Lay-it-out: Interactive Design of Layout-Sensitive Grammars

(Preliminary Draft)

FENGMIN ZHU*†, MPI-SWS, Germany and Tsinghua University, China
JIANGYI LIU*, Tsinghua University, China
FEI HE, Tsinghua University, China

Layout-sensitive grammars have been adopted in many modern programming languages. However, tool support for this kind of grammars still remains limited and immature. In this paper, we present Lay-it-out, an *interactive* framework for layout-sensitive grammar design. Beginning with a user-defined context-free grammar, our framework refines it by synthesizing *layout constraints* through user interaction. For ease of interaction, a *shortest* nonempty ambiguous sentence (if exists) is automatically generated by our *bounded ambiguity checker* via SMT solving. The *soundness* and *completeness* of our SMT encoding are mechanized in the Coq proof assistant. A comprehensive evaluation, including 2 case studies on real-world grammar and a user study on potential users of our tool, demonstrates the effectiveness, scalability, and usability of our approach.

Additional Key Words and Phrases: layout-sensitive grammar, ambiguity, SMT, synthesis, Coq

1 INTRODUCTION

Layout-sensitive (or *indentation-sensitive*) grammars were first proposed by Landin [1966]. Nowadays, they have been adopted in many programming languages, e.g., Python [Van Rossum and Drake 2011], Haskell [Marlow et al. 2010], F# [Syme et al. 2010], Yaml [Evans et al. 2014] and Markdown [Gruber 2012]. These grammars are also introduced as new features in latest versions of languages, e.g., optional braces in Scala 3[1]. Due to the presence of layout constraints, indentations and whitespaces affect how a program should be parsed. Although this amazing feature gives rise to a stylized, structural, and elegant syntax, it *increases the complexity* of theoretical study (e.g., ambiguity problems) and tool development.

In the recent decade, pioneering studies have been made on *layout-sensitive parsing* [Adams 2013; Amorim et al. 2018; Erdweg et al. 2013]. To apply these parsing techniques, one must, first of all, have the grammar formally specified. In a serious language design phase, this grammar should be *unambiguous*. Due to the complexity introduced by layout constraints, *manually* checking the ambiguity (especially for large grammars) is *tedious* and *error-prone*. Therefore, tool support for *automated* ambiguity checking is essential to *reduce human labor* and *avoid human mistakes*.

For *context-free grammars* (CFGs), a common practical approach of ensuring unambiguity is to check if it belongs to an *unambiguous fragment* of CFG, such as $LL(k)$ [Aho et al. 1986] and $LR(k)$ [Knuth 1965]. Off-the-shelf tools such as Lex/Yacc [Levine et al. 1992], Flex/Bison [Levine 2009], and SDF [Visser et al. 1997] have been built to automatically check whether a user-input CFG is $LL(k)$ or $LR(k)$ (and generate the parser as well). For layout-sensitive grammar, however, this approach is not yet practical: as far as we know, there is no similar theory or tool support.

In grammar design, proving unambiguity is an ultimate goal, but it is *hard*: checking the ambiguity of a CFG is already *undecidable* [Chomsky and Schützenberger 1963; Hopcroft et al. 2001], so it is

*Both authors contributed equally to this work.
†Early revisions of this work were done when the author was in Tsinghua University.
[1]https://docs.scala-lang.org/scala3/reference/other-new-features/indentation.html

Authors' addresses: Fengmin Zhu, MPI-SWS, Saarbrücken, Saarland, 66123, Germany, paulzhu@mpi-sws.org, Tsinghua University, Beijing, 100084, China; Jiangyi Liu, Tsinghua University, Beijing, 100084, China, liujiang19@mails.tsinghua.edu.cn; Fei He, Tsinghua University, Beijing, 100084, China, hefei@tsinghua.edu.cn.

even harder for layout-sensitive grammar (more expressive than CFG). Alternatively, we aim to tackle the *bounded ambiguity problem*, which is more restrictive and thus decidable, but yet still *practical*: is there any ambiguous sentence (i.e., has at least two different parse trees) within a given length of a grammar? A sound and complete checker for this problem can *automatically* detect ambiguity during the entire process of grammar design, which helps to *exposes human mistakes* as early as possible.

This paper presents a novel interactive framework, Lay-it-out, for layout-sensitive grammar design. This framework saves time and efforts of grammar designers in two aspects: (1) a *bounded ambiguity checker* automatically generates a *shortest* nonempty ambiguous sentence (if any) so that the grammar designer can detect the cause of ambiguity by a *simplest* and *concrete* example; (2) a *layout constraints synthesizer* recommends candidate *layout constraints* to refine the input-grammar via user interaction based on the ambiguous sentence, so that the user needs *not manually* specify any layout rules. This feature also helps an *ordinary developer* who does not master layout rules to design layout-sensitive grammars for their own domain-specific languages.

In a nutshell, the workflow of Lay-it-out is a grammar *refinement* loop guided by the detection of ambiguous sentences, as depicted in Fig. 1. In the initial round, the user specifies a layout-free grammar (i.e., a CFG) as input. Then, the bounded ambiguity checker produces a shortest nonempty ambiguous sentence (if any) and presents it with all its parse trees to the user. Due to inadequate layout constraints in the input grammar, these parse trees shall be distinguishable once more layout constraints are added. The missing layout constraints are synthesized from the user's *feedback* that reflects their intents: the user is asked to *format* (e.g., insert indentations and newlines between tokens) the ambiguous sentence in distinct ways to match each parse tree. The synthesizer recommends a set of candidate layout constraints: the user can review and select the ones that meet their intents. The user-selected layout constraints will be added to the input grammar. Multiple interaction rounds may be taken until the user is satisfied with the refined grammar.

SMT (*Satisfiability Modulo Theories*) solving is a powerful technique to verification [Beyer et al. 2018; Bjørner and de Moura 2014; Johnson 2009; Leino 2010] and synthesis [Jha et al. 2010; Phothilimthana et al. 2016; Reynolds et al. 2015]. We find it also suitable for building a bounded ambiguity checker. The layout constraints contain relational operations (e.g., $<$, $>$, $=$) on line/column numbers of tokens—they can be easily expressed in *integer difference logic* that most SMT solvers support.

To apply SMT solving, one must encode the problem as an SMT formula. Instead of directly using the standard notion of ambiguity, we propose an alternative definition called *local ambiguity*, which leads to a more *straightforward* and *efficient* encoding. The skeleton of our local ambiguity is inspired by CFGAnalyzer [Axelsson et al. 2008], a tool that studies meta properties of CFGs, including bounded ambiguity, via SAT solving. Our key innovation lies in a *reachability* relation defined for layout-sensitive grammars, taking the influence of layout constraints on ambiguity into careful account. In this way, we can show the *logical equivalence* between local ambiguity and standard ambiguity.

With the equivalence theorem, we are able to construct a *sound* and *complete* SMT encoding for solving the bounded ambiguity problem on *acyclic* layout-sensitive grammars. Being acyclic is *not* a strong assumption: well-designed grammars should all have this nice property. The definitions and theorems related to local ambiguity and encoding are mechanized in the Coq proof assistant. In fact, Coq helped us a lot in finding pitfalls in early revisions of the encoding.

Our layout constraint synthesizer performs analysis on the position information conveyed in user-provided reformatted words and computes a set of candidate layout constraints that *refine* the input grammar. We allow the user to accept a subset of the candidate based on their intents. Note that our synthesizer is targeting the disambiguation caused by inadequate layout constraints,

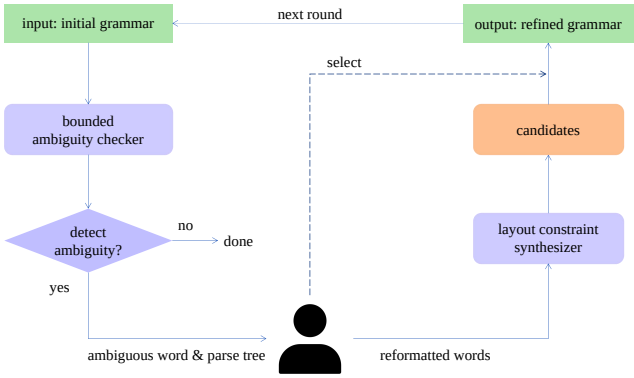

Fig. 1. High-level framework of Lay-it-out.

which is typical and unique for layout-sensitive grammars. Other kinds of ambiguity, such as the well-known binary operator precedence (were studied in CFGs [Klint and Visser 1994; Leung et al. 2015; Thorup 1996]), are not focused on in this work.

To measure the performance of our approach, we conducted a comprehensive evaluation. Two case studies on real languages—YAML's layout-sensitive subset and the *full* SASS grammar—demonstrate that Lay-it-out is effective in grammar disambiguation and scales to full grammar. In particular, we successfully added 74 layout constraints for the SASS grammar within 16 rounds of interaction, among which the longest ambiguous sentence had a length of 12. Besides, a user study conducted on eight recruited participants reveals usability of Lay-it-out. In particular, the user survey shows that all participants prefer our semi-automated approach to the traditional manual approach for layout-sensitive grammar design.

To sum up, this paper makes the following contributions:

- Lay-it-out, the first (to the best of our knowledge) interactive framework for layout-sensitive grammar design (§ 2, § 6, § 8);
- local ambiguity, an equivalent ambiguity definition that is SMT-friendly (§ 4);
- bounded ambiguity checker, developed from a sound and complete SMT encoding for acyclic grammars (§ 5, § 7);
- a comprehensive evaluation that consists of two case studies and one user study, reveals the effectiveness, scalability, and usability of Lay-it-out (§ 9).

2 LAY-IT-OUT BY EXAMPLE

In this section, we illustrate the workflow of Lay-it-out with a toy grammar G_{block}:

$$\begin{aligned} \text{block} &\longrightarrow \text{stmt}^+ \\ \text{stmt} &\longrightarrow \textsf{nop} \mid \textsf{do}\ \text{block} \end{aligned}$$

Imagine you are designing an imperative language where a block is defined as a sequence of statements, each of which can be an empty statement (nop) or a do-block that recursively takes a block as its body. You intend to employ a Haskell-like syntax in this language, and since you have no idea what layout constraints should be added, you specify the above CFG at first, which is apparently ambiguous. Lay-it-out can help you disambiguate this CFG interactively.

