, while making sure that it is "valid" (validated using the grammar G and the constraints C). In case of multiple such candidates, we break ties by leveraging the probability distribution of "natural" programs that real users compose in the target language, defined by Pr.

Now we can formalize the problem statement as follows. Given a grammar $G := (V, \Sigma, R, S)$, constraints $C : \Sigma^* \mapsto \mathbb{B}$, a distance measure $\texttt{dist} : \Sigma^* \times \Sigma^* \mapsto \mathbb{R}^+$, a distance threshold δ, and a string $s \in \Sigma^*$, the *last-mile repair problem* seeks to find a string $\hat{s} \in \Sigma^*$ such that $\hat{s} \in L(G)$ and $C(\hat{s})$, $\text{dist}(\hat{s}, s) \leq \delta$, and $\hat{s} = \text{argmax}_{X \in \{x \text{ s.t. } \texttt{dist}(x,s) \leq \delta\}} Pr(X)$, where Pr is the probability distribution over human-composed strings in L_{full}.

4 REPAIR-ENGINE GENERATOR

We present the LaMirage framework, our specific solution to the last-mile repair problem, and describe how it can be instantiated to generate repair engines for different LC formula languages.

The class of possible repairs can be large, and consequently, the search space of potential repairs can be enormous. Therefore, we need to focus on classes that represent a large set of common mistakes that LC users make when authoring programs in the target language. This target-specific information is captured using the concept of *unreliable terminals* and *domain-specific parser state transformations*. While unreliable terminals capture pure syntax errors users may make, domain-specific parser state transformers allow LaMirage to incorporate more semantic fixes.

Let $G := (V, \Sigma, R, S)$ be the grammar. LaMirage further assumes access to the following:

- A set $U \subset \Sigma$ of *unreliable terminals*: A subset of the terminals is classified as unreliable based on their likelihood of being erroneously omitted or included in user-authored buggy formulas. For example, in formula languages, parentheses and/or punctuation marks are observed to be unreliable as users often tend to misplace them in expressions.
- A set of *domain-specific parser state transformations*, where each transformation takes a parser state and returns a set of (modified) parser states. We will later see examples in Section 4.4.

The subset U and domain-specific transformations are specified by the language developer using LaMirage to instantiate a repair engine for their LC formula language.

Intuitively, LaMirage can be seen as a syntax-guided repair engine, generating repair candidates by performing rule-based transformations (enumerating candidate valid programs) on top of an LL parser that accepts strings in G. However, a LaMirage-generated repair engine differs from a regular LL parser in the following ways:

(1) the repair engine explores and produces multiple parses;
(2) the repair engine backtracks when a failure state is reached by the underlying LL parser;
(3) the repair engine thereafter proceeds searching over valid candidates by transforming unreliable terminals (and trusting reliable terminals to constrain the search space);
(4) at every step, the repair engine also calls parser state transformers to get new parser states that accumulate context sensitive information (e.g., number of arguments in the current call); and
(5) the repair engine limits the number of repairs it considers by tracking the edits (cost) it has already made (incurred) on the input string.

Next, we formalize this intuition of an error-correcting parser.

4.1 Parser States

We describe the error-correcting parser using inference rules. The inference rules operate on parser states. Given a grammar $G := (V, \Sigma, R, S)$, a *parser state* is a 4-tuple $\langle A, T, p, c \rangle$, where A is the parsing stack, T is the stream of remaining tokens that need to be processed, p is the parse-tree constructed so far, and c is the cost of the state. The parsing stack A is represented as a list, with the first element of the list corresponding to the top of the stack. Similarly, T is also represented as a list, with the

$$\begin{aligned}
First(t) &= \{t\} && \text{if t is a terminal}\\
First(X) &= First(\gamma_1) \cup \ldots \cup First(\gamma_n) && \text{if } \forall i : X \rightarrow \gamma_i \in R\\
First(s\gamma) &= First(s) \cup (First(\gamma) \text{ if s can derive } \epsilon \text{ else } \emptyset)\\
Follow(X) &= First(\delta) \cup (Follow(Y) \text{ if } \delta \text{ can derive } \epsilon \text{ else } \emptyset) && \text{for all } Y \rightarrow \gamma X \delta \text{ rules in the grammar}
\end{aligned}$$

Fig. 3. Fixed-point equations for the standard *First* and *Follow* helper functions used in an LL parser for grammar $G := (V, \Sigma, R, S)$.

$$\textsc{T-Terminal-Match}\ \frac{a = t \land a \in \Sigma \cup \{\$\}}{\langle a : A, t : T, p, c\rangle \rightarrow \langle A, T, p, c\rangle} \qquad \textsc{T-Accept}\ \frac{}{\langle [], [], p, c\rangle \rightarrow accept}$$

$$\textsc{T-Non-Terminal-Expansion}\ \frac{\begin{array}{c}(a \in V) \land (a \rightarrow X_1 \ldots X_k) \in R \qquad t \in First(X_1 \ldots X_k) \lor (t \in Follow(a) \land \epsilon \in First(X_1 \ldots X_k))\\ p' \text{is obtained from p by adding } X_1 \ldots X_k \text{ as children of the leftmost leaf } a \text{ in p}\end{array}}{\langle a : A, t : T, p, c\rangle \rightarrow \langle [X_1, \ldots, X_k]\text{++}A, t : T, p', c\rangle}$$

Fig. 4. Transition rules for the syntax-guided repair engine states given a grammar $G := (V, \Sigma, R, S)$. Definitions for *First* and *Follow* are provided in Figure 3. The end-of-sequence token is denoted by \$. Symbols : and ++ correspond to Haskell-style list prepend and append syntax.

first element being the next immediate token. For convenience, we use $Stack(x)$, $RemTokens(x)$, $ParseTree(x)$, and $Cost(x)$ to refer to the parsing stack, remaining tokens, the parse-tree, and the cost of a search-state x respectively.

Let $G := (V, \Sigma, R, S)$ and the input string be s. Let `toks` denote the tokenization of s represented as a list of tokens. The initial state of the error-correcting parser is $([S, \$], \texttt{toks}, p_0, 0)$, where \$ is the end-of-sequence symbol, p_0 is parse tree containing just a single (root) node S, and cost is 0. Starting from this initial state, the inference rules describe how states are updated. In some cases, multiple rules may be applicable, or the same rule may result in multiple states. In these cases, the interpretation is that of a *non-deterministic choice* - the actual implementation considers *all possibilities* and explores all states, providing a completeness guarantee with respect to G. The top-level algorithm, shown as Algorithm 1, is just a specific strategy for applying these inference rules. The goal is to start with the initial state and reach the special state, *accept*, that is the terminating state for the algorithm. Any state of the form $([], [], p, c)$, for any parse tree p and cost c, rewrites to the *accept* state (Figure 4, detailed in Section 4.2).

Informally, starting from $([S, \$], \texttt{toks}, p_0, 0)$, where `toks` is a tokenization of input string s, if we reach $([], [], p, c)$, then p will be a parse of some string $\hat{s}$ obtained by repairing s, and c will be $\texttt{dist}(\hat{s}, s)$ (the cost of the repair).

4.2 Repair Algorithm

Algorithm 1 describes the overall approach. At a high level, we explore the search space using the transition rules in Figure 4 and maintains a priority queue of states, ordered by their costs (Line 4).

The priority queue is initialized to contain just the initial state (Line 3). Every time it encounters a state corresponding to an accept state, it translates the parse-tree into a repair and returns it to the user (Lines 8-9). Otherwise, it applies domain-specific strategies to obtain a set of new states in Line 10. Domain-specific strategies are explained later in Section 4.4. The next step is to compute

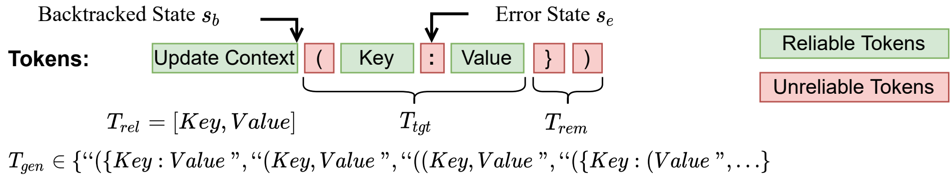

Fig. 5. Illustration of repair enumeration, along with token reliability and deterministic backtracking.

the set of next states using the transition rules from Figure 4 (Line 11). There are two conditions for the state to be considered an error state (Line 12). The set of next states (N) can be empty, in which case no progress can be made without making edits, or the state is part of the set of states predicted as possible error states by our neural error localizer (detailed later in Section 4.3.2). If the state is considered to be an error state, the algorithm then goes into *repair mode*. It uses a sub-procedure *EnumerateRepairs* (detailed in Section 4.3.3) to apply a correction on the error state and put it back in the search queue (Line 13).

The transition rules in Figure 4 describe a standard LL parser. The transition T-Terminal-Match handles the case when the top of stack is a token that matches the next token in the input stream. The transition T-Non-Terminal-Expansion replaces a nonterminal X at the top of stack by the list of elements from the right-hand side of some production rule based on the lookahead. The standard helper functions, *Follow* and *First*, used in this rule are obtained using fix-point computation over the equations shown in Figure 3. Note that these two functions are pre-computed for a grammar.