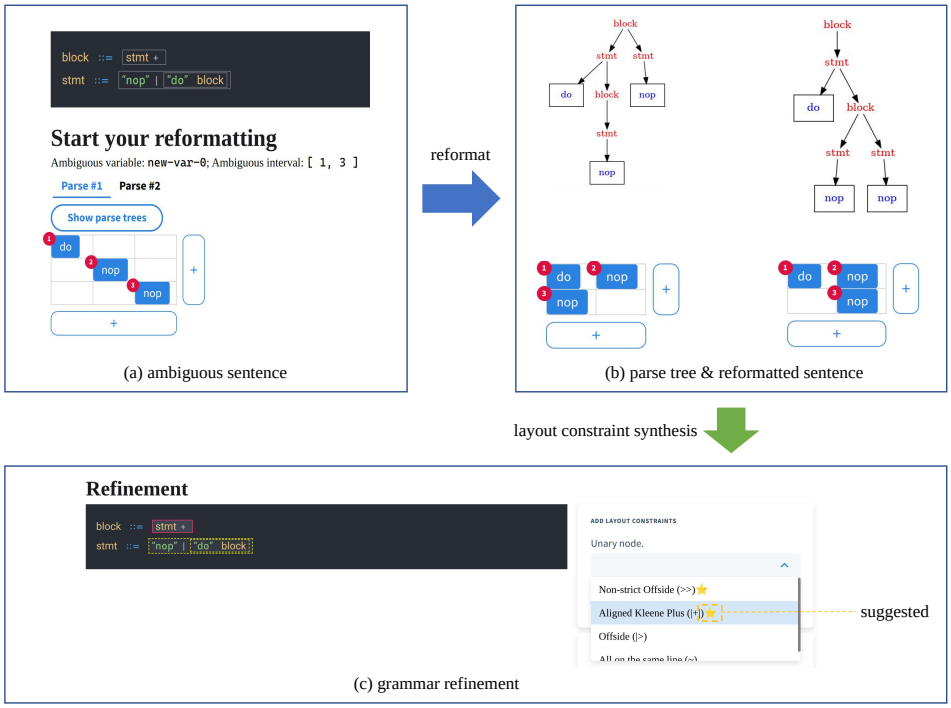

Fig. 2. Apply Lay-it-out to disambiguate G_{block}.

Step 1: Input grammar and check bounded ambiguity. Lay-it-out takes G_{block} as input, preprocesses it, and invokes the bounded ambiguity checker. It produces a *shortest* ambiguous sentence do nop nop, as shown in Fig. 2 (a).

Step 2: View parse trees and reformat sentences. To help with disambiguation, our tool provides you with the parse trees of the given ambiguous sentence. As indicated by Fig. 2 (b), the ambiguous sentence has two possible parses; each is associated with a tab page—"Parse #1" and "Parse #2". By switching between them and clicking "Show parse trees", you can view the two parse trees as depicted at the top of Fig. 2 (b). You realize that both are reasonable parses of do nop nop, and the ambiguity is caused by the uncertainty of which block the last nop statement belongs to: it belongs to the top-level block on #1 (left) and the do-block on #2 (right). A possible way to eliminate this uncertainty is to enforce the statements of one block to align with each other, just like in Haskell. To tell Lay-it-out this intent, you re-layout the tokens by dragging them (located in blue boxes) around in the UI, formatting the ambiguous sentences in two distinct ways that each matches its parse tree, as displayed on the bottom of Fig. 2 (b).

Step 3: Select from suggested layout constraints. Once the reformatted sentences are submitted, Lay-it-out suggests layout constraints from which you choose and add to the grammar. For this example, some layout constraints (marked with yellow stars) are synthesized for clauses "block", "stmt" and "do block". To eliminate the ambiguity, the statements inside one block should be aligned; you thus select the suggested aligned Kleene plus constraint (from the dropdown list displayed on the right in Fig. 2 (c)) and add it to the "stmt$^+$" clause (this clause is selected in the grammar displayed on the left in Fig. 2 (c)). Moreover, since a statement may be multiline, like the

$$\begin{aligned}
\mathsf{align}(w_1, w_2) &\triangleq (w_1 \neq \varepsilon \wedge w_2 \neq \varepsilon) \implies w_1[0].col = w_2[0].col \\
\mathsf{indent}(w_1, w_2) &\triangleq (w_1 \neq \varepsilon \wedge w_2 \neq \varepsilon) \implies (w_2[0].col > w_1[0].col \wedge w_2[0].line = w_1[-1].line + 1) \\
\mathsf{offside}(w) &\triangleq w \neq \varepsilon \implies (\forall t \in w, t.line > w[0].line \implies t.col > w[0].col) \\
\mathsf{single}(w) &\triangleq w \neq \varepsilon \implies \forall t \in w, t.line = w[0].line
\end{aligned}$$

Fig. 3. Logical predicates of built-in layout constraints.

offside rules in Haskell, you need also to add the offside constraint for the subclause "do block"[2]. So far, you obtain a refined layout-sensitive grammar that is unambiguous:

$$\begin{aligned}
\text{block} &\longrightarrow \|\text{stmt}\|^{+} \\
\text{stmt} &\longrightarrow \mathsf{nop} \mid (\mathsf{do}\ \text{block})^{\triangleright}
\end{aligned}$$

where "$\|\cdot\|^{+}$" and "$(\cdot)^{\triangleright}$" denote the aligned Kleene plus and offside constraints, respectively. For this toy example, only one round of interaction is adequate. But in general, multiple rounds of interaction may be needed to disambiguate the grammar eventually.

3 PRELIMINARY

Before diving into details on the technical components of our Lay-it-out framework, this section provides necessary preliminary definitions and notations.

In formal language theory, a grammar G is a quadruple (N, Σ, P, S), where N is a finite (nonempty) set of *nonterminals*, Σ is a finite set of *terminals* (or *tokens*), $P \subseteq N \times (N \cup \Sigma)^*$ is a finite set of *production rules*, and $S \in N$ is the *start symbol*.

In a CFG, a *sentence* (sometimes also called a *word*) is a token sequence. In a *layout-sensitive grammar*, a sentence is a sequence of *positioned tokens*[3]. Each positioned token has the form $a@(line, col)$, where $a \in \Sigma$ gives the terminal, and $(line, col)$ gives the position (i.e. line and column number) in the source file. We denote an empty sentence by ε. For a nonempty sentence w, we denote its length by $|w|$. We write $w[i]$ to access the positioned token at index i, and $w[i].term$, $w[i].line$ and $w[i].col$ resp. the terminal, line number and column number.

Layout constraints. An unary (resp. binary) *layout constraint* φ is a logical predicate over one (resp. two) sentence(s), specifying positional restrictions on tokens of the sentences. As a convention, no constraint can be given to empty sentences: $\varphi(\varepsilon) = \text{true}$ (resp. $\varphi(\varepsilon, \cdot) = \varphi(\cdot, \varepsilon) = \text{true}$ for binary case).

We support the following built-in layout constraints: alignment (binary), indentation (binary), offside (unary), offside-align (a variant of offside, unary), and single-line (unary). The first three are widely used in many practical grammars and have been well-studied in previous work [Amorim et al. 2018]. Offside-align and single-line rules are what we find useful in case studies (§ 9). Their logical predicates are presented in Fig. 3, where $w[-1]$ denotes the last positioned token of w.

Alignment restricts that the first token of the two sentences should be aligned at the column. Indentation restricts the second part has its first token to the right (at column) of the first part, and a newline exists in between. The offside rule was first proposed by Landin [1966] and later revised by Adams [2013]. It restricts that in the parsed sentence, any subsequent lines must start

[2]It is fine if you forget this constraint in the first round. In that case, in the next round, Lay-it-out will generate an ambiguous sentence witnessing this ambiguity which can be eliminated via this layout constraint.

[3]We assume all sentences are well-formed: positions of tokens must be in ascending order. Ill-formed sentences do not physically exist in reality, e.g. $[a@(1,2), b@(1,1)]$ (b comes before a) and $[a@(1,1), b@(1,1)]$ (a and b overlap).

from a column that is further to the right of the start token of the first line. We also consider the offside-align rule, a variant of the above, where subsequent lines can start from the same column as that of the first line. The single-line rule restricts the parsed sentence to be one-line.

Binary normal form. It is well-known [Lange and Leiß 2009] that every CFG can be converted to an equivalent *binary normal form* (or *Chomsky normal form*). We migrate the binary normal form to layout-sensitive grammars as follows.

Definition 3.1 (LS2NF). A layout-sensitive grammar is in *binary normal form*, called LS2NF, if for every production rule, its right-hand side (called a *clause*) is one of the following:

- an empty clause ε;
- an atomic clause a, where $a \in \Sigma$;
- an unary clause A^{φ}, where $A \in N$ and φ is a *unary layout constraint*;
- a binary clause $A\ \varphi\ B$, where $A, B \in N$ and φ is a *binary layout constraint*.

Our definition is the same with CFG, but with layout constraints in unary/binary clauses. To avoid inconsistent layout constraints, the production rule set cannot contain the following rules simultaneously: (1) $A \longrightarrow B^{\varphi}$ and $A \longrightarrow B^{\varphi'}$ where $\varphi \neq \varphi'$; (2) $A \longrightarrow B_1\ \varphi\ B_2$ and $A \longrightarrow B_1\ \varphi'\ B_2$ where $\varphi \neq \varphi'$.

Throughout the paper, we study layout-sensitive grammars in LS2NF, and use *Extended Backus-Naur form* (EBNF) to express complex grammars for readability: they are converted to equivalent LS2NF in the standard way [Lange and Leiß 2009]. We also use notations in place of rule names for built-in layout constraints: we write $\alpha \parallel \beta$ for α align β, $\alpha \boxplus \beta$ for α indent β, $\alpha^{\triangleright}$ for $\alpha^{\mathsf{offside}}$, $\alpha^{\trianglerighteq}$ for $\alpha^{\mathsf{offside-align}}$ and $\langle\!|\alpha|\!\rangle$ for α^{single}. The alignment constraint is extended to lists in EBNF: $\|\alpha\|^{+}$ and $\|\alpha\|^{*}$ denote all sentences parsed by α are aligned.

Parse trees. *Parse trees* are rooted trees (let $A \in N$ be the root) inductively defined as follows:

$$t ::= \mathsf{empty}(A) \mid \mathsf{leaf}(A, a@(i,j)) \mid \mathsf{unary}(A, t) \mid \mathsf{binary}(A, t_1, t_2)$$

They resp. correspond to the derivation using the empty, atom, unary and binary clause. We write $\mathsf{root}(t)$ and $\mathsf{word}(t)$ to denote the root node and the represented sentence of a parse tree t. A parse tree is said *valid* if it follows the production rules and fulfills the layout-constraints:

- $\mathsf{empty}(A)$ is valid if $A \longrightarrow \varepsilon \in P$;
- $\mathsf{leaf}(A, a@(i,j))$ is valid if $A \longrightarrow a \in P$;
- $\mathsf{unary}(A, t)$ is valid if $A \longrightarrow \mathsf{root}(t)^{\varphi} \in P$, $\varphi(\mathsf{word}(t))$ is true and t is valid;
- $\mathsf{binary}(A, t_1, t_2)$ is valid if $A \longrightarrow \mathsf{root}(t_1)\ \varphi\ \mathsf{root}(t_2) \in P$, $\varphi(\mathsf{word}(t_1), \mathsf{word}(t_2))$ is true and both t_1, t_2 are valid.

We say a parse tree t *witnesses* a derivation from a nonterminal A to a sentence w, written $t \triangleright A \Rightarrow_* w$, if t is valid, $\mathsf{root}(t) = A$ and $\mathsf{word}(t) = w$. Using the above notions, the usual notion of derivation $A \Rightarrow_* w$ is actually a short-hand for "$\exists t, t \triangleright A \Rightarrow_* w$". Specially, we use the predicate $\mathsf{null}(A)$ to indicate that $A \Rightarrow_* \varepsilon$, a.k.a. A is *nullable*.