Algorithm 1 Overall Repair Engine Algorithm. Given the list of tokens corresponding to a buggy string s, and a grammar G, it returns a list of repairs. The \$ symbol corresponds to the end-of-stream token. *GenRepair*(s) converts the parse tree of state s back into a string. The optional S_{pred} correspond to states associated with error locations predicted by our neural error localizer – which are only used when deterministic backtracking does not yield viable repair candidates.

1: **procedure** *Repair*(toks, G, $S_{pred} = \emptyset$)
2: $p_0 \leftarrow$ tree with a single node for *StartSymbol*(G)
3: $s_0 \leftarrow \langle [StartSymbol(G)\$], \text{toks}, p_0, 0 \rangle$ ▷ Initial State
4: $P \leftarrow$ An empty priority queue
5: Insert s_0 into P with cost 0.
6: **while** P is not empty **do**
7: Pop s from P
8: **if** $s \longrightarrow accept$ **then** ▷ Rules in Figure 4
9: **yield** *GenRepair*(s)
10: $S_D \leftarrow ApplyDomainStateTransformers(s)$
11: $N \leftarrow \{s'' \mid s' \longrightarrow s'' \wedge s' \in S_D\}$ ▷ Rules in Figure 4
12: **if** N is empty or $s \in S_{pred}$ **then**
13: N $\leftarrow EnumerateRepairs(s)$
14: **for each** s' in N **do**
15: Insert s' into P with cost $Cost(s')$

We extend this initial rule set with those in Figure 7 to convert the LL parser into a syntax-guided repair candidate enumerator, which LaMirage uses to produce repair candidates (which then need to be ranked). Algorithm 2 presents a strategy for applying these new extension rules.

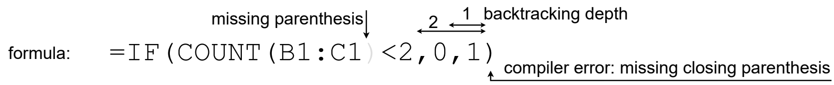

Fig. 6. The true location of an error may be relatively far from where the compiler detects an error. For example, in this formula the missing parenthesis is not identified until the end of the formula, as COUNT is variadic. LaMirage uses a pointer-network to predict such error locations.

4.3 Repair Candidate Enumeration

Algorithm 1 calls *EnumerateRepairs* to *repair* an error state, as described in Algorithm 2.

We use $\tau(s) = \langle s_0, \dots, s \rangle$ to refer to the *trace* of s i.e. to the sequence of states resulting from repeated application of the transition rules in Figure 4 starting from the initial state s_0, as detailed in Algorithm 1, and ending at s. Note that the trace is just a history of parser states that we assume the algorithm saves. We say a token t is *unreliable* if `t.value == a` for some unreliable terminal a; i.e., when the value attribute of the token corresponds to an unreliable terminal.

Given an erroneous search state s_e, we first identify the portion of input processed so far that can be edited to induce candidate repairs. LaMirage accomplishes this using deterministic backtracking.

4.3.1 Deterministic Backtracking. To produce the editable formula range, LaMirage deterministically *backtracks* to an ancestral state $s_b \in \tau(s_e)$. The state s_b corresponds to the state in $\tau(s_e)$ where the d^{th} previous *reliable token* was added to the parse-tree of s_e, or if no such state exists, s_b is the first state in $\tau(s_e)$. Consider the example in Figure 5. The error state corresponds to the point after `Key` is consumed. With $d = 2$, the backtracked state corresponds to the point right after `UpdateContext` was consumed. The constant d here refers to the *backtracking depth*, which controls the extent to which we can reach and edit previously processed tokens.

We use T_{tgt}, T_{rem} and T_{rel} to refer to three special lists of tokens. Let t_r be the first reliable token in $Tokens(s_e)$. T_{tgt} is the list of tokens in $Tokens(s_b)$ up until and including t_r (note that $Tokens(s_e)$ is guaranteed to be a suffix of $Tokens(s_b)$). T_{rem} is the list of tokens after, and excluding t_r in $Tokens(s_e)$. Finally, T_{rel} is the list of all reliable tokens, in order, in T_{tgt}. This logic is encapsulated within the *Backtrack* procedure in Line 2 of Algorithm 2. Consider the example in Figure 5. T_{tgt} corresponds to the four tokens `(`, `Key`, `:`, and `Value`. T_{rem} is the list of tokens following `Value`. T_{rel} is the list of two reliable tokens in T_{tgt} - `Key` and `Value`.

Tokens in T_{tgt} that are unreliable (i.e. not in T_{rel}) are candidates for editing during candidate enumeration. Meanwhile, tokens outside of T_{tgt}, such as tokens in T_{rem}, remain untouched.

Scoping edits to a subset of tokens in T_{tgt}, in contrast to considering the entire program prefix, helps constrain the search space for candidate repairs, allowing LaMirage to quickly produce viable repairs. While deterministic backtracking is efficient, it is not complete: there may be required edits outside of the range identified. LaMirage uses neural techniques to address this challenge.

4.3.2 Neural Error Localization. Compilers often raise errors at locations far away from the actual source of mistake [Traver 2010]. In Figure 6, the user forgot a closing parenthesis for the COUNT function in their Excel formula. However, the Excel compiler will parse the formula until the end-of-stream token is encountered, and will *then* raise a missing parenthesis error. Why is no error reported earlier? The function COUNT is variadic, so the compiler greedily accepts the string contents as valid arguments for the function call. A purely symbolic repair technique might rely on backtracking to identify possible repair locations, however, as the formula grows, so does the backtracking depth required. This increase in depth can substantially increase the search space. To mitigate this problem, we leverage neural methods to complement our deterministic backtracking.

We train a Pointer Network [Vinyals et al. 2015] to predict error locations in arbitrary length formulas by learning distributions over input tokens, which has been shown in previous work [Vasic et al. 2018] to be effective for error localization in general purpose programming languages. Specifically, we take a corpus of well-formed formulas in the corresponding language and then generate broken variants by introducing synthetic errors. These errors are introduced by randomly adding, deleting, or changing unreliable tokens in the formula. We then train the pointer network to predict the locations (token indices) where these edits were performed as a function of the broken formula. At prediction time, we take the top 5 locations predicted by the network, though in practice the network mostly predicts 1 or 2 locations. For each predicted location, we take the parsing state up to that location when processing the input and treat it as an error state s_e. We can then apply our deterministic backtracking strategy starting from each such state. As a result the pointer network need not be perfect to still provide useful enumeration locations, in contrast to prior work [Vasic et al. 2018] that jointly localizes and repairs. The symbolic candidate enumerator can then perform edits as before to yield new candidates.

To mitigate the increase in the size of the search space as a result of additional candidate edit locations, LaMirage employs a *fall-back* strategy. Specifically, LaMirage first attempts to repair a buggy program using only the deterministic backtracking strategy, and if no viable repair candidates are produced, LaMirage then employs the neural error localizer to predict error states. These neurally predicted error states augment the set identified by the deterministic approach and are passed transparently to the candidate enumeration algorithm *EnumerateRepairs*.

This use of the neural localizer mirrors previous lines of work in program synthesis and program repair that integrate model-based approaches to restrict or prioritize elements in their search space, for example [Long et al. 2017; Yu et al. 2019]. The novelty in our approach is in showing a simple pointer-network-based localizer can work well in the low-code domain, particularly when employed in a fall-back strategy.

4.3.3 Candidate Enumeration with Guarantees. *EnumerateRepairs* returns all states s_r that are the same as s_b except for their remaining token sequences, where T_{tgt} is replaced by a new sequence of tokens T_{gen} (Line 10). T_{gen} satisfies the constraint that it can be obtained from T_{tgt} by inserting and/or deleting unreliable tokens. Additionally, the parsing stacks of the repaired states s_r are guaranteed to be able to derive the string corresponding to T_{gen} as a prefix. Furthermore, the modified states are guaranteed to satisfy custom checks (as they are applied prior to returning candidates). These guarantees are a key benefit of LaMirage compared to purely neural alternatives. Figure 5 shows a few possibilities for T_{gen} that are generated by the algorithm.

At the core of *EnumerateRepairs* is the ability to enumerate valid T_{gen}. This is achieved by repeated application of the transition rules in Figure 7. The first rule states that if the top of the stack is an unreliable terminal, add it to the generated token sequence so far. The second rule covers the case when the top of the stack is reliable; in this case it must match against the next expected reliable token as per T_{rel}. The third rule governs the production rules chosen to expand the top of the stack when it is a non-terminal. This is analogous to the non-terminal rule in the standard parsing transition rules in Figure 4. The only difference is the use of *FirstReliable* and *FollowReliable*. These two functions are similar to their standard counterparts *First* and *Follow* used in Figure 4. The only difference is that they are only concerned with reliable terminals, not all terminals.