Derivation ambiguity. We say the derivation $A \Rightarrow_* w$ is *ambiguous* if there exist at least two different parse trees that witness $A \Rightarrow_* w$. In this way, the bounded ambiguity problem is restated as: for a given bound k, is there a w s.t. $|w| \leq k$ and $S \Rightarrow_* w$ is ambiguous?

Throughout the paper, we reserve A, B, C for nonterminals, a, b, c for tokens/terminals, w for sentences, t, T for parse trees, and φ for both unary and binary layout constraints (inferred from context).

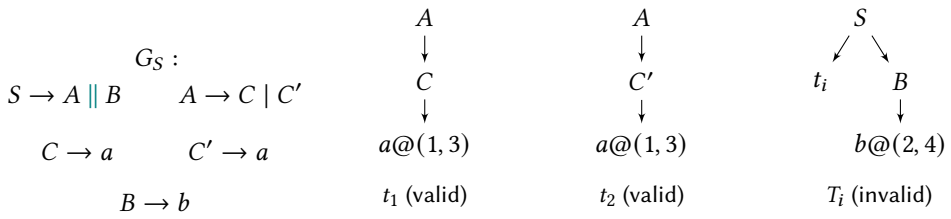

Fig. 4. A counterexample where bAMB does not apply to G_S (where S is the start symbol).

4 LOCAL AMBIGUITY

Lay-it-out is guided by ambiguous sentences generated via constraint solving. The key technical problem is to find an SMT formula $\Phi(k)$ such that it is *satisfiable* iff there exists an ambiguous sentence of length k, and that every satisfying model corresponds to such an ambiguous sentence. The most direct approach is to encode derivation ambiguity—"there exists two *different* parse trees that both witness the derivation"—but how to encode the existential of those trees is *challenging*. Generally speaking, the node pair that shows the difference can appear at an *arbitrary level* (or depth). Apparently, enumerating every possibility is *sophisticated* and *inefficient*.

An *indirect* but *more efficient* approach could be, use an *alternative definition* that is logically equivalent to derivation ambiguity, while being *conveniently encodable* in SMT theories. In this section, we will propose a definition called *local ambiguity*. This definition will eventually allow us to construct a *sound* and *complete* SMT encoding (§ 5) for the bounded ambiguity checking problem.

4.1 Motivation: Lessons from a Failure Attempt

Our first attempt is to apply *bAMB*[4], an existing condition for checking the bounded ambiguity of *reduced*[5] CFGs via SAT solving [Axelsson et al. 2008], in the setting of layout-sensitive grammars. The bAMB condition is stated as: "there exists a nonterminal A and a sentence w such that w has at least two different (valid) parse trees rooted at A and differ in a node on level 1". Since parse trees are visual representations of derivations, such two trees correspond to two distinct ways of deriving w: using either two distinct production rules for A, or one rule in two distinct ways. In terms of logical constraints, bAMB is *weaker* than derivation ambiguity: (1) A is *existentially* quantified and may not be the start symbol; (2) the two parse trees differ in a node just below (i.e., a child of) their root A, which is *determined* and *shallow*. In this way, bAMB is conveniently encodable in SMT theories.

In order to use bAMB as an alternative definition, we must also show that "bAMB $\Leftrightarrow$ derivation ambiguity". The $\Leftarrow$-part is trivial as we have seen that bAMB is weaker. The $\Rightarrow$-part, however, does *not* always hold, as demonstrated by Fig. 4: According to the production rules of G_S, both t_1 and t_2 witness $A \Rightarrow_* [a@(1,3)]$. They differ in a node on level 1 ($C \neq C'$), which matches the definition of bAMB. Since A is not the start symbol, t_1 and t_2 alone cannot witness the ambiguity of G_S. To do so, we have to expand t_1 and t_2 each to a valid parse tree rooted at S, say T_1 and T_2 ($T_1 \neq T_2$ because $t_1 \neq t_2$). Unfortunately, this is impossible: the only possible trees having t_i ($i = 1, 2$) as a subtree are T_i ($i = 1, 2$) according to the production rules, but they are both *invalid* because the *alignment constraint* $A \parallel B$ is *not fulfilled*, i.e., $a@(1,3)$ and $b@(2,4)$ are at distinct columns ($3 \neq 4$).

[4]In *their* paper, "bAMB" stands for "bounded ambiguity". We stick to the abbreviation "bAMB" here because the term "bounded ambiguity" has a different meaning in *this* paper.

[5]A CFG is said reduced if all nonterminal symbols are reachable from the start symbol.

We learn from the above example that the presence of layout constraints can affect parse tree validity in a *non-obvious* way. In Fig. 4, the invalidity of T_i ($i = 1, 2$) is caused by the nonfulfillment of the alignment constraint, which further prevents us from expanding t_1 and t_2 into two distinct valid parse trees rooted at the start symbol. This never happens to CFGs: we can always compose two valid parse trees t_l and t_r into a parse tree $\mathsf{binary}(A, t_l, t_r)$ that is valid as well, as long as $A \longrightarrow \mathrm{root}(t_l)\ \mathrm{root}(t_r) \in P$.

Based on this observation, we find the following property is crucial to make the $\Rightarrow$-part provable: any valid parse tree t shall be expanded into a larger valid parse tree T having t as its subtree. Formally, we define a relation over *signatures*, each is a pair of nonterminal and sentence:

Definition 4.1. (Sub-derivation) Let (A, w) and (B, v) be two signatures. We say (B, v) is a *sub-derivation* of (A, w), if for every parse tree $t \triangleright B \Rightarrow_* v$, there exists a parse tree T such that $T \triangleright A \Rightarrow_* w$ and t is a subtree of T.

With this notion, the idea of proving the $\Rightarrow$-part is as follows: Given $t_1 \neq t_2 \triangleright A \Rightarrow_* w_A$, if some premise indicates that (A, w_A) is a sub-derivation of (S, w_S) for some w_S, then we can conclude that $S \Rightarrow_* w_S$ is ambiguous by expanding t_1 (resp. t_2) into T_1 (resp. T_2), where $T_1 \neq T_2 \triangleright S \Rightarrow_* w_S$. To execute this idea, we first propose *reachability* as the missing premise that implies sub-derivation (§ 4.2), and then define *local ambiguity* to be essentially "bAMB $\wedge$ reachability", so that "local ambiguity $\Leftrightarrow$ derivation ambiguity" becomes provable (§ 4.3).

4.2 Reachability: The Missing Piece

The definition of reachability should encode necessary conditions that establish a sub-derivation relation. To make (B, v) a sub-derivation of (A, w), by definition we must prove $A \Rightarrow_* w$, which can be split into two big steps: $A \Rightarrow_* s_1 B s_2 \Rightarrow_* w_1 v w_2 = w$. Given that $B \Rightarrow_* v$, it suffices to find sentential forms $s_1 \Rightarrow_* w_1$ and $s_2 \Rightarrow_* w_2$ such that $A \Rightarrow_* s_1 B s_2$. Recall that in CFGs, $A \Rightarrow_* s_1 B s_2$ means "B is reachable from A". This motivates us to extend the reachability relation on CFGs to layout-sensitive grammars—taking sentences and the layout constraints they should fulfill into account.

Definition 4.2. (Reachability) The *reachability* relation $(A, w) \rightarrow_* (B, v)$ is the reflexive transitive closure of a one-step reachability relation $\rightarrow_1$ inductively defined as follows:

(1) If $A \longrightarrow B^{\varphi} \in P$ and $\varphi(w)$, then $(A, w) \rightarrow_1 (B, w)$.
(2) If $A \longrightarrow B\ \varphi\ B' \in P$, $\varphi(w_1, w_2)$ and $B' \Rightarrow_* w_2$, then $(A, w_1 w_2) \rightarrow_1 (B, w_1)$.
(3) If $A \longrightarrow B'\ \varphi\ B \in P$, $\varphi(w_1, w_2)$ and $B' \Rightarrow_* w_1$, then $(A, w_1 w_2) \rightarrow_1 (B, w_2)$.

The three rules for $\rightarrow_1$ enumerate all possibilities we can derive a nonterminal B from A in one derivation step, via the production rule $A \longrightarrow B^{\varphi}$, $A \longrightarrow B\ \varphi\ B'$ and $A \longrightarrow B'\ \varphi\ B$ respectively. This definition is expressive enough to indicate a sub-derivation relation, as stated in the following lemma (All lemmas/theorems in this and the next section are mechanized in Coq; we only provide proof sketches in the paper):

Lemma 4.3. *If $B \Rightarrow_* v$, then $(A, w) \rightarrow_* (B, v)$ iff (B, v) is a sub-derivation of (A, w).*

Proof Sketch. For the $\Rightarrow$-part: induction on the reachable relation (in this part, the hypothesis $B \Rightarrow_* v$ is useless). For the $\Leftarrow$-part: we have $T \triangleright A \Rightarrow_* w$ for every $t \triangleright B \Rightarrow_* v$; induction on T and check in every case $(A, w) \rightarrow_* (B, v)$. □

4.3 Local Ambiguity: Foundation of SMT Encoding

With the reachability relation defined, *local ambiguity* is essentially "bAMB $\wedge$ reachability". In bAMB, "two parse trees *differ on a node at level 1*" is formally expressed by "they are *dissimilar*"

(denoted by $\not\simeq$), via a *similarity* relation $\simeq$ inductively defined by the rules below:

$$\mathsf{empty}(A) \simeq \mathsf{empty}(A) \qquad \mathsf{leaf}(A, tk) \simeq \mathsf{leaf}(A, tk)$$

$$\frac{\mathsf{root}(t_1) = \mathsf{root}(t_2) \quad \mathsf{word}(t_1) = \mathsf{word}(t_2)}{\mathsf{unary}(A, t_1) \simeq \mathsf{unary}(A, t_2)}$$

$$\frac{\mathsf{root}(t_{11}) = \mathsf{root}(t_{21}) \quad \mathsf{word}(t_{11}) = \mathsf{word}(t_{21}) \quad \mathsf{root}(t_{12}) = \mathsf{root}(t_{22}) \quad \mathsf{word}(t_{12}) = \mathsf{word}(t_{22})}{\mathsf{binary}(A, t_{11}, t_{12}) \simeq \mathsf{binary}(A, t_{21}, t_{22})}$$

Definition 4.4 (Local ambiguity). A signature (A, w) is said *local ambiguous* if there exists (H, h) such that $(A, w) \rightarrow_* (H, h)$, and there exist $t_1 \not\simeq t_2$ that both witness $H \Rightarrow_* h$.

Local ambiguity is conveniently encodable in SMT theories: both the similarity and reachability relations are inductively defined, and all their side premises are encodable in SMT theories. Our encoding (which will be explained in § 5) is essentially translating the local ambiguity definition into an SMT formula. The following theorem—"local ambiguity ⇔ derivation ambiguity"—provides guarantees towards a sound and complete SMT encoding:

Theorem 4.5. *(A, w) is local ambiguous iff the derivation $A \Rightarrow_* w$ is ambiguous.*

Proof Sketch. For the $\Rightarrow$-part: Let (H, h) be a signature such that $(A, w) \rightarrow_* (H, h)$. Let $t_1 \not\simeq t_2$ be two parse trees that both witness $H \Rightarrow_* h$. By Lemma 4.3, there exist $t \triangleright A \Rightarrow_* w$ and t_1 being a subtree of t. By substituting t_2 for t_1 in t, we obtain a new parse tree $t' \triangleright A \Rightarrow_* w$. We have $t_1 \not\simeq t_2 \Rightarrow t_1 \neq t_2 \Rightarrow t \neq t'$. Therefore, t and t' witness the ambiguity of $A \Rightarrow_* w$.

For the $\Leftarrow$-part: Let $t_1 \neq t_2 \triangleright A \Rightarrow_* w$. It suffices to extract two subtrees, say t_1' from t_1 and t_2' from t_2, such that they have the same signature, say (H, h), and that $t_1' \not\simeq t_2'$. The subtrees can be found via tree difference: compare node pairs between t_1 and t_2 in a depth-first order and return immediately when the nodes are different. □

By this theorem, we argue that G_S shown in Fig. 4 is ambiguous, because (S, w_{ab}) where $w_{ab} = [a@(1,3), b@(2,3)]$ is local ambiguous: (1) $(S, w_{ab}) \rightarrow_* (A, [a@(1,3)])$ (note that the alignment constraint $A \parallel B$ is now fulfilled), (2) $t_1 \not\simeq t_2$ since $C \neq C'$, and (3) both t_1 and t_2 witness $A \Rightarrow_* [a@(1,3)]$.

Comparison. Our local ambiguity strengthens bAMB in two dimensions. First, a reachability relation is defined for layout-sensitive grammars as a nontrivial extension of the usual reachability notion on CFGs. This relation "locally" encodes the necessary conditions for ensuring the equivalence theorem on each signature. Second, our equivalence theorem (Theorem 4.5) is held for any layout-sensitive grammar and obviously also for any CFG, which makes it more general than bAMB that applies to reduced CFGs merely.

5 BOUNDED AMBIGUITY CHECKING

With the local ambiguity defined in the previous section, we are now ready to construct encoding for $\Phi(k)$—the existential of a k-length ambiguous sentence—by translating local ambiguity into an SMT formula. We will present the technical details of this translation in a top-down manner (§ 5.1, § 5.2) and show that it is sound and complete (§ 5.3). Relying on an SMT solver (e.g., Z3) as the backend, our bounded ambiguity checker is facilitated by a bounded loop that finds the smallest $k > 0$ such that $\Phi(k)$ is satisfiable, whose satisfying model gives a shortest ambiguous sentence of the input grammar (§ 5.4).

5.1 Satisfying Model

Before presenting $\Phi(k)$, let us see which variables should be included in a satisfying model m of this formula. Above all, we encode the ambiguous sentence w^m that m represents—a k-length positioned token sequence—by three groups of variables $\mathcal{T}_i$, $\mathcal{L}_i$ and $\mathcal{C}_i$ (for $0 \le i < k$): they resp. encode the terminal, the line number, and the column number of each positioned token in the sequence.

Besides, we introduce auxiliary propositional variables to state whether a derivation or reachability judgment holds, as required by the definition of local ambiguity. The variables are split into three groups (where $w^m_{x,\delta}$ denote the δ-length subword of w^m starting at index x):

(1) $\mathcal{D}^A_{x,\delta}$ for $A \in N, 0 < \delta < k - x$ states whether $A \Rightarrow_* w^m_{x,\delta}$;
(2) $\mathcal{R}^A_\varepsilon$ for $A \in N$ states whether $(S, w^m) \to_* (A, \varepsilon)$;
(3) $\mathcal{R}^A_{x,\delta}$ for $A \in N, 0 < \delta < k - x$ states whether $(S, w^m) \to_* (A, w^m_{x,\delta})$.

We use two groups of variables to encode the reachability judgments, $\mathcal{R}^A_\varepsilon$ for the empty subword and $\mathcal{R}^A_{x,\delta}$ for nonempty subword $w^m_{x,\delta}$. For the derivation judgments, however, we do not need variables $\mathcal{D}^A_\varepsilon$ to encode $A \Rightarrow_* \varepsilon$ because nullability does not depend on a sentence—it can be precomputed from the grammar. Thus, we can use the precomputed results in our encoding: we write $\mathsf{null}(A)$ if A is nullable.

5.2 SMT Encoding

We now present the top-level encoding for $\Phi(k)$, where the auxiliary definitions $\Phi_D(k)$, $\Phi^\varepsilon_R(k)$, $\Phi^\not\varepsilon_R(k)$ and $\Phi_{\text{multi}}(H, x, \delta)$ will be introduced later:

$$\Phi(k) := \Phi_D(k) \wedge \Phi^\varepsilon_R(k) \wedge \Phi^\not\varepsilon_R(k) \wedge \bigvee_{H \in N} \left((\mathcal{R}^H_\varepsilon \wedge \Phi_{\text{multi}}(H, 0, 0)) \vee \bigvee_{0 < \delta \le k - x} (\mathcal{R}^H_{x,\delta} \wedge \Phi_{\text{multi}}(H, x, \delta)) \right).$$

The first three conjuncts $\Phi_D(k)$, $\Phi^\varepsilon_R(k)$ and $\Phi^\not\varepsilon_R(k)$ resp. provide logical restrictions on the three groups of propositional variables $\mathcal{D}^A_{x,\delta}$, $\mathcal{R}^A_\varepsilon$ and $\mathcal{R}^A_{x,\delta}$, so that when $\Phi(k)$ is satisfiable, their truth values will indicate the validity of the derivation/reachability judgments as mentioned in § 5.1.

The last conjunct encodes local ambiguity by Definition 4.4: there exists a nonterminal H and a subword $w^m_{x,\delta}$ such that (1) $(S, w^m) \to_* (H, w^m_{x,\delta})$, expressed by $\mathcal{R}^H_\varepsilon$ (when $w^m_{x,\delta} = \varepsilon$) or $\mathcal{R}^H_{x,\delta}$ (when $w^m_{x,\delta} \ne \varepsilon$), and (2) there are two dissimilar parse trees that witness $H \Rightarrow_* w^m_{x,\delta}$, expressed by $\Phi_{\text{multi}}(H, x, \delta)$.

Well-foundedness. Realizing that derivation and reachability relations are recursively defined on nonterminals, the nonterminal set N should be *well-founded* so that *sound* encodings for $\Phi_D(k)$, $\Phi^\varepsilon_R(k)$, $\Phi^\not\varepsilon_R(k)$ and $\Phi_{\text{multi}}(H, x, \delta)$ can be constructed. To obtain a well-founded relation on N, we require the grammar to be *acyclic*, which intuitively means any cyclic derivation such as $A \Rightarrow_+ A$ is not allowed. This requirement does *not* weaken the practicality of our approach: a well-designed grammar should never be cyclic.

Formally, the *graph representation* of an LS2NF (N, Σ, P, S) is a directed graph $\langle N, E \rangle$ where $(A, B) \in E$ if

$$(A \to B^\varphi \in P) \vee (A \to B\, \varphi\, B' \in P \wedge \mathsf{null}(B')) \vee (A \to B'\, \varphi\, B \in P \wedge \mathsf{null}(B')).$$

Then, an LS2NF is said *acyclic* if its graph representation is acyclic (in graph theory). It is well-known that deciding the acyclicity of a directed graph is solvable in linear-time [Tarjan 1972].

In an acyclic LS2NF, the edge set E of its graph representation forms a well-founded relation, called the *predecessor* relation, written $A \prec B$ for $(A, B) \in E$. The inverse (or transpose) relation E^{-1} is also well-founded, called the *successor* relation, written $A \succ B$ for $(B, A) \in E$.

Encoding derivation. The key observation to encode a derivation judgment $A \Rightarrow_* w$ $(w \neq \varepsilon)$ is that, it can be rewritten (being logically equivalent) in a "more verbose" disjunctive form that is closer to an SMT formula:

$$\begin{aligned}(A \Rightarrow_* w) \Leftrightarrow \bigvee_{A \to a \in P} (|w| = 1 \wedge w_i.term = a) \vee \bigvee_{A \to B^{\varphi} \in P} (B \Rightarrow_* w \wedge \varphi(w)) \\ \vee \bigvee_{\substack{A \to B_1 \varphi B_2 \in P \\ w = w_1 w_2}} (B_1 \Rightarrow_* w_1 \wedge B_2 \Rightarrow_* w_2 \wedge \varphi(w_1, w_2))\end{aligned} \tag{1}$$

where the three disjuncts enumerate all possible ways to achieve $A \Rightarrow_* w$ by production rule $A \to \alpha \in P$ where α can be an atomic, a unary or a binary clause. Formula $\Phi_D(k)$ provides restrictions that every $\mathcal{D}^A_{x,\delta}$ is true iff $A \Rightarrow_* w^m_{x,\delta}$, and the latter is a direct translation of Eq. (1):

$$\begin{aligned}&\Phi_D(k) := \forall A \in N, 0 < \delta \leq k - x : \mathcal{D}^A_{x,\delta} \Leftrightarrow \\ &\quad \bigvee_{A \to a \in P} (\delta = 1 \wedge \mathcal{T}_x = a) \vee \bigvee_{A \to B^{\varphi} \in P} (\mathcal{D}^B_{x,\delta} \wedge \Phi_{\varphi}(x, \delta)) \vee \bigvee_{A \to B_1 \varphi B_2 \in P} \\ &\quad \left((\mathsf{null}(B_1) \wedge \mathcal{D}^{B_2}_{x,\delta}) \vee (\mathsf{null}(B_2) \wedge \mathcal{D}^{B_1}_{x,\delta}) \vee \bigvee_{0 < \delta' < \delta} (\mathcal{D}^{B_1}_{x,\delta'} \wedge \mathcal{D}^{B_2}_{x+\delta', \delta-\delta'} \wedge \Phi_{\varphi}(x, \delta', \delta)) \right)\end{aligned}$$

In this formula, the three disjuncts on the rhs of $\Leftrightarrow$ respectively correspond to the three disjuncts in Eq. (1). The last disjunct of Eq. (1) is split into three cases, depending on if w_1 or w_2 is empty. This is necessary because $\mathcal{D}^{B_1}_{x,\delta'}$ is only defined for nonempty subwords ($\delta' > 0$).

Moreover, for every possible layout constraint φ used in the grammar, we assume there is an SMT formula that *consistently* encodes its semantics: (1) for unary constraint φ, $\Phi_{\varphi}(x, \delta)$ holds iff $\varphi(w^m_{x,\delta})$; (2) for binary constraint φ, $\Phi_{\varphi}(x, \delta', \delta)$ holds iff $\varphi(w^m_{x,\delta'}, w^m_{x+\delta', \delta-\delta'})$. The encodings for the built-in layout constraints are trivially obtained via a direct translation of their definitions (Fig. 3).