However, recall that, as per the last-mile repair definition, we do not want just any T_{gen} that is possible. It must be within some edit-distance of the original. Thus we ensure that the edit-distance (Lines 9-11), or its lower-bound estimate (Lines 16-17) is less than the hyper-parameters *MaxGlobalCost* and *MaxLocalCost*, which restrict the maximum edit distance.

$$\textsc{T-Unreliable-Terminal}\quad \frac{a \in \Sigma \wedge a \in U}{\langle a : A, T_{gen}, T_{rel} \rangle \longrightarrow \langle A, T_{gen}{+}{+}[a], T_{rel} \rangle} \qquad \textsc{T-Reliable-Terminal}\quad \frac{a \in \Sigma \wedge a \notin U \wedge a = t_{rel}}{\langle a : A, T_{gen}, t_{rel} : T_{rel} \rangle \longrightarrow \langle A, T_{gen}{+}{+}[a], T_{rel} \rangle}$$

$$\textsc{T-Non-Terminal}\quad \frac{(a \in V) \wedge (a \longrightarrow X_1 \dots X_k) \in R \qquad (t_{rel} \in \mathit{FirstReliable}(X_1 \dots X_k)\ \vee (t_{rel} \in \mathit{FollowReliable}(a) \wedge \epsilon \in \mathit{FirstReliable}(X_1 \dots X_k)))}{\langle a : A, T_{gen}, t_{rel} : T_{rel} \rangle \longrightarrow \langle [X_1, \dots, X_k]{+}{+}A, T_{gen}, t_{rel} : T_{rel} \rangle}$$

$$\textsc{T-Accept}\quad \frac{}{\langle a : A, T_{gen}, [] \rangle \longrightarrow \mathit{accept}}$$

Fig. 7. State transition rules for *EnumerateRepairs* given a grammar $G := (V, \Sigma, R, S)$ and unreliable terminals $U \subset \Sigma$. Definitions for *FirstReliable* and *FollowReliable* are provided in Figure 8. Symbols : and ++ correspond to Haskell-style list prepend and append syntax.

Algorithm 2 Enumerating repairs given an error state s_e

```
procedure EnumerateRepairs(s_e)
    s_b, T_rel, T_tgt, T_rem ← Backtrack(s_e)
    P ← An empty queue
    Insert ⟨Stack(s_b), [], T_rel⟩ into P.
    repairs ← empty list
    while P is not empty do
        Pop ⟨A, T_gen, T_rel⟩ from P
        if ⟨A, T_gen, T_rel⟩ → accept then                         ▷ Figure 7
            c_r ← EditDist(T_gen, T_tgt) + Cost(s_b)
            s_r ← ⟨Stack(s_b), T_gen++T_rem, ParseTree(s_b), c_r⟩
            if c_r ≤ MaxGlobalCost then
                Add s_r to repairs
            continue
        NextStates ← {x | ⟨A, T_gen, T_rel⟩ → x}                     ▷ Figure 7
        for each ⟨A′, T′_gen, T′_rel⟩ ∈ NextStates do   ▷ Compute lower-bound on edit-distance
            c ← EditDist(T′_gen, T_tgt)
            if c ≤ MaxLocalCost then
                Append ⟨A′, T′_gen, T′_rel⟩ to P
    return repairs
```

4.4 Domain-Specific Parser State Transformers

The repairs generated by Algorithm 2 are guaranteed to satisfy G. However, valid formulas in the target language must also satisfy constraints C, which capture semantic properties, such as correct typing or using only defined variable names. Fixing formulas that do not satisfy these constraints often requires context-sensitive information and additional knowledge about the underlying LC formula language beyond that reflected in G. A key innovation of LaMirage, in contrast to state-of-the-art symbolic repair systems like grmtools, is the increase in repair expressiveness – which can capture some semantic errors – enabled by our use of C.

$$
\begin{aligned}
\mathit{FirstReliable}(t) &= \{t\} \text{ if t is a terminal and t is reliable} \\
\mathit{FirstReliable}(t) &= \emptyset \text{ if t is a terminal and t is unreliable} \\
\mathit{FirstReliable}(X) &= \mathit{FirstReliable}(\gamma_1) \cup \ldots \cup \mathit{FirstReliable}(\gamma_n) \text{ where } X \to \gamma_1 \ldots X \to \gamma_n \text{ are production rules} \\
\mathit{FirstReliable}(s\gamma) &= \mathit{FirstReliable}(s) \cup (\mathit{FirstReliable}(\gamma) \text{ if s can derive } \epsilon \text{ else } \emptyset) \\
\mathit{FollowReliable}(X) &= \mathit{FirstReliable}(\delta) \ \cup (\mathit{FollowReliable}(Y) \text{ if } \delta \text{ can derive } \epsilon \text{ else } \emptyset) \\
&\quad \text{for all } Y \to \gamma X \delta \text{ rules in the grammar}
\end{aligned}
$$

Fig. 8. Fixed-point equations for *FirstReliable* & *FollowReliable*.

$$
\begin{aligned}
\text{FuncCall} &::= \text{FuncName (ArgsList)} & \text{FuncName} &::= \text{ident} \\
\text{ArgsList} &::= \epsilon \mid \text{Arg ArgsListTail} & \text{ArgsListTail} &::= \epsilon \mid \text{, Arg ArgsListTail} \\
\text{Arg} &::= \text{Expr} & \text{Expr} &::= \text{Var} \mid \text{Constant} \mid \text{BinaryExpr} \mid \ \ldots \\
\text{Var} &::= \text{ident}
\end{aligned}
$$

Fig. 9. Fragment of grammar used for Excel/Power Fx corresponding to function calls and expressions.

We introduce the concept of *domain-specific repair strategies* via *parser state transformers* to tackle these classes of errors, as captured by the call to *ApplyDomainStateTransformers* in Line 10 of Algorithm 1. Transformers are essentially a collection of symbolic transition rules, that create new parser states from a given state, or flag a state as an *error state*, denoted as $\perp$. Given the input parser state, the *ApplyDomainStateTransformers* function simply returns the set of states obtained by applying all the eligible transition rules, or an empty set if the input state is flagged as an error state by *any* of the rules. To support different use cases, LaMirage supports transformers that are called at different points in the input processing procedure. For example, error state transformers are called when an error state is raised, other transformers may trigger on a particular token/rule.

The transition rules for various strategies that can be implemented in our framework are listed in Figure 10. All the rules are based on the grammar fragment described in Figure 9. This fragment captures function calls and basic expressions as allowed by the LC domains of Excel and Power Fx. Next, we explain each of our strategies individually.

Arity Analysis. Most formulas in LC languages use built-in functions, which have a fixed minimum and maximum arity. The `IsBlank` function in the motivating example in Figure 1 has a minimum and maximum arity of 1, and thus repairing P1 involves inserting a parenthesis after the first argument to `IsBlank` which is `LunchSeminar`.

The rule T-Arity in Figure 10 captures the arity analysis strategy that allows for such repairs. Essentially, given an input parser state, it first computes the current unclosed function f, and the number of arguments parsed so far for f, denoted by n, by analyzing the parse-tree p using the convenience functions *CurFunc* and *CurNumArgs* respectively. Then, it flags the input parser state as an error state if the top of the stack is either ArgsList or ArgsListTail (Figure 9) and one of two cases hold: (1) $n \geq \mathit{MaxArity}(f)$, and the next token is going to force the parse of another argument, and (2) $n < \mathit{MinArity}(f)$, and the next token does not indicate the start of a new argument. Whether or not a new argument is going to be parsed can be checked by checking the membership of the kind of the next token against the *Follow* set of the non-terminal at the top of the stack, as described in the rule T-Non-Terminal-Expansion in Figure 4.

$$\textsc{T-Arity}\quad \frac{a \in \{\text{ArgsList}, \text{ArgsListTail}\} \qquad f = \mathit{CurFunc}(p) \qquad n = \mathit{CurNumArgs}(p, f) \\ (n \geq \mathit{MaxArity}(f) \wedge t.\mathit{kind} \notin \mathit{Follow}(a) \vee \qquad n < \mathit{MinArity}(f) \wedge t.\mathit{kind} \in \mathit{Follow}(a))}{\langle a : A, t : T, p, c\rangle \longrightarrow \bot}$$

$$\textsc{T-Combine-Tokens}\quad \frac{\text{Tokens } t_1 \text{ and } t_2 \text{ are eligible to be combined into } t_3 \qquad \text{with cost } c'}{\langle A, t_1 : t_2 : T, p, c\rangle \longrightarrow \langle A, t_3 : T, p, c + c'\rangle}$$

$$\textsc{T-Symbol}\quad \frac{a \in \{\text{FuncName}, \text{Var}\} \qquad t.\mathit{val} \notin \mathit{AvailableSymbols} \qquad \langle t', c'\rangle \in \mathit{AvailableCorrections}(t.\mathit{val})}{\langle a : A, t : T, p, c\rangle \longrightarrow \langle a : A, t' : T, p, c + c'\rangle}$$

$$\textsc{T-Symbol-Fail}\quad \frac{a \in \{\text{FuncName}, \text{Var}\} \qquad t.\mathit{val} \notin \mathit{AvailableSymbols} \qquad \mathit{AvailableCorrections}(t.\mathit{val}) = \emptyset}{\langle a : A, t : T, p, c\rangle \longrightarrow \bot}$$

$$\textsc{T-Typing}\quad \frac{p^t = \mathit{ComputeTypes}(p) \qquad p^t \text{ has typing errors} \qquad \langle p', c'\rangle \in \mathit{ComputeTypeRepairs}(p^t)}{\langle A, T, p, c\rangle \longrightarrow \langle A, T, p', c + c'\rangle}$$

$$\textsc{T-Typing-Fail}\quad \frac{p^t = \mathit{ComputeTypes}(p) \qquad p^t \text{ has typing errors} \qquad \mathit{ComputeTypeRepairs}(p^t) = \emptyset}{\langle A, T, p, c\rangle \longrightarrow \bot}$$

Fig. 10. Inference rules for various domain-specific, context-sensitive repairs based on the grammar fragment in Figure 9. The symbol $\bot$ denotes an error state.

How does this help in repair? Since the input parser state is flagged as an error state, this triggers *EnumerateRepairs* in Algorithm 1 in Line 13. Thus, this strategy will be able to fix arity errors by inserting/deleting unreliable tokens (punctuation).

Combining Tokens. Another common class of errors in both our domains involves incorrect tokenization due to presence of extra whitespace. For example, in Excel, if a space is included between the < and = symbols, the whole string is tokenized into two separate tokens, instead of the more likely <= (less-than-equal) token, and will raise a syntax error (reported by the compiler as "missing operand"). In both our domains, these binary operators are reliable terminals/tokens; hence Alg. 2 can't generate repairs for such errors on its own and requires a domain specific strategy.

The strategy is formalized as the rule T-Combine-Tokens in Figure 10. Essentially it says that if the next two tokens are eligible to be combined into a new token with cost c', return a new state where the two are combined and c' is added to the cost. Thus, this strategy directly modifies the remaining token stream. In our implementation, we use this rule to combine tokens corresponding to relational operators such as <=, ==, and >=.

Fixing Symbol Errors. This strategy helps generate repairs for errors where function names or variables are misspelled (`IsBlnk` instead of `IsBlank`), or synonyms for functions are used, such as the use of `Length` instead of `Len` in the second example in Figure 1.

The strategy is formalized as the rule `T-Symbol` in Figure 10. If the top of the stack corresponds to the non-terminals FuncName or Var in the grammar in Figure 9, and the next token t's value is not present in the set of available symbols, denoted by *AvailableSymbols*, and a corrected token t' is available with cost c', then a new state is returned where the next token is replaced with

t'. The set of symbols *AvailableSymbols* can be determined from the runtime context within the GUI/IDE interface to the underlying domain. The function *AvailableCorrections* returns available symbols within a threshold edit-distance of the value of the original token t, which captures the misspelling case, and built-in functions that are known synonyms of the value of t. If no such correction is available, then the state is flagged as an error state (rule T-Symbol-Fail). Note that this will invoke *EnumerateRepairs* but none of its repairs would fix the issue and overall, no repairs would be returned by Algorithm 1.

Fixing Type Errors. The final strategy helps generate repairs for typing errors. Specifically, types are computed for the parse-tree of the input parse state, and if there is a type error, one of two things can happen: (1) if a repair is available in terms of a fixed parse-tree with cost c', then a new state is returned with the fixed parse tree and an additional cost of c', and (2) if a repair is not available, the state is flagged as an error state. The scenarios are captured by rules T-Typing and T-Typing-Fail in Figure 10 respectively. Note that a repair on a parse-tree must apply to a node which is completely parsed i.e. it has no non-terminal leaf nodes. An example of a repair is explicit type conversion, such as converting an int to a string to enable concatenation with another string.

Ease of Use. LaMirage enables repair-engine developers to create new domain-specific strategies with a few lines of code – this increase in repair expressiveness beyond syntax fixes, despite being syntax-guided, is a key contribution of LaMirage. For example, the equality operator in Excel (=) is different from that in other languages such as Python and Java (==); confusing these is a common mistake for some Excel users. To implement this DSS, a developer need only define a few constants and conditions; *Trigger token*: "=", *condition*: "Next token in the input stream is also =", and the *Transformation*: "skip the next = token from the input stream". Most such DSS are reusable.

4.5 Ranking Repairs by Naturalness

Algorithm 1 is guaranteed to return the repairs in increasing order of cost, which is the edit-distance from the original buggy program in our implementation. However, a scenario may arise when there are multiple repairs within the allowable edit-distance, in which case we need to select the most *natural* repairs to show to the user. We fine-tune a pre-trained language model, CodeBERT [Feng et al. 2020], to approximate the probability that a formula would be written by a user. CodeBERT is pre-trained on millions of aligned natural language and code snippets across programming languages such as Python, and Java, so we need to fine-tune it for our LC domains.

A simple way to fine-tune CodeBERT would be to train it with the *causal-LM* objective [Dai and Le 2015] i.e. train it to predict a formula one token at a time, by taking into account the tokens generated so far. Then the product of the associated probabilities with every generated token can be used for ranking the formula. Since the bulk of our algorithm is focused towards producing repairs involving insertion/deletion of unreliable tokens, we can restrict the causal-LM objective to only train the model to predict contiguous unreliable sequences given the list of tokens before and after the target sequence. An issue arises here: the frequency distribution of unreliable tokens is highly skewed: e.g., a parenthesis occurs more frequently than a curly brace. Thus training with this objective over the available well-formed formulas introduces a bias in the models towards more frequently occurring tokens and will not work well, in our experience. To address this, we introduce a new setup that mitigates this bias. Specifically, we break this task further into predicting a single unreliable token given the prefix and suffix lists of tokens, turning it into a classification task, where we can appropriately balance the training dataset by undersampling and oversampling as necessary. To rank repairs, we sum the negative log-probabilities of predicted tokens, and use this to break edit-distance ties. In our experience, this modified setup worked well for both domains and allowed us to use relatively modest amounts of data during training.

Now that we have our full approach, Theorem 4.1 summarizes the properties of our repairs.

Theorem 4.1. *Let G be the grammar provided to LaMirage and C be the set of domain specific constraints provided to complement G. Let L_{full} be the language of inputs accepted by T, the target execution engine. Jointly, G and C approximate L_{full}, as described in Section 3. Let $\hat{s}$ be a repair returned by LaMirage. Let $\texttt{dist} : \Sigma^* \times \Sigma^* \mapsto \mathbb{R}^+$ be an edit distance metric between two strings. Let $\delta \in \mathbb{R}^+$ be the maximum permissible edit distance. Then $\hat{s} \in L(G) \wedge C(\hat{s}) \wedge (\texttt{dist}(s, \hat{s}) \leq \delta)$. If a ranking model is used and has learned the appropriate prior distribution over L_{full}, then $\hat{s}$ maximizes $Pr(\hat{s})$.*

5 EVALUATION

We present the results of our empirical evaluation of an implementation of LaMirage on two critical LC domains: Excel and Power Fx. First, we describe our experimental setup and methodology, including a description of our datasets and the baselines we compare against. We also carry out a set of ablation studies (Section 5.4) to evaluate the impact of different design decisions on LaMirage.

5.1 Benchmarks and Datasets

We evaluate performance on two LC languages with significant user bases: Excel and Power Fx.

Benchmarking. We created a benchmark set of 200 Excel formulas by gathering buggy formulas listed in the publicly available third-party Excel forum MrExcel [MrExcel Message Board 2021]. We performed a similar collection of 100 Power Fx formulas from the official PowerApps help forum [Power Apps Community 2021]. We then added 100 Power Fx formulas collected by the PowerApps team at Microsoft, who used basic system telemetry to passively log anonymized Power Fx formulas written by real anonymized users. These anonymized formulas replace all user content with dummy values and only preserve names associated with built-in (available to all) functions. The combined set of 200 Power Fx buggy formulas (and their groundtruth solutions) constitute our Power Fx benchmark. We manually annotated the ground truth for all formulas.

To avoid introducing bias to the selection of formulas for our benchmarks, we adhered to the following procedure. For forum sourced benchmarks, we sampled uniformly at random from formulas scraped that did not pass the domain's parser/analyzer. For telemetry benchmarks in the case of Power Fx, we sampled 500 formulas that did not pass Power Fx's analyzer – we removed any broken formula that was incomplete (i.e. it was the result of a user still in the process of writing). For both forum and telemetry formulas, we removed formulas for which we could not unambiguously determine the ground truth solution manually. The final collection of formulas include error such as: unmatched delimiters, invalid function call syntax (including invalid spaces, extra commas, and supurious symbols), invalid function call arity (both excess and insufficient arguments), incorrect types, malformed references (such as sheet, cell, and range references in Excel), malformed records (in Power Fx), invalid operator uses (including operators without operands or invalid use of an operator in a function call), malformed relational operators (such as using incorrect syntax for a comparison operation), and inappropriate string quoting. As part of our contribution, we are releasing these benchmarks.

For our evaluation, we consider a candidate repair to be successful if it matches the ground-truth formula (after normalizing for white-space and casing, whenever not relevant).

Training datasets. To produce training data for methods that require it, we relied on MrExcel forum posts and PowerApps forum posts. We restrict ourselves to training data that is disjoint from the formulas used in our evaluation benchmarks. We extracted formulas present in user posts. To improve the quality of Power Fx formulas collected, in that domain we restrict ourselves to text in <code> HTML tags. We perform semi-automated curation using manually written rules to remove partial formulas or inputs outside of the language domain targeted. We use a native parser for Excel and Power Fx, respectively, to label formulas collected as parseable or un-parseable.