Lemma 5.1. *If $m \models \Phi_D(k)$, then for every $A \in N$, $0 < \delta < k - x$, $m(\mathcal{D}^A_{x,\delta}) = true$ iff $A \Rightarrow_* w^m_{x,\delta}$.*

Proof Sketch. By well-founded induction on δ and $\prec$. □

Encoding reachability. The idea of rewriting a derivation judgment as a disjunctive form applies to a reachability judgment too:

$$\begin{aligned}(S, w) \to_* (B, w_B) \Leftrightarrow (B = S \wedge w_B = w) \vee \bigvee_{A \to B^{\varphi} \in P} ((S, w) \to_* (A, w_B) \wedge \varphi(w_B)) \\ \vee \bigvee_{A \to B \varphi B' \in P} ((S, w) \to_* (A, w_B w') \wedge B' \Rightarrow_* w' \wedge \varphi(w_B, w')) \\ \vee \bigvee_{A \to B' \varphi B \in P} ((S, w) \to_* (A, w' w_B) \wedge B' \Rightarrow_* w' \wedge \varphi(w', w_B))\end{aligned} \tag{2}$$

It encodes $\to_*$, the reflexive and transitive closure of $\to_1$, by explicitly stating reflexivity as the first disjunct, and then integrating transitivity into $\to_1$, yielding the last three disjuncts that respectively correspond to the three rules for $\to_1$ as defined in Definition 4.2. Depending on whether w_B is

empty, we obtain the encodings of $\Phi_R^{\varepsilon}(k)$ and $\Phi_R^{\not\varepsilon}(k)$ via a translation of Eq. (2):

$$\Phi_R^{\varepsilon}(k) := \forall B \in N : \mathcal{R}_{\varepsilon}^{B} \Leftrightarrow$$
$$(B = S \wedge k = 0) \vee \bigvee_{A \to B\varphi \in P} \mathcal{R}_{\varepsilon}^{A} \vee \bigvee_{\substack{A \to B\varphi B' \in P \\ or A \to B'\varphi B \in P}} \left((\mathcal{R}_{\varepsilon}^{A} \wedge \mathsf{null}(B')) \vee \bigvee_{0 < \delta \le k - x} (\mathcal{R}_{x,\delta}^{A} \wedge \mathcal{D}_{x,\delta}^{B'}) \right)$$
$$\Phi_R^{\not\varepsilon}(k) := \forall B \in N, 0 < \delta \le k - x : \mathcal{R}_{x,\delta}^{B} \Leftrightarrow$$
$$(B = S \wedge x = 0 \wedge \delta = k) \vee \bigvee_{A \to B\varphi \in P} (\mathcal{R}_{x,\delta}^{A} \wedge \Phi_{\varphi}(x, \delta))$$
$$\vee \bigvee_{A \to B\varphi B' \in P} (\mathcal{R}_{x,\delta+\delta'}^{A} \wedge \mathit{ite}(\delta' = 0, \mathsf{null}(B'), \mathcal{D}_{x+\delta,\delta'}^{B'}) \wedge \Phi_{\varphi}(x, \delta, \delta'))$$
$$\vee \bigvee_{A \to B'\varphi B \in P} (\mathcal{R}_{x-\delta',\delta'+\delta}^{A} \wedge \mathit{ite}(\delta' = 0, \mathsf{null}(B'), \mathcal{D}_{x-\delta',\delta'}^{B'}) \wedge \Phi_{\varphi}(x - \delta', \delta', \delta))$$

The layout predicates $\varphi(w_B)$, $\varphi(w_B, w')$ and $\varphi(w', w_B)$ of Eq. (2) are not translated in $\Phi_R^{\varepsilon}(k)$ because they trivially hold ($w_B = \varepsilon$). When translating $B' \Rightarrow_* w'$ of Eq. (2), we must be careful that w' might be empty: In $\Phi_R^{\varepsilon}(k)$, w' is always empty so the translation is $\mathsf{null}(B')$. In $\Phi_R^{\not\varepsilon}(k)$, we use the "if-then-else" predicate $\mathit{ite}(c, \Phi_1, \Phi_2)$, a shorthand for $(c \wedge \Phi_1) \vee (\neg c \wedge \Phi_2)$, to cover both cases.

Lemma 5.2. *If $m \models \Phi_D(k) \wedge \Phi_R^{\varepsilon}(k)$, then for every $B \in N$, $m(\mathcal{R}_{\varepsilon}^{B}) = true$ iff $(S, w^m) \to_* (B, \varepsilon)$.*

Proof Sketch. By well-founded induction on $\succ$ (reverse of $\prec$). Apply Lemma 5.1 where necessary. □

Lemma 5.3. *If $m \models \Phi_D(k) \wedge \Phi_R^{\not\varepsilon}(k)$, then for every $B \in N$, $0 < \delta < k - x$, $m(\mathcal{R}_{x,\delta}^{B}) = true$ iff $(S, w^m) \to_* (B, w_{x,\delta}^m)$.*

Proof Sketch. By well-founded induction on $k - \delta$ and $\succ$. Apply Lemma 5.1 where necessary. □

Encoding the existence of dissimilar parse trees. Our goal is to express "there exist two dissimilar parse trees that witness $H \Rightarrow_* w_{x,\delta}^m$" in an SMT formula $\Phi_{\text{multi}}(H, x, \delta)$. The key technical problem is how to encode the existence of two such dissimilar parse trees. By definition, to tell if two given parse trees are similar or not, it suffices to only compare their children of the root nodes[6]. Since every parse tree t is a visual representation of a derivation trace for $\mathsf{root}(t) \Rightarrow_* \mathsf{word}(t)$ (or a proof tree that evidences the validity of that derivation judgment), and that the root node with its children on the parse tree corresponds to the *first step* of the trace, "being dissimilar" is essentially "having at least two derivation traces that *differ in the first step*".

Specifically, to make this distinction on two derivation traces for $H \Rightarrow_* w_{x,\delta}^m$, we use (in the first step) either two distinct production rules for H or one binary rule $H \to B_1\ \varphi\ B_2$ in two distinct ways. In the latter, each splits $w_{x,\delta}^m$ into two parts where the prefix is derivable from B_1 and the suffix is derivable from B_2, but the length of the two prefixes (and suffixes) are different. Suppose we can compute the set of all possible choices of such first steps, the formula $\Phi_{\text{multi}}(H, x, \delta)$ simply states that "this set has at least two elements".

We start by defining the elements of this set—possible choices of the first step in deriving $H \Rightarrow_* w_{x,\delta}^m$. A compact *syntactic* representation could simply be a clause α, indicating the production rule $H \to \alpha$ is chosen. In case a binary rule $H \to B_1\ \varphi\ B_2$ is chosen, we further need to specify how

[6]This coincides with the intuitive explanation of being dissimilar—they differ on a node at level 1.

$w^m_{x,\delta}$ gets split—an additional parameter, the prefix length, is enough. Thus, we introduce *choice clauses* with the following syntax:

$$\gamma := \varepsilon \mid a \mid B^{\varphi} \mid B_1^{\delta'}\ \varphi\ B_2$$

where in the last case, δ' ($0 \le \delta' \le \delta$) is the aforementioned prefix length. *Semantically*, a choice clause indicates that the derivation $H \Rightarrow_* w^m_{x,\delta}$ must be valid under the condition that the corresponding first step is taken, which is encodable in SMT formulae:

$$\begin{aligned}
[\![\varepsilon]\!]_{x,\delta} &:= \delta = 0\\
[\![a]\!]_{x,\delta} &:= \delta = 1 \wedge \mathcal{T}_x = a\\
[\![B^{\varphi}]\!]_{x,\delta} &:= \mathit{ite}(\delta = 0, \mathsf{null}(B), \mathcal{D}^B_{x,\delta} \wedge \Phi_{\varphi}(x,\delta)))\\
[\![B_1^{\delta'}\ \varphi\ B_2]\!]_{x,\delta} &:= \Phi_{\varphi}(x,\delta',\delta) \wedge \mathit{ite}(\delta' = 0, \mathsf{null}(B_1), \mathcal{D}^{B_1}_{x,\delta'}) \wedge \mathit{ite}(\delta = \delta', \mathsf{null}(B_2), \mathcal{D}^{B_2}_{x+\delta',\delta-\delta'})
\end{aligned}$$

The set of all possible choices of the first step in deriving $H \Rightarrow_* w^m_{x,\delta}$ is given by

$$\{\gamma \mid \gamma \in \Gamma(H,\delta) \wedge [\![\gamma]\!]_{x,\delta}\},$$

where

$$\begin{aligned}
\Gamma(H,\delta) := \{\epsilon \mid H \to \varepsilon \in P\} \cup \{a \mid H \to a \in P\} \cup \{B^{\varphi} \mid H \to B^{\varphi} \in P\}\\
\cup \{B_1^{\delta'}\ \varphi\ B_2 \mid H \to B_1\ \varphi\ B_2 \in P, 0 \le \delta' \le \delta\}
\end{aligned}$$

gives all choice clauses of H according to the production rules. With this, the formula $\Phi_{\text{multi}}(H,x,\delta)$ should be $|\{\gamma \mid \gamma \in \Gamma(H,\delta) \wedge [\![\gamma]\!]_{x,\delta}\}| \ge 2$, which is equivalent to the following form that uses $\mathit{Two}(\mathcal{P})$, a standard approach to encode "at least two of the propositions in the set $\mathcal{P}$ are true" in propositional logic:

$$\Phi_{\text{multi}}(H,x,\delta) := \mathit{Two}(\{[\![\gamma]\!]_{x,\delta}\}_{\gamma \in \Gamma(H,\delta)})$$

Lemma 5.4. *Let $x + \delta \le k$ and $m \models \Phi_D(k)$, then $m \models \Phi_{multi}(H,x,\delta)$ iff there exist two dissimilar parse trees that witness $H \Rightarrow_* w^m_{x,\delta}$.*

5.3 Formal Properties

For any acyclic LS2NF and any length $k \ge 0$, the proposed SMT encoding is sound and complete:

Theorem 5.5 (Soundness). *If $m \models \Phi(k)$, then $S \Rightarrow_* w^m$ is ambiguous.*

Proof Sketch. By Theorem 4.5 it suffices to show local ambiguity, which is straightforward by applying Lemmas 5.2 to 5.4. □

Theorem 5.6 (Completeness). *If there exists w such that $|w| = k$ and $S \Rightarrow_* w$ is ambiguous, then $\Phi(k)$ is satisfiable.*

Proof Sketch. By Theorem 4.5 we have $(S,w) \to_* (H,h)$. We pick a model m s.t.

$$\begin{aligned}
w^m &= w\\
m(\mathcal{D}^A_{x,\delta}) &\Leftrightarrow A \Rightarrow_* w^m_{x,\delta}\\
m(\mathcal{R}^B_{\varepsilon}) &\Leftrightarrow (S,w) \to_* (B,\varepsilon)\\
m(\mathcal{R}^B_{x,\delta}) &\Leftrightarrow (S,w) \to_* (B, w^m_{x,\delta})
\end{aligned}$$

By definition, $\Phi_D(k)$, $\Phi^{\varepsilon}_R(k)$ and $\Phi^{\not\varepsilon}_R(k)$ are satisfiable. Given that $(S,w) \to_* (H,h)$, h must be a subword of w, say $h = w_{x_h,\delta_h}$ for some x_h, δ_h. By Lemma 5.4, $\Phi_{\text{multi}}(H,x_h,\delta_h)$ is satisfiable. □

For every length k, $O(k^2)$ SMT variables are created, and for each variable, an enumeration loop similar to CYK [Younger 1967] creates $O(k)$ terms, where we also use $O(n)$ space to go through the n production rules. Thus, the space complexity for $\Phi(k)$ is $O(n \cdot k^3)$.