From this collection, we prepared 267,653 Excel formulas and 29,154 Power Fx formulas that satisfy the domain's analyzer and can be used for language modeling and baseline training. In addition, we collected 27,501 Excel formulas and 1,183 Power Fx formulas that are rejected by the domain's analyzer and are used during baseline training.

5.2 Baselines

We compare LaMirage to state-of-the-art symbolic and neural approaches. We produce up to 50 candidate repairs with each system. We report the number of benchmarks that are successfully repaired if we consider the top 1, 3, and 5 candidate repairs produced. We base our cutoffs (up to 5 candidates) on working memory capacity [Cowan 2001] which has been applied for other recommendation systems [Henley 2018], though more studies are warranted.

Our configuration. Our evaluation implementation of LaMirage includes domain specific strategies for arity analysis and fixing symbol errors for both Excel and Power Fx, combining tokens strategy for Excel, and repairing ill-formed cell references in Excel. In terms of ranking, our implementation of LaMirage ranks candidate repairs lexicographically based on their edit-distance and language-model score. For neural error localization, we use the implementation as described in Section 4.3.2. We train the Excel and Power Fx neural error localizers on pairs of original formula and synthetically broken formula, where we introduce at most 3 and 5 errors, respectively.

We set the local and global edit distances for LaMirage to 3 and a per-formula timeout of 10 seconds, such that we only consider LaMirage candidates produced within the timeout.

Symbolic approaches. We consider a state-of-the-art symbolic error recovery system and the publicly available error recovery in Excel desktop. No such feature is available for Power Fx.

grmtools. We use grmtools, a parser framework that exposes the official implementation of a symbolic state-of-the-art parsing recovery technique [Diekmann and Tratt 2020]. grmtools produces a set of one or more edit operations at one or more locations in the original input code. When there are multiple locations, grmtools applies the first edit operation at location i before generating the set of repairs at location $i + 1$. To produce a repaired candidate, we take the top edit operation at each location and apply it to the input formula. grmtools does not rank repairs, as long as they have the same edit distance, and produces the edit operations in non-deterministic order for each location. For each formula, we run grmtools 50 times and take the set of repairs returned in the given order. This approach is based on correspondence with the authors (via Github issues) for approaches to enumerate more than 1 repair candidate. To account for non-determinism, we repeat our grmtools evaluation 10 times and report the best performance across cut-offs.

Excel-Desktop Error Recovery. We compare against the error recovery provided by Excel desktop Version 2203, which can correct errors such as adding missing closing parentheses.

Neural approaches. We compare to three neural systems that represent two popular approaches in neural program repair: 1) task-specific models and 2) large pre-trained language models.

Break-It-Fix-It (BiFi). BiFi iteratively trains an encoder-decoder-based neural code fixer and breaker – the latter is used to improve the fixer that generates repair candidates. Both the fixer and breaker are implemented using transformers [Vaswani et al. 2017]. We refer the interested reader to the associated paper [Yasunaga and Liang 2021] for more details. We train BiFi on our Excel and Power Fx data and set the maximum generation length to 10 tokens beyond the input length.

Table 2. *More repaired formulas:* LaMirage can successfully repair more formulas than baseline approaches across all top-K cutoffs in both domains. For Codex and Codex-Edit we combine results across temperatures, reporting the best value for each cutoff. The temperature with best performance for top-1, top-3, and top-5 correspond to 0.3, 0.7, 0.7 (Codex in Excel); 0.4, 0.7, 0.7 (Codex in Power Fx); 0.1, 0.3, 0.4 (Codex-Edit in Excel); and 0.0, 0.4, 0.5 (Codex-Edit in Power Fx). The results for Codex and Codex-Edit were last obtained on 14 April 2022 using OpenAI's public API. We also report the median time (in milliseconds) to produce the repair candidates for each formula. While LaMirage can produce its candidate repairs faster than neural systems, the purely symbolic grmtools produces repairs fastest.

System	Type	Excel				PowerFx			
		Top-1	Top-3	Top-5	Time	Top-1	Top-3	Top-5	Time
Excel-Desktop	Symbolic	83	83	83	-	-	-	-	-
grmtools	Symbolic	97	104	108	**13.6**	98	110	113	**17.2**
BiFi	Neural	115	130	134	363.1	34	45	48	592.8
Codex	Neural	111	156	160	1651.8	86	117	132	1997.9
Codex-Edit	Neural	147	163	165	5806.6	106	137	140	6417.6
LaMirage	NeuroSymbolic	**174**	**182**	**182**	32.1	**170**	**177**	**177**	134.4

Codex. We use OpenAI's REST API to conduct experiments with Codex [Chen et al. 2021], a state of the art large language model trained on code. We provide the following prompt to Codex:

```
##### Fix bugs in the below code \n ### Buggy <domain> \n <buggy-code> \n ### Fixed <domain>
```

where we replace `<domain>` with Excel or Power Fx, and the `<buggy-code>` with the formula we want to repair. To use Codex's few-shot learning abilities, we include three manually written examples of buggy and fixed code from the appropriate domain chosen to cover common mistakes. We rank Codex produced repairs based on their average per-token log probability.

The three examples cover common mistakes: missing parentheses, extra parentheses, and extra commas. Note that we also experimented with zero-shot learning and found that including our predefined examples improved performance across the board. Our prompt design did not exhaust the token limit but rather included relevant examples Poesia et al. [2022].

Codex-Edit. On March 22nd 2022, OpenAI released a new version of Codex designed for the task of editing existing user inputs [Open AI 2022], rather than completing prompts. We use the REST edit API provided by OpenAI. In contrast to the traditional Codex API, the edit API does not take a standard prompt but rather takes an "instruction" parameter, which states (in natural language) what the model should do. We use the phrase "`Fix bugs in the <domain> code`" as the instruction and replace "`<domain>`" with Excel or Power Fx, as appropriate. Additionally, note that this API does not return multiple possible candidates but rather a single result. To mitigate this restriction, we call the edit API 50 times to produce up to 50 candidate repairs for each formula.

For Codex/Codex-Edit, we use temperatures 0, 0.1, 0.3, 0.4, 0.5, and 0.7. We report the best performance for each cutoff in our main evaluation; detailed breakouts in supplementary materials.

5.3 Results

Table 2 shows the number of formulas from our benchmark set that are successfully repaired by each system. In the Excel domain, we see that Excel-Desktop, which only produces a single repair candidate for each faulty formula, performs worst, repairing only 83 of the 200 formulas. The symbolic state-of-the-art grmtools repairs up to 108 formulas, but still lags the neural and neurosymbolic approaches. In particular, grmtools fails to repair many formulas where the edit is far away from the location where the error state is raised. In terms of neural approaches, we find that BiFi, which is trained specifically for our task and domain, can repair substantially more formulas than the symbolic approaches (across all cutoffs). BiFi also outperforms Codex at top-1, but repairs fewer formulas at more lenient cutoffs. Codex-Edit, which is designed for the task of editing user input, improves over both BiFi and Codex, outperforming both approaches over all top-K cutoffs. LaMirage, which combines the advantages of symbolic candidate enumeration with neural error localization and ranking, fixes the most formulas across all systems. LaMirage fixes 27 more formulas at the top-1 cutoff than the next best system, Codex-Edit.

In the Power Fx domain, LaMirage similarly outperforms all baselines across all cutoffs, repairing 64 more formulas at the top-1 cutoff compared to the next best system (Codex-Edit). All systems with data-driven components, in this case neural approaches and LaMirage, experienced a drop in performance compared to the Excel domain, though LaMirage experienced a smaller drop. The purely symbolic approach grmtools repaired a comparable number of formulas in both domains. We believe this is due to the fact that while Power Fx is a growing language, publicly available Power Fx code is significant scarcer than Excel code. This challenge in data availability is reflected in our own training data as well. Neural approaches that had significant pre-training (e.g. Codex and Codex-Edit) experienced a smaller drop than BiFi, which was trained specifically on the Power Fx data we collected. LaMirage, which combines symbolic and neural benefits, experienced the smallest drop in performance (4 fewer formulas repaired at top-1 cutoff) compared to the next best approach (Codex-Edit, which repaired 41 fewer at top-1 cutoff).

In both domains, we notice that Codex and Codex-Edit results improve substantially between top-1 and top-3 cutoffs (while still trailing LaMirage) and stabilize thereafter. We believe these large pretrained models fail to account for differences in low-code formulas, as their training corpus primarily consist of programs written in general purpose languages. Both engines are capable of producing viable candidates, but do not suitably distinguish among them.

We also compute the median time to produce all candidate repairs for each formula in our benchmarks. We exclude Excel-Desktop as error recovery is exposed via a pop-up which requires user interaction, thus invalidating time measurements. Codex and Codex-Edit use the OpenAI REST API, so they include network time, and we compute the minimum time across temperatures for each benchmark before summarizing as a median. For grmtools, we run the tool 50 times on each formula to obtain 50 candidate repairs, as such we add these times up. These repetitions incur repeated processing within grmtools, unavoidable without significant modifications to the tool.

In both domains, the median repair time for LaMirage is substantially lower than for our baseline neural methods. However, the symbolic tool – grmtools – is significantly faster than all the systems. The speed with which grmtools can produce repairs comes at a trade-off: its search space is smaller and as a result it performs less expressive repairs. In particular, grmtools does not backtrack from the location where the error is originally identified. Repairs that require edits to other locations (e.g. earlier in the program) are out of its scope. This design decision reflects the fact that grmtools is designed as a parsing tool, with state-of-the-art parsing recovery, targeting traditional compilation workflows, which require speed.

5.4 Ablation Study Over LaMirage

We now present results (Table 3) that explore the impact of different design choices in LaMirage.

Enumeration ablation. We compare LaMirage to two ablations that replace LaMirage's candidate enumeration with neural enumeration. *CodeBERT* and *Codex* in Table 3 use CodeBERT and Codex, respectively, to generate candidate repairs. To use these neural models for candidate enumeration, when LaMirage reaches an error state, we make the corresponding neural model predict the sequence of unreliable tokens bounded by the reliable tokens at the particular error location (see Section 4.3 for details on this bounding). In the case of Codex, we prompt the model

Table 3. *LaMirage outperforms ablations:* We consider the following ablations: purely neural (CodeBERT, Codex) enumeration, naive symbolic enumeration (Whole-Prefix), no domain specific strategies (DSS), no neural ranker, and no localizer. In all but one case, LaMirage repairs more formulas across all top-K cutoffs. We find that neural ranking (on average) does not play as big a role in Excel as there are fewer candidates with edit-distance ties and so neural ranking tie-breaks are less important.

Ablation	**System**	**Excel**			**PowerFx**		
		Top-1	**Top-3**	**Top-5**	**Top-1**	**Top-3**	**Top-5**
Enumeration	CodeBERT	135	137	137	125	130	131
	Codex	128	128	128	112	121	121
	Whole-Prefix	71	97	110	116	141	150
DSS	No DSS	138	145	145	153	164	165
Neural	No Neural Ranker	173.9	182.2	182.6	150.9	167.8	169
	No Neural Localizer	167	175	175	159	166	166
Full	**LaMirage**	**174**	**182**	**182**	**170**	**177**	**177**

to produce the sequence of unreliable tokens as a completion to a prompt which corresponds to the formula prefix. Zero-shot prediction worked better for this task of generating unreliable token sequences. In the case of CodeBERT, we predict the next unreliable token, given the prefix/suffix of the formula, and we add a beam-search layer to output sequences (up to stop token).

We also compare LaMirage to an ablated version (Whole-Prefix in the table) that uses symbolic backtracking and candidate enumeration but considers the entire formula prefix (up to where the error state was identified) when generating candidate repairs. This is in contrast to LaMirage's approach of identifying edit ranges based on surrounding reliable tokens.

Our results show that LaMirage's enumeration outperforms all ablated versions across all top-K cutoffs in both domains. Ablation with CodeBERT outperformed the version that uses Codex for candidate enumeration in both domains as well. The ablated version with whole-prefix deterministic backtracking solves much fewer benchmarks as the search space explodes leading to time outs.

Domain specific strategies. Next, we evaluate the impact of domain specific strategies (DSS) on LaMirage's performance. Our results show that removing DSS substantially decreases the number of benchmarks solved in the Excel domain across all top-K cutoffs. DSS play a major role in the number of benchmarks solved in Power Fx across all cutoffs, but this benefit is smaller in this domain relative to Excel. DSS plays a bigger role in Excel owing to idiosyncrasies in its formula language that are difficult to capture with just a grammar approximation. For example, Excel's parser does not recognize a function call if there is a space between the function name and the opening parentheses – a DSS, consisting of a single line of code, can resolve this. Relatedly, malformed cell references, another frequent user error, is more easily handled using DSS as compared to a grammar. Additional DSS can be added to LaMirage to further improve coverage of user scenarios.

Neurosymbolic Ranking and Localization. Finally, we compared LaMirage to two ablations that remove the neural error localizer and the neural ranker. Without the ranker, the ablated version cannot distinguish between candidate repairs with the same edit distance. So we run the ablated version without neural ranker 100 times, compute results, and report the average across each cutoff.

In Excel, full LaMirage outperforms the version without neural localizer but is comparable to the version without neural ranker. This behavior is due to the fact that we generate fewer repair candidates with edit-distance ties in Excel, so there is less opportunity to exploit neural ranking for tie breaking. In contrast, both neural ranking and localization play a significant role in Power Fx.

5.5 Discussion

Error Analysis. We consider the formulas in each domain that we cannot repair with LaMirage after considering the top-5 candidates. In Excel, we find that we can increase our coverage to 4 additional formulas if we change LaMirage hyperparameters: a deeper backtracking would cover 3 formulas and a longer repair timeout would cover 1 more formula. Extending our approximate grammar G with new productions would cover 4 additional formulas. Finally, if we add more type constraints to C and token preprocessing DSS we would cover the remaining 9 formulas.

In the case of Power Fx, increasing the set of type constraints and DSS would increase coverage to 14 additional formulas. If we extend our approximate grammar, we would cover 2 of the remaining formulas. For one formula, our neural localizer incorrectly predicts that a reliable token is the source of the error – improving the localizer would resolve this issue. If we perform deeper deterministic backtracking, we can solve 3 more formulas. If we increase the edit distance threshold, we solve one more. The remaining two formulas require inserting reliable tokens.

The solutions proposed here for increasing coverage all come with tradeoffs. By increasing backtracking depth, timeouts, and edit distance thresholds, expanding the approximate grammar, and adding more DSS and constraints, we can increase the number of formulas repaired at the expense of efficiency. LaMirage does not make a decision on these fronts, but rather lets the language developer choose the appropriate tradeoffs for their use case.

Neural vs LaMirage. In our evaluation, LaMirage outperformed neural baselines such as Codex-Edit. Cases where Codex-Edit failed (but LaMirage succeeded) provide interesting insights into challenges of purely neural approaches.

Given the Excel formula `=IF(B6="", "",`, Codex-Edit (even at temperature=0.0) returns a degenerate candidate repair that nests the user's input repeatedly `=IF(B6="","",IF(B6="",...`. The fix simply requires a closing parenthesis. On other occasions, Codex-Edit adds spurious additional code. For example, given the formula `=LEN(MID(A2,1,SEARCH("<",A2))`, Codex-Edit returns `=LEN(MID(A2,1,SEARCH("<",A2)-1))` which changes the computation. This is especially challenging for LC settings, where the user is less likely to spot errors due to their lack of development experience. In other cases, Codex-Edit fails to recognize that there is an error altogether. For example, given `=B2< =EDATE(TODAY(),-33)`, Codex-Edit returns the original input rather than remove the space between the less than and equals operator (caught by LaMirage's constraints C).

Training a neural model for low code language repair can improve performance but may be challenging due to data availability. Additionally, fine-tuning large pretrained models like Codex and Codex-Edit, while appealing, raises resource challenges (e.g. Codex has 12 billion parameters). Alternatively, developers can rely on paid APIs, like OpenAI's, but this can represent substantial costs for high-volume low-code platforms (e.g. OpenAI's completion API costs 6 cents per 1k tokens as of early 2022). These models are also expensive at inference (prediction) time and unlikely to to be deployed in resource-constrained environments for the foreseeable future.

Neural models will continue to improve, but symbolic systems can also be improved through the addition of more domain knowledge. In practice, systems like LaMirage, which combine both approaches, are a particularly interesting point in the design space. LaMirage can quickly (and cheaply, in terms of compute resources) produce effective repairs, but at the one-time cost of the language developer providing an annotated CFG and DSS.

Applicability. We evaluated LaMirage's applicability on two popular low-code languages: Excel and Power Fx. The former counts hundreds of millions of daily users, the latter is the language used in PowerApps, one of Microsoft's fastest growing offerings. While the ideas presented in LaMirage may provide a good starting point for tackling similar last-mile repair tasks in other languages, such as small Python expressions, we leave that for future work and keep our focus on the low-code domain. It is worth highlighting that the low-code domain is *not* restricted to a single language, but rather consists of different platforms with different languages – e.g., Salesforce Lightning, Creatio, Google Sheets, Mendix, and other low-code platforms all have their own languages.

6 INSTANTIATING THE FRAMEWORK

LaMirage is a framework that can be instantiated to create a specific repair engine for a particular language. This instantiation is guided by the language developer, and influences the kind of repairs that will be produced for consumption by the end-user. By design, the framework supports many choices so that a language developer can choose the tradeoffs that are right for their domain/platform. We will discuss the *necessary* components, along with additional extensions.

First, the developer needs to provide LaMirage with an annotated CFG. The CFG annotations correspond to marking terminals as unreliable, and potentially setting different edit costs for different unreliable tokens. If the user does not specify per-token edit costs, we use a default value of 1. Most language developers already have a CFG for their domain, or can craft one that targets the subset of the language they believe may benefit most from repairs. Similarly, in our experience, language developers can quickly identify a set of unreliable tokens for their domain – often those that are associated with punctuation.