5.4 Ambiguous Sentence Generation

Our bounded ambiguity checker is facilitated by a bounded loop where the bound is specified by the user. In the k-th iteration (initially $k = 1$), logical constraints for finding an ambiguous sentence with length k are encoded as an SMT formula $\Phi(k)$. We rely on a backend SMT solver (e.g., Z3) to check its satisfiability: if it is satisfiable under some model, say $m \models \Phi(k)$, then we are able to decode an ambiguous sentence w^m from m, and the loop exits immediately; otherwise, the next iteration is entered until the user-specified loop bound is reached. In this way, a shortest nonempty ambiguous sentence is obtained.

6 LAYOUT CONSTRAINT SYNTHESIS

In Lay-it-out, user interaction happens when an ambiguous sentence is found. The user describes their intent in disambiguating this sentence by formatting it in distinct ways to accord with the parse trees, as demonstrated in § 2. Then, the layout constraint synthesizer recommends a set of candidate *layout transformation rules*, each of which transforms a layout-free clause into one with layout constraint. The user selects a subset to accept, and Lay-it-out applies them to the original grammar, yielding a refined grammar with layout constraints added. Intuitively, the output grammar is more restricted ("less ambiguous") than the original one. The more rounds of interactions being made, the closer the refined grammar will be to the user's expectation.

6.1 Refinement

The synthesis algorithm aims to *refine* an input grammar. This notion is defined as follows:

Definition 6.1 (Refinement). Let $G = (N, \Sigma, P, S)$ and $G' = (N, \Sigma, P', S)$ by two LS2NFs. We say G' is a *refinement* of G, if for every production rule $r \in P'$, one of the following holds:

(1) $r \in P$;
(2) $(A \longrightarrow B) \in P$ and $r = A \longrightarrow B^{\varphi}$ for some φ;
(3) $(A \longrightarrow BC) \in P$ and $r = A \longrightarrow B\ \varphi\ C$ for some φ.

Proposition 6.2. *Let G' be a refinement of G. If w is unambiguous under G, then it is also unambiguous under G'.*

This proposition explains the key idea of the refinement relation: no more ambiguous sentences are introduced in the refined grammar, with several layout constraints added (case (2) and (3) of Definition 6.1). However, the refined grammar might be equal to the original one as the relation is *reflexive*. To avoid this useless case, the user is asked to accept *at least one* candidate synthesized by the algorithm (otherwise, the algorithm fails) so that the refined grammar will contain fewer ambiguous sentences. In this way, our grammar refinement loop is converging.

6.2 Synthesis Algorithm

Our synthesis algorithm (Algorithm 1) takes an original grammar with user feedback as input and produces a refined grammar (when not fail) according to the user's selection of layout transformation rules. The user *feedback* F is a set of reformatted sentences. The algorithm builds the parse trees t_w for each $w \in F$ under the original grammar G (line 7). Based on these parse trees, candidate layout transformation rules are synthesized. Each *layout transformation rule* is in the form of $\alpha \rightsquigarrow \alpha'$, by Definition 6.1, the two clauses α and α' must contain the same nonterminals and the left-hand side

Algorithm 1: Layout Constraint Synthesis

Input: grammar $G = (N, \Sigma, P, S)$, user's feedback F
Output: refined grammar G'

1 **foreach** $r \in P$ **do**
2 **if** $r = A \to B$ **then**
3 $\Psi[r] \leftarrow \{B \rightsquigarrow B^{\varphi} \mid \varphi \in \Phi^{\text{unary}}\}$
4 **else if** $r = A \to B_1 B_2$ **then**
5 $\Psi[r] \leftarrow \{B_1 B_2 \rightsquigarrow B_1 \; \varphi \; B_2 \mid \varphi \in \Phi^{\text{binary}}\}$
6 **foreach** $w \in F$ **do**
7 $t_w \leftarrow$ parse tree of w
8 **foreach** $t \in t_w$ *in depth-first order* **do**
9 **if** $t = \mathsf{unary}(A, t')$ **then**
10 **let** $B = \mathsf{root}(t')$, $w = \mathsf{word}(t')$
11 remove $\{B \rightsquigarrow B^{\varphi} \mid \neg\varphi(w)\}$ from $\Psi[A \to B]$
12 **else if** $t = \mathsf{binary}(A, t_1, t_2)$ **then**
13 **let** $B_i = \mathsf{root}(t_i)$, $w_i = \mathsf{word}(t_i)$ for $i = 1, 2$
14 remove $\{B_1 B_2 \rightsquigarrow B_1 \; \varphi \; B_2 \mid \neg\varphi(w_1, w_2)\}$ from $\Psi[A \to B_1 B_2]$
15 **if** $\forall r : \; \Psi[r] = \varnothing$ **then return** "inconsistent"
16 **let** $\varnothing \neq \Psi' \subseteq \bigcup_{r \in P} \Psi[r]$ be the user accepted candidates
17 $G' \leftarrow$ apply Ψ' to G
18 **return** G'

must be layout-free. Otherwise, inconsistent layout constraints are potentially introduced in the refined grammar, which is forbidden in our approach.

Realizing that the set of layout constraints one uses in a concrete grammar is *determined* and *finite*, the synthesis algorithm can be facilitated in an *enumerative* fashion: every possible layout constraint is attempted, and the ones that are *consistent*, i.e., all parse trees are still valid when the layout constraint is added, are included in the candidate set. By default, our algorithm considers all the built-in layout constraints (§ 3).

In the algorithm, we use $\Psi[r]$ to maintain the set of all layout transformation rules for r that are consistent with the parse trees we have visited so far. The complete set of candidates is their union $\bigcup_{r \in P} \Psi[r]$. Before any parse tree is visited, $\Psi[r]$ is initialized to the full set (line 3 and 5), including all possible unary (and binary) layout constraints Φ^{unary} (and Φ^{binary}). Then, we traverse every subtree t in every parse tree t_w of $w \in F$ (line 6 and 8) and remove the inconsistent ones in the corresponding $\Psi[r]$ (note that r matches the structure of t), by checking if the layout constraint is fulfilled on t (line 11 and 14). When all parse trees get processed, $\Psi[r]$ now contains the set of all consistent rules. If $\Psi[r]$ is empty for all r, then G cannot be refined due to inconsistency (line 15). Otherwise, the user selects a subset of the candidate transformation rules (line 16), then the algorithm applies them on G (line 17), yielding a refined grammar G' (line 18).

7 COQ MECHANIZATION

All the definitions and theorems presented in § 4 and § 5 were mechanized in the Coq proof assistant. Our proof artifact[7] consists of 10 Coq files and ~2 k lines of code (excluding comments and blanks).

[7] https://github.com/lay-it-out/LS2NF-theory

It relies on helpers lemmas (mostly on lists) from a popular library `coq-stdpp`[8]. On a MacBook Pro with an Apple M1 chip and 16 GB memory, a complete verification took ~12 s.

There was only one axiom we made: in the type "grammar Σ N" for LS2NFs, the two type parameters Σ and N (they resp. encode the terminal and nonterminal set) have *decidable equality*, that is, any two elements are either equal or not equal (i.e., in Coq notion $\forall x, y : \{x = y\} + \{x \neq y\}$). This is true in reality: the terminal and nonterminal sets consist of a finite number of user-specified symbols whose equality can be trivially judged.

8 IMPLEMENTATION

We implemented the proposed interactive grammar design framework as a prototype system called Lay-it-out[9]. It has a graphical *frontend* for user interaction, and a *backend* that realizes the core functionality—ambiguous sentence generation and layout constraint synthesis. The two were connected via a group of RESTful APIs and WebSocket.

The frontend was developed in Vue.js, a JavaScript Framework for building web user interfaces (UIs). Web UIs were built to support: (1) viewing the parse trees of a generated ambiguous sentence, (2) reformatting the ambiguous sentence via clicks and drags, and (3) selecting layout constraints from a candidate set. The frontend also supports loading input grammars from EBNF files and saving refined grammars to EBNF files.

The backend was developed in Python 3. Requests from the Web UI are delegated to the backend with the help of a middleware programmed in Node.js. Mainly three APIs are supported. (1) *Preprocessing*: Load an input grammar (in EBNF) and reduce it into an LS2NF. Meanwhile, nullable nonterminals are precomputed via breadth-first search [Lange and Leiß 2009], and cycles are detected via Tarjan's algorithm [Tarjan 1972] performed on the graph representation of the LS2NF (the cyclic rules will be reported, and the user is required to remove them manually). (2) *Ambiguous sentence generation*: Generate a shortest ambiguous sentence (if any) following the bounded loop explained in § 5.4. SMT formulae are generated according to § 5.2 and we rely on the PySMT library [Gario and Micheli 2015] to invoke the Z3 solver. (3) *Layout constraint synthesis*: Recommend candidate layout constraints based on the user's feedback, following Algorithm 1.

9 EVALUATION

At the heart of Lay-it-out is a novel *semi-automated* approach for designing layout-sensitive grammars, where the layout constraints are synthesized and confirmed through possibly multiple rounds of user interaction. To better understand the effectiveness of this approach and its pros and cons over the traditional *manual* approach—users specify layout constraints totally by hand—we conducted a comprehensive evaluation to seek answers to the following research questions:

- RQ1: How *effective* is Lay-it-out at disambiguating grammars?
- RQ2: How *useful* is Lay-it-out, from the user's perspective?

We answer RQ1 by case studies of two real-world layout-sensitive grammars (§ 9.1,§ 9.2) and RQ2 via a user study on 8 recruited participants (§ 9.3).

Environment. All experiments were conducted on a machine with AMD(R) EPYC(TM) 7H12 CPU and 1024 GB memory, running Ubuntu 20.04 and Python version 3.9.7. We set bound $k = 20$ for ambiguity checking.

[8] https://gitlab.mpi-sws.org/iris/stdpp

[9] The artifact, together with the evaluation data (of the next section), is included in our supplementary material.

start → block-node
block-node → tokens | block-sequence | block-map
off-node → token | block-map
off0-node → block-sequence
tokens → token*
token → t
block-sequence → ‖sequence-item‖+ (3)
sequence-item → (- start)▷ (4)
block-map → key-val
key-val → explicit-key-val | implicit-key-val
explicit-key-val → explicit-key ‖ explicit-val (5)
explicit-key → off-explicit-key | off0-explicit-key
off-explicit-key → (? off-node)▷ (6)
off0-explicit-key → (? off0-node)▷ (7)
explicit-val → off-explicit-val | off0-explicit-val
off-explicit-val → (: off-node)▷ (8)
off0-explicit-val → (: off0-node)▷ (9)
implicit-key-val → off-implicit-key-val | off0-implicit-key-val
off-implicit-key-val → implicit-key off-node
off0-implicit-key-val → implicit-key off0-node
implicit-key → ⟨tokens :⟩ (10)

Fig. 5. The final refined grammar (after 3 rounds of interaction) of YAML's layout-sensitive subset. The initial input grammar is the layout-free version.