After receiving the annotated CFG, LaMirage can produce a repair engine capable of fixing syntax errors (as captured by the CFG) by performing edits on unreliable tokens (as captured by the annotations). For some domains or use-cases, such a repair engine may suffice. To expand the scope of repairs to semantic fixes, such as identifying incorrect function call arity and performing edits to resolve this, we require that the language developer provide domain-specific strategies. There are domain-specific strategies (DSS) that are likely to be useful across domains (such as checking call arity) and need only be instantiated with domain-specific values (such as the function names and associated number of arguments) – this was our experience with both the Excel and Power Fx domains. Importantly, the language developer can add DSS incrementally, focusing on implementing a check/fix strategy for observed user errors. In discussions with partner teams, this data-driven approach to DSS development fits well within their traditional workflow.

If the language developer has access to a corpus of well-formed programs in their domain, they may consider training the neural ranker component, which complements the edit-distance-based ranking that takes place in the purely symbolic approach. To train the ranker successfully, the language developer's corpus should contain at least 10s of thousands of well-formed programs. In our experience, these programs can be scraped from online resources (e.g. help forums or repositories) or production resources (e.g. anonymized product telemetry). We found that a relatively small model performed well for ranking purposes, so training on modest GPUs like a K80 works well.

Furthermore, the language developer can also choose to train the neural localizer, which complements the symbolic backtracking implemented by default. If the language developer has pairs of real buggy programs and their corrected versions, as might be available from product telemetry, this data is well suited for training the pointer network. If the paired corpus is relatively small (fewer than 10s of thousands of pairs), the language developer can produce synthetic pairs by introducing errors (uniformly at random) into their corpus of well-formed programs. Additionally, the language developer can train the pointer network initially on these synthetic pairs and *then* fine-tune on their real paired corpus. Similarly to the ranker, we successfully trained the pointer networks for the Excel and Power Fx domains using modest GPU resources.

Note that without training a neural ranker and localizer, the language developer still has the ability of using LaMirage, as we can default to their symbolic alternatives. While the resulting repair engine will fix less programs, as demonstrated by our ablation studies presented in Section 5.4, the purely symbolic system can more easily be deployed in limited-resource platforms such as the browser. Furthermore, the added value from the neural components may vary across domains – in practice, it may be possible to implement post/pre-processing heuristics that cover enough common cases to avoid the need for the neural components, when they are cumbersome to deploy.

Finally, the language developer has the option of setting different values for three hyperparameters: the backtracking depth, a maximum local cost, and a maximum global cost – all of which come with defaults that work well in our two evaluation domains. The appropriate values for each of these hyperparameters depends on the use case of the language developer, as they tradeoff coverage and efficiency. We expect language developers will have benchmarks, and these can be used to pick appropriate hyperparameter values, as is standard in many AI-based systems.

7 RELATED WORK

Symbolic Approaches for Error Correction Historically, most practical implementations of error correction for mistakes such as syntax errors have relied on greedy/simple approaches such as panic error recovery [Aho et al. 1986] – where the processing system just deletes tokens until it can resume parsing. However, more complete error recovery strategies do exist [Aho and Peterson 1972; Cerecke 2003; Corchuelo et al. 2002; Degano and Priami 1995; Fischer et al. 1979; Kim and Yi 2010; Rajasekaran and Nicolae 2014; Spenke et al. 1984]. The state-of-the-art symbolic approach to error recovery, developed by Diekmann and Tratt [2020] and implemented in grmtools, improves on Corchuelo et al. [2002]'s approach both in completeness (the ability to return complete minimum edit sets) and speed (they optimize their implementation).

Like LaMirage, these approaches enumerate repairs as edit operation sequences that allow parsing to continue whenever an error is encountered. The key difference with LaMirage stems from the increased expressiveness of its repairs. First, LaMirage is syntax-guided but is not limited to syntax repairs. LaMirage allows language developers to add domain-specific strategies, which capture non-context free properties of well-formed programs, to increase the scope of repairs. This allows LaMirage repair engines to produce fixes for some semantic mistakes like incorrect function call arity. Second, LaMirage can address non-local errors – meaning errors that require a repair that is not in the immediate location where the error is detected – by performing backtracking. State-of-the-art symbolic systems such as grmtools do not perform backtracking as it leads to an explosion in the search space. LaMirage mitigates this by combining two ideas. When it backtracks, LaMirage still limits editing actions to unreliable tokens (constraining the space of possible candidates). Additionally, it backtracks up to a fixed depth and relies on a neural localizer to identify candidate locations beyond that depth bound.

As both of these ideas increase the size of LaMirage's search space, the system also needs to be more effective at comparing candidates. Tools such as grmtools rely exclusively on minimum edit distance. In contrast, LaMirage also employs a neural ranker that can break ties between otherwise equidistant candidates.

Neural approaches for Error Correction Program repair systems are increasingly using deep learning to correct syntax and semantic errors in general purposes programming languages [Ahmed et al. 2021; Gupta et al. 2017; Santos et al. 2018; Tufano et al. 2018; Yasunaga and Liang 2020, 2021]. Our evaluation compares to three such methods (BiFi, Codex, Codex-Edit). Other work in the space includes DeepFix [Gupta et al. 2017], which fixes syntactic errors in C programs using a sequence-to-sequence approach; SynFix [Ahmed et al. 2021], which fixes Java syntax errors by applying multiple deep learning models in sequence; and TFix [Berabi et al. 2021], which pretrains a T5 transformer [Raffel et al. 2019] on natural language and fine-tunes it on buggy/repaired code snippets mined from Github commits.

A challenge with purely neural methods is the lack of guarantees over outputs. For example, language models are known to generate plausible but incorrect content, known as "hallucinations" [Guo et al. 2021]. Such models are also known to introduce basic errors [Poesia et al. 2022], which complicates repair. In contrast, LaMirage's candidate repairs are guaranteed to satisfy our approximate grammar G and our constraints C. One way to mitigate this issue with neural models is to increase their training data but availability (at scale) can be challenging in the LC domain.

Repair tools for general purpose languages, such as TFix and SynFix or even non-neural methods such as GetAFix [Bader et al. 2019], typically assume a more detailed oracle that provides detailed diagnostics. This oracle is not often available for LC domains, in part due to lack of tooling.

General Program Repair and Synthesis Monperrus [2020] reviews the vast literature of automated program repair, most of which is focused on repair in the context of general purpose programming languages[Bader et al. 2019; Gao et al. 2021; Goues et al. 2019b; Kim et al. 2013; Long et al. 2017; Long and Rinard 2016; Mechtaev et al. 2016; Nguyen et al. 2013; Weimer et al. 2009] and not low-code domains. In contrast, LaMirage specifically focuses on last-mile repair in low-code domains, where oracles such as test suites or static analyzers are not readily available. The lack of such oracles has influenced the scope and design of LaMirage. Specifically, we focused on repairing errors that require small changes and that can be detected without substantial additional context. Intuitively, these error correspond to those that a typical user might post to a help forum, where an expert low-code user can often provide a fix simply from the posted formula.

As discussed, there are parallels between LaMirage's syntax-guided approach and other syntax-guided program synthesis systems [Devlin et al. 2017; Gulwani 2011; Microsoft PROSE Github 2022; Polozov and Gulwani 2015]. However, there are key differences. First, we only have the buggy formula and no additional specification (e.g., input/output examples). Second, a generic string transformation DSL is unlikely to result in valid repairs. In contrast to synthesis for program transformations [Miltner et al. 2019; Rolim et al. 2017], LaMirage does not mine edit patterns from groups of programs but rather relies on the CFG and DSS to induce transformations of the user's buggy formula. Incorporating mined patterns into LaMirage as DSS is left as future work.

Prior program repair work has explored the use of machine learning models during search. For example, Prophet [Long and Rinard 2016] used a log-linear model to rank candidate patches before validating them with a test suite. In contrast to this work, LaMirage uses two *neural* models during search: a pointer-network-based localizer and a transformer-based ranker. While Prophet computes manually defined features over C patches, LaMirage relies on the ability of neural models to capture high-dimensional patterns without the need for manually defined features. Prophet produces their candidate patches by applying template-based rewrites to the originally buggy program. In contrast, LaMirage performs a syntax-guided search as it processes the input buggy program. Finally, LaMirage performs pruning while searching compared to Prophet, which first generates all patches, ranks them, and then validates them using a test suite.

More broadly, combining machine learning and symbolic methods has been explored by past work both in the areas of program synthesis generally and program repair specifically, for example [Chen et al. 2018; Ellis et al. 2018; Kalyan et al. 2018] and [Tang et al. 2021; Yu et al. 2019; Zhu et al. 2021], respectively. Many of these systems use neural methods to reduce the search time (by constraining or prioritizing search direction) and improving the ranking of competing candidates produced. In this regard, LaMirage takes a similar approach. However, in this work we introduce a novel application of these techniques to the task of producing last-mile repairs that fix syntax and some semantic errors in the low-code domain. Furthermore, we present various insights that show how to effectively combine these techniques for our particular use case. Specifically, we show that we can use relatively small models for both error localization and ranking in the low-code domain.

Both of these can be successfully trained with modest amounts of data – enabled by the smaller model sizes – collected from a combination of help forums and product telemetry. In the case of the neural localizer, we show that using a fallback strategy, where we employ the neural model only when symbolic tracking fails to produce a candidate, lets us exploit the additional coverage of the neural localizer without blowing up the search space. In the case of the neural ranker, we show that applying a standard causal language modeling objective fails to work due to the skew in unreliable token sequences – we introduced instead a balanced training set and a classification task.

Neurosymbolic Methods for Programming Tasks Synchromesh [Poesia et al. 2022] improves the extent to which large language models, such as Codex, can generate syntactically and semantically valid code from natural language utterances. To do so, Synchromesh biases the decoding process towards tokens that would produce valid completions. These tokens are derived from the associated grammar and domain constraints (such as table schemas for SQL). Additionally, Synchromesh introduces target similarity tuning – a method for picking few shots for the prompt that are expected to have a similar program structure to the intended program.

Rahmani et al. [2021] note that LLMs often fail to generate the desired program directly from natural language descriptions, but the programs that they do generate often contain most or all of the components that should appear in the desired program. Thus, their approach consists of generating candidate programs from the natural language utterance using an LLM, and then mining code fragments from these as components, and performing component-based synthesis over these mined components (and a DSL) to satisfy the input/output examples provided. They show that such an approach performs well in the domains of regular expressions and CSS selectors.

Verbruggen et al. [2021] combine LLMs into a traditional inductive synthesizer by identifying subproblems that cannot be solved through syntactic transformations but can be handed off to a LLM to produce a solution. By incorporating an LLM into a symbolic synthesizer, their approach can support semantic transformations of inputs, such as returning the currency symbol given a country name, without the need to manually define such operators.

Similarly to this line of work, LaMirage combines both symbolic and neural methods. In particular, LaMirage uses a neural localizer and ranker, and integrates these into a symbolic (syntax-guided) framework for processing buggy input programs and generating candidate repairs. LaMirage applies these ideas to a new task (last-mile repair in the low-code domain) and arrives at the specific design that works well in practice (as shown by our evaluation).

8 CONCLUSION

We presented LaMirage, a last-mile repair-engine generator for programs written in low-code (LC) formula languages. LaMirage targets “last-mile repairs”, where the formula is almost correct and has a few subtle errors. We designed LaMirage to combine the advantages of symbolic and neural techniques. We evaluated LaMirage on real Excel and Power Fx formulas. Our results showed that LaMirage outperforms state-of-the-art symbolic and neural techniques in both domains. We carried out ablation studies on the design decisions in LaMirage. We discussed the useability of our framework and design considerations, motivated by our ongoing partnership with engineering teams to integrate our repair engines into leading LC platforms.

ACKNOWLEDGMENT

We thank the authors of baselines used in our research. We would like to extend particular gratitude to the grmtools authors for valuable feedback. We thank the Excel and PowerApps teams for their insights and assistance. We also thank Yair Helman and Gio Della-Libera for sponsoring this work for Excel and PowerApps, respectively.

REFERENCES

Toufique Ahmed, Noah Rose Ledesma, and Premkumar T. Devanbu. 2021. SYNFIX: Automatically Fixing Syntax Errors using Compiler Diagnostics. *CoRR* abs/2104.14671 (2021).

Alfred V. Aho and Thomas G. Peterson. 1972. A Minimum Distance Error-Correcting Parser for Context-Free Languages. *SIAM J. Comput.* 1, 4 (1972), 305–312.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., USA.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. *Syntax-guided synthesis.* IEEE.

Appian 2022. Appian. https://appian.com/.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 159:1–159:27.

Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 780–791. https://proceedings.mlr.press/v139/berabi21a.html

Carl Cerecke. 2003. Locally least-cost error repair in LR parsers. (2003).

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021).

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. In *International Conference on Learning Representations.*

Rafael Corchuelo, José A Pérez, Antonio Ruiz, and Miguel Toro. 2002. Repairing syntax errors in LR parsers. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24, 6 (2002), 698–710.

Nelson Cowan. 2001. Metatheory of storage capacity limits. *Behavioral and brain sciences* 24, 1 (2001), 154–176.

Andrew M. Dai and Quoc V. Le. 2015. Semi-supervised Sequence Learning. In *Advances in Neural Information Processing Systems.* 3079–3087.

Pierpaolo Degano and Corrado Priami. 1995. Comparison of syntactic error handling in LR parsers. *Software: Practice and Experience* 25, 6 (1995), 657–679.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning.* PMLR, 990–998.

Lukas Diekmann and Laurence Tratt. 2020. Don't Panic! Better, Fewer, Syntax Errors for LR Parsers. In *34th European Conference on Object-Oriented Programming, ECOOP 2020 (LIPIcs, Vol. 166).* 6:1–6:32.

Ian Drosos, Philip J Guo, and Chris Parnin. 2017. HappyFace: Identifying and predicting frustrating obstacles for learning programming at scale. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* IEEE, 171–179.

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems* 31 (2018).

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP (Findings of ACL, Vol. EMNLP 2020).* 1536–1547.

Charles Fischer, Bernard Dion, and Jon Mauney. 1979. *A locally least-cost LR-error corrector.* Technical Report. University of Wisconsin-Madison, Department of Computer Sciences.

Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. 2021. APIfix: Output-Oriented Program Synthesis for Combating Breaking Changes in Libraries. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 161 (oct 2021), 27 pages. https://doi.org/10.1145/3485538

Google Sheets 2019. Google Sheets. https://www.google.com/sheets/about/.

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

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

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Found. Trends Program. Lang.* 4, 1-2 (2017), 1–119.

Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. 2021. Learning to Complete Code with Sketches. In *International Conference on Learning Representations*.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17)*. 1345–1351.

William T Hallahan, Ennan Zhai, and Ruzica Piskac. 2017. Automated repair by example for firewalls. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 220–229.

Austin Zachary Henley. 2018. *Human-centric Tools for Navigating Code*. The University of Memphis.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186* (2018).

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering, ICSE*. 802–811.

Ik-Soon Kim and Kwangkeun Yi. 2010. LR error repair using the A* algorithm. *Acta Inf.* 47 (2010), 179–207.

Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 727–739.

Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL*. 298–312.

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *International Conference on Software Engineering (ICSE)*. 691–701.

Microsoft Excel 2021. Microsoft Excel. https://www.microsoft.com/en-us/microsoft-365/excel.

Microsoft Power Apps 2019. Microsoft Power Apps. https://powerapps.microsoft.com/en-us/.

Microsoft Power Automate 2019. Microsoft Power Automate. https://flow.microsoft.com/en-us/.

Microsoft Power Fx overview 2022. Microsoft Power Fx overview. https://docs.microsoft.com/en-us/power-platform/power-fx/overview.

Microsoft PROSE Github 2022. Microsoft PROSE Github. https://github.com/microsoft/prose.

Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the Fly Synthesis of Edit Suggestions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 143 (oct 2019), 29 pages. https://doi.org/10.1145/3360569

Martin Monperrus. 2020. *The Living Review on Automated Program Repair*. Technical Report hal-01956501. HAL.

Morgan Stanley 2015. Morgan Stanley Technology, Media & Telecom Conference. https://www.microsoft.com/en-us/investor/events/FY-2015/morgan-stanley-qi-lu.aspx?EventID=156417

MrExcel Message Board 2021. MrExcel Message Board. https://www.mrexcel.com/board/.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *International Conference on Software Engineering (ICSE)*. 772–781.

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

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable Code Generation from Pre-trained Language Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=KmtVD97J43e

Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126.

Power Apps Community 2021. Power Apps Community. https://powerusers.microsoft.com/t5/Power-Apps-Community/ct-p/PowerApps1.

Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. 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, 1066–1082. https://doi.org/10.1145/3385412.3386001

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

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. *CoRR* abs/1406.3405 (2014).

Mohammad Raza and Sumit Gulwani. 2017. Automated data extraction using predictive program synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 31.

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415.

Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 311–322.

Michael Spenke, Heinz Muhlenbein, Monika Mevenkamp, Friedemann Mattern, and Christian Beilken. 1984. A language independent error recovery method for LL(1) parsers. *Software: Practice and Experience* 14, 11 (1984), 1095–1107.

Yu Tang, Long Zhou, Ambrosio Blanco, Shujie Liu, Furu Wei, Ming Zhou, and Muyun Yang. 2021. Grammar-based patches generation for automated program repair. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. 1300–1305.

V Javier Traver. 2010. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction* 2010 (2010).

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *International Conference on Automated Software Engineering, ASE*. 832–837.

UiPath 2019. UiPath. https://www.uipath.com/.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2018. Neural Program Repair by Jointly Learning to Localize and Repair. In *International Conference on Learning Representations*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

VentureBeat 2022. Low-code app development platform Crowdbotics raises $22M. https://venturebeat.com/2022/01/20/low-code-app-development-platform-crowdbotics-raises-22m/.

Gust Verbruggen, Vu Le, and Sumit Gulwani. 2021. Semantic programming by example with pre-trained models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–25.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. *Advances in neural information processing systems* 28 (2015).

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*. 364–374.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. In *International Conference on Machine Learning*, Vol. 119. 10799–10808.

Michihiro Yasunaga and Percy Liang. 2021. Break-It-Fix-It: Unsupervised Learning for Program Repair. In *International Conference on Machine Learning, ICML*, Vol. 139. 11941–11952.

Zhongxing Yu, Matias Martinez, Tegawendé F Bissyandé, and Martin Monperrus. 2019. Learning the relation between code features and code transforms with structured prediction. *arXiv preprint arXiv:1907.09282* (2019).

Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.