9.1 Case Study: YAML

YAML is a human-readable serialization format for both describing configurations and exchanging data between systems/applications. In this case study, we reproduced the design phase of YAML's layout-sensitive subset extracted from its language reference. This subset covers a rich set of layout-sensitive features and is conceptually complex because lists and maps are allowed to be deeply nested.

Fig. 5 depicts both the initial and the final refined versions of the studied grammar, with 21 rules in EBNF and 37 rules in LS2NF. The initial input we fed to Lay-it-out was the layout-free version (i.e., removal of all layout constraints). It took us three rounds of interactions, with ~30 min of work, until we obtained the refined version that has no ambiguity up to the preset bound ($k = 20$). This refined grammar consists of 8 layout constraints: 2 alignment, 3 offside, 2 offside-align, and 1 one-line.

Detailed information for the three interaction rounds is presented in Table 1. In all rounds, the ambiguous sentence was generated within 2 seconds. The longest sentence had six tokens (in rounds 2 and 3, see the second column). The number of parse trees for each generated sentence was 2. The parse trees succinctly represented in equivalent JSON format, together with the reformatted sentences provided by us based on our understanding of YAML's official grammar, are listed in the third column. The layout constraints we added to the initial grammar for each round are listed in the last column, and in total, we added 8 layout constraints.

Table 1. Interaction rounds of disambiguating the YAML's subset.

Round	Ambig. sentence	Parse trees with reformatted sentences		Layout constraints added (linking Fig. 5)

1	`- -`	`[null,null]`
		`-`
		`-`
			`[[null]]`
			`-`
			`  -`
				Eq. (3), Eq. (4)

2	`? - : t : -`	`{[null]:`
		`  {"t": [null]}}`
		`?`
		`-`
		`:`
		`  t :`
		`  -`
			`{[{null: "t"}]:`
			`  [null]}`
			`?`
			`-`
			`  :`
			`  t`
			`:`
			`-`
				Eq. (5), Eq. (7), Eq. (9)

3	`? t`
	`: t`
	`: -`
		`{"t":{"t":[null]}}`
		`?`
		`  t`
		`:`
		`  t : -`
			`{{"t":"t"}: [null]}`
			`?`
			`  t : t`
			`: -`
				Eq. (6), Eq. (8), Eq. (10)

9.2 Case Study: SASS

SASS is a CSS (Cascading Style Sheets) preprocessor language widely used in web frontend development. It is self-described as "Syntactically Awesome Style Sheets". A survey [Coyier 2012] shows that among all developers who use a CSS preprocessor, SASS took 41% of the market, being the second most popular option. As a preprocessor language, SASS has a rich feature set comparable to general-purpose programming languages, such as nested blocks, mixins, and control structures, which makes it Turing-complete. Its grammar has 133 lines of rules, which is on the same scale as Python. These rich features use layout-sensitive syntax, which makes SASS a complex and interesting case to test the ability of Lay-it-out on designing a complete grammar.

This case study was conducted on the *full grammar* of SASS. The initial input we fed to Lay-it-out—the layout-free version—consists of 83 rules in EBNF and 564 rules in LS2NF (after preprocessing). It took us 16 rounds of interactions, with ~10 h amount of work, to obtain a (bounded) unambiguous grammar. In the refined grammar, 73 layout constraints were added: 17 were Kleene stars/pluses, 30 were indentation/offside, 2 were alignment, and 24 were one-line.

Among all interaction rounds, the lengths of the generated ambiguous sentences are depicted in Fig. 6, with 12 being the longest. Each generated ambiguous sentence had two parse trees. On average, 3.9 constraints were introduced in each round. Like most programming languages, SASS has a layered structure, e.g., every document is composed of statements, block statements usually contain a selector and children statements, etc. We exploited the *layered structure* of SASS in the last round: we changed the start symbol and generated an ambiguous subword instead.

SMT solving time and formula complexity. We also investigated the execution time under different sentence length k. Here we define the total execution time by "formula generation time and SMT solving time". Most iterations finished in 0.5 h ($k < 10$) with an exception of 4 cases. The

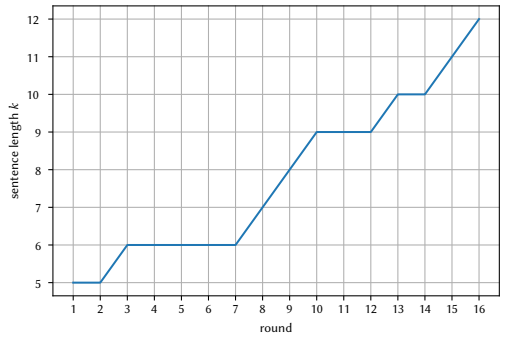

Fig. 6. Ambiguous sentence length in each round.

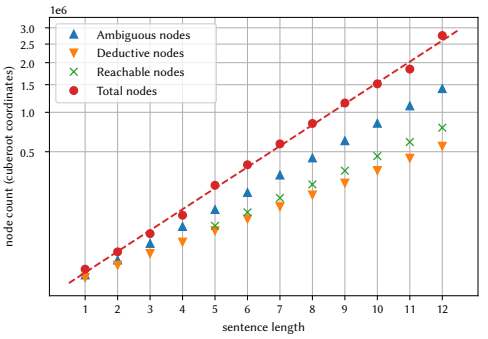

Fig. 7. Growth of SMT formula size.

other 4 cases each took around 0.75 h to 1.5 h. For cases within 0.5 h, the average formula size is 303,695. In contrast, the SMT formula sizes of the other 4 cases, annotated on the figure, are all above 10^6. In general, formulae of such size are quite hard for SMT solvers.

To characterize the complexity of the encoded formula, we look into the average formula size[10] of different sentence lengths. Note that SMT solver was invoked many times in our experiment, and we collected the formula size of each invocation (no matter whether the solving result is sat or unsat) as samples. The relationship between SMT formula size and sentence length is plotted in Fig. 7. We used linear regression to check if the growth conforms to our theoretical complexity. The R^2 is a statistical measure between -1 and 1, indicating how much one variable relates to another. The regression line has $R^2 \approx 0.9975$. Thus, the results are consistent with the theoretical complexity $O(n \cdot k^3)$.

Although the SMT solving took hours due to the large complexity, it is still much better than manually constructing an ambiguous sentence: without the help of Lay-it-out, we (and we believe for many users too) would either give up after several hours' painful attempts or construct a sentence that is too complicated to analyze the cause of ambiguity.

Answers to RQ1. The two case studies give us confidence that our approach is *effective* and *robust* for a complete grammar design phase. The generated ambiguous sentences were relatively short. By viewing the parse trees of the generated ambiguous sentences, it was quite intuitive for us to understand the cause of the ambiguity and provide the reformatted sentences. Moreover, this intuition gave us insights to select from the suggested layout constraints to eliminate this ambiguity.

9.3 User Study

This user study targets the usability of Lay-it-out (RQ2). Our main purpose is to study the user's preference between the manual and the semi-automated approach for layout-sensitive grammar design. We prepared ambiguous CFGs as disambiguation tasks and recruited participants to solve them using the two approaches. Based on their experience obtained from solving the tasks, we conducted a survey to collect their thoughts, comments, and—most importantly—preferences between the two approaches.

[10]The formula size is the number of nodes on its AST representation, computed by the PySMT [Gario and Micheli 2015] library function `pysmt.shortcuts.get_formula_size(f)`.

Table 2. Disambiguation tasks with reference disambiguated grammars.

#	Lang.	Ambiguous CFG	Reference disambiguation
T1	YAML	document → list list → item* item → `-` list | `-` `id`	document → list list → ‖item‖* item → (`-` list)$^\triangleright$ | `-` `id`
T2	F#	let-stmt → `let` decl* `in` expr decl → expr `=` expr expr → `id` | `id` `id` | let-expr	let-stmt → `let` ‖decl‖* `in` expr decl → (expr `=` expr)$^\triangleright$ expr → `id` | `id` `id` | let-expr
T3	F#	start → `match-id` rules match-id → match `id` with rules → ((| `id` `->`) (start | `id`))$^+$	start → `match-id` ◫ rules match-id → ⟨match `id` with⟩ rules → ‖(| `id` `->`) (start | `id`)$^\triangleright$‖*
T4	Haskell	document → stmt$^+$ do-block → `do` stmt$^+$ stmt → (`main` `=` do-block) | (`putStrLn` `id`)	document → ‖stmt‖$^+$ do-block → (`do` ‖stmt‖$^+$)$^\triangleright$ stmt → (`main` `=` do-block)$^\triangleright$ | (`putStrLn` `id`)$^\triangleright$
T5	Python	body → stmt$^+$ for-stmt → `for` binder `in` `range(n)` `:` body stmt → for-stmt | `pass` binder → (`id` `,`)* `id`	body → ‖stmt‖$^+$ for-stmt → ⟨`for` binder `in` `range(n)` `:`⟩ ◫ body stmt → for-stmt | `pass` binder → (`id` `,`)* `id`
T6	Haskell	document → assign-stmt$^+$ assign-stmt → (`id` `=` expr) (where$^?$) where → `where` assign-stmt$^+$ expr → `id` `+` `id`	document → ‖assign-stmt‖$^+$ assign-stmt → ((`id` `=` expr) (where$^?$))$^\triangleright$ where → `where` ‖assign-stmt‖$^+$ expr → `id` `+` `id`
T7	YAML	map → `id` `id` map-body map-body → expl-or-impl-kv$^+$ expl-kv → expl-k expl-v$^?$ expl-k → `?` k-or-v expl-v → `:` k-or-v impl-kv → `id` `:` `id`$^?$ k-or-v → (map-body | `id`)$^?$ expl-or-impl-kv → expl-kv | impl-kv	map → `id` `id` map-body map-body → ‖expl-or-impl-kv‖$^+$ expl-kv → expl-k ‖ expl-v$^?$ expl-k → `?` k-or-v$^\triangleright$ expl-v → `:` k-or-v$^\triangleright$ impl-kv → (`id` `:` `id`$^?$)$^\triangleright$ k-or-v → (map-body | `id`)$^?$ expl-or-impl-kv → expl-kv | impl-kv
T8	Python	document → `stmt`$^+$ stmt → `pass` | while-stmt | decl while-head → `while` `True` `:` body-stmt → stmt | `break` while-stmt → while-head body-stmt$^+$ decl → ident `=` list ident → `foo` | `bar` | `baz` list → `[` (ident `,`)* ident `]`	document → ‖`stmt`‖$^+$ stmt → `pass` | while-stmt | decl while-head → ⟨`while` `True` `:`⟩ body-stmt → stmt | `break` while-stmt → while-head ◫ ‖body-stmt‖$^+$ decl → ident `=` list ident → `foo` | `bar` | `baz` list → `[` (ident `,`)* ident `]`

Preparing CFGs as disambiguation tasks. Like in our case studies, we obtained ambiguous CFGs by removing the layout constraints from layout-sensitive grammars extracted from language references/manuals. As listed in Table 2, we prepared eight ambiguous CFGs extracted from 4 popular languages as disambiguation tasks, numbered T1-T8. Considering the limited time (~2 h) that participants have, the maximum number of the production rules (in EBNFs) did not exceed 10.

Recruting participants. The eligible participants for this user study should be potential users of Lay-it-out: they should have experience in CFGs and basic knowledge of the layout-sensitive grammars they wish to design. To select potential participants from the students (both undergraduate and graduate) in our university, we developed the following specific requirements: (1) the participant has taken the compiler course (that covers CFGs and parsing) or reached the same level of this course; (2) the participant has a reasonable interest in grammar design; and (3) the

participant is familiar with the four layout-sensitive languages that appeared in Table 2, tested by a quiz created by us (the questions were mostly about how to parse a given code snippet). In the end, we recruited 8 participants: 5 undergraduates and 3 graduates.

Assigning tasks to participants. In order to balance the amount of work for each participant (numbered P1-P8), we assigned the 8 disambiguation tasks (as listed in Table 2) in the following way: Odd-numbered participants solve the odd-numbered tasks using the manual approach and the even-numbered using the semi-automated approach; Even-numbered participants do the opposite. In this way, each participant was asked to solve 4 tasks manually and the other 4 tasks using Lay-it-out. Each task was solved the same number of times using two different methods.

Note that we do *not* simply ask each participant to solve all the tasks in both approaches because this is indeed *biased*: For one particular task, whether the participant applies which approach first, the old knowledge can inevitably influence their solutions using the other approach afterward, which makes their user experience with the latter approach inaccurate.

Executing the experiments. Prior to the experiments, we gave participants a tutorial of Lay-it-out's frontend, which the participants will rely on to solve the disambiguation tasks, including a new functionality we built for manually adding layout constraints without the aid of Lay-it-out. Then, the participants started to solve the tasks assigned to them (as mentioned above). We tested the bounded ambiguity of their submitted grammars for manual tasks against our bounded ambiguity checker (with bound $k = 20$). We found 6 out of the 32 grammars submitted by 3 out of the 8 participants were ambiguous due to two reasons: missing a layout constraint and using a weaker layout constraint (i.e., $(\cdot)^{\trianglerighteq}$ instead of $(\cdot)^{\triangleright}$). By contrast, no (bounded) ambiguity was found in the refined grammar for semi-automated tasks.

Distributing questionnaires. After the user experiments, all participants were invited to provide their responses to 4 survey questions (SQs) in a questionnaire depicted in Fig. 8: SQ1 for measuring the difficulty of the manual approach, SQ2 and SQ3 for evaluating the usability of Lay-it-out, and SQ4 for the preferences between the two approaches. All participants responded to all SQs, and their answers are summarized in Fig. 8 too.

Responses to SQ1. Participants rated an average difficulty of 3.6/5.0 on manually adding layout constraints, with half rated "neutral" and the other half "difficult" or "very difficult". Regarding the reasons, all agreed that ensuring the unambiguity of the grammar by hand is difficult, and half agreed that finding an ambiguous sentence is challenging.

Responses to SQ2 & SQ3. We created these two SQs to evaluate the usability of the two kinds of interactions a user needs to make: providing reformatted sentences based on an understanding of the ambiguous sentence with its parse trees, and selecting from the set of recommended layout constraints. For the former, participants scored an average *satisfaction* of 4.4/5.0, with most participants (7/8) scored either "satisfied" or "very satisfied"— they are quite satisfied with the readability of the generated ambiguous sentence and its parse trees. For the latter, participants rated an average *difficulty* of 2.6/5.0, with most participants (7/8) rated "very easy", "easy" or "neutral"— they found the interaction not hard. We further interviewed the participant who rated "difficult": the participant was unsatisfied with the lack of a "undo" command, which we consider a primary future enhancement of our frontend.

Responses to SQ4 (Answers to RQ2). Based on participants' votes, *all* preferred the semi-automated approach guided by Lay-it-out. Regarding why they made this decision, the top 3 reasons were: (1) Lay-it-out can automatically generate ambiguous sentences, while it is hard to construct them by hand; (2) Lay-it-out can guarantee a grammar is unambiguous under a

SQ1: How would you rate the difficulty of manually adding layout constraints to disambiguate grammars?

★☆☆☆☆	(very easy)	0/8
★★☆☆☆	(easy)	0/8
★★★☆☆	(neutral)	4/8
★★★★☆	(difficult)	3/8
★★★★★	(very difficult)	1/8

(average: 3.6, median: 3.5)

Why manually adding layout constraints is easy/difficult? (Multiple choice)

It is easy because it is intuitive to figure out the needed layout constraints. 0/8

It is easy because I can quickly identify an ambiguous sentence with its parse trees. 0/8

It is difficult to find an ambiguous sentence. 4/8

It is difficult to guarantee the unambiguity of the grammar. 8/8

It is difficult because I have no idea where to add which layout constraint. 2/8

SQ2: How satisfied are you with the readability of the ambiguous sentences and parse trees produced by Lay-it-out?

★☆☆☆☆	(very unsatisfied)	0/8
★★☆☆☆	(unsatisfied)	0/8
★★★☆☆	(neutral)	1/8
★★★★☆	(satisfied)	3/8
★★★★★	(very satisfied)	4/8

(average: 4.4, median: 4.5)

SQ3: How would you rate the difficulty of selecting from the candidate layout constraints recommended by Lay-it-out?

★☆☆☆☆	(very easy)	1/8
★★☆☆☆	(easy)	2/8
★★★☆☆	(neutral)	4/8
★★★★☆	(difficult)	1/8
★★★★★	(very difficult)	0/8

(average: 2.6, median: 3.0)

SQ4: Which approach do you think is better?

The manual approach. 0/8

The semi-automated approach guided by Lay-it-out. 8/8

Fig. 8. User survey questions and answers.

certain bound; and (3) Lay-it-out can suggest candidate layout constraints, narrowing down the possibilities compared to the manual approach. In short, Lay-it-out addresses some of the key difficulties of the manual approach.

9.4 Threats to Validity

The participants we recruited—undergraduate/graduate students from our university—may not represent all sorts of potential users of Lay-it-out. However, they represent novice users with an elementary knowledge of grammar. Since they had positive feedback on the usability of Lay-it-out, we believe that expert users, who are more experienced in grammar, will find this tool even easier to use.

Due to the limited time the participants could spend on the experiments, the grammars we prepared (i.e., Table 2) were small. A potential threat is that our findings may not generalize to larger grammars. We mitigated this issue in two ways. First, we conducted a follow-up via online meetings with 2 participants, collecting their thoughts on disambiguating the SASS grammar (one of our case studies, § 9.2). Both said that it is "nearly impossible" to manually disambiguate such a large grammar, but the semi-automated approach makes disambiguation somewhat easier. Second, one of the authors conducted case studies (§ 9.1 and § 9.2) on disambiguating a middle-scale and a large-scale grammar (i.e., SASS) via Lay-it-out. Based on our experience, the usage of the tool on large grammars is similar to those in the user study. In particular, the autogenerated ambiguous sentences were helpful in identifying the cause of ambiguity.

10 RELATED WORK

Layout-sensitive languages & parsing. In 1966, Landin [1966] first introduced layout-sensitive languages, which has influenced the syntax design of many later programming languages. As an extension to CFG, *Indentation-sensitive CFG* (ISCFG) [Adams 2013] is expressive of layout rules and derives LR(k) and GLR algorithms for parsing these layout-sensitive grammars. In ISCFG, symbols are annotated with the numerical relation that the indentation of every nonterminal must have with that of its children. Although ISCFG has a formal theory and is parser-friendly, it takes too much effort for a human to manually specify those annotations.

In a declarative layout specification [Erdweg et al. 2013], layout constraints are expressed with primitives that support direct access to the position of a certain "border" of a code block. In order to realize layout-sensitive parsing, a naive approach is to generate every possible parse tree with GLR parsing first while neglecting layouts, and then filter with layout constraints. This approach is, however, inefficient for practical applications. To improve performance, they identify a subset of constraints that are independent from context-sensitive information and enforce them at parse time. Later, another version with high-level specification is proposed by Amorim et al. [2018]. Our grammar declarative specification notations are inspired by theirs. The difference lies in how we regard layout constraints: in their specification, constraints are attached to a normal layout-free production rule, whereas in ours, constraints are part of the rule. Apart from generalized parsing, Iguana [Afroozeh and Izmaylova 2015, 2016] is a novel parsing framework based upon *data-dependent grammars* [Jim et al. 2010], which extends a CFG with arbitrary computation, variable binding and logical constraint. Iguana translates high-level declarations into equations that are expressive in data-dependent grammars.

Brzozowski derivatives [Brzozowski 1964] have been rediscovered recently to simplify the explanation to parsers. Brachthäuser et al. [2016] propose a new parser combinator library with first-class derivatives, gaining fine-grained control over an input stream. It is still an open question whether their framework can implement alignment and offside rules modularly, e.g., Haskell's grammar.

Word enumeration. Madhavan et al. [2015] propose a practical approach to check the equivalence of CFGs, based on random enumeration of parse trees and words. To generate words of a fixed length, they first transform the input CFG into a restricted CFG that can only derive words

of the given length, and then apply the random enumeration on it. Based on our knowledge, it is more challenging to apply random enumeration techniques to generate ambiguous words. Instead, constraint solving is more suitable to solve such a conditional search problem.

Grammar synthesis. Is it possible to generate a parser from examples? Mernik et al. [2003] raised this question and attempted genetic programming methods. In the recent decade, the research community has had significant interest in *programming by examples* [Gulwani 2011; Polozov and Gulwani 2015] and novel approaches have been proposed to automate parser construction.

Parsify [Leung et al. 2015] is a graphical, interactive system for synthesizing and testing parsers from user-provided examples (a set of sentences). They rely on a GLL parser to identify ambiguous grammars and a few *disambiguation filters* [Klint and Visser 1994; Thorup 1996] to eliminate the well-known associatively and priority issues in grammars of binary expressions. They disambiguate in an interactive manner: possible parse trees are presented to the user, and only one of them is accepted. This does not apply in our situation where all parse trees are acceptable *simultaneously* due to the inadequate layout information.

Glade [Bastani et al. 2017] is an oracle-based grammar inference system. The algorithm synthesizes a CFG which encodes the language of valid program inputs, beginning with a small set of the target language that the user provides as seed inputs. Although grammar synthesis is a promising direction that can ease the design of grammars and parsers, based on our knowledge, there is still no work on the automated synthesis of layout-sensitive grammars and parsers.

Grammar-based fuzzing. In order to improve the test coverage on programs whose inputs are highly structured, like compilers and interpreters, grammar-based fuzzing is proposed to leverage a user-defined grammar for generating syntactically valid inputs. Black-box fuzzing has been integrated with manually specified grammars to test C compilers [Lindig 2005; Yang et al. 2011], find bugs in PHP and JavaScript interpreters [Holler et al. 2012], and generate plausible inputs with the help of a parser merely [Mathis et al. 2019]. There are also studies integrated with white-box techniques. For example, Godefroid et al. [2008] use a handwritten grammar in combination with a custom grammar-based constraint solver to fuzz a JavaScript interpreter of Internet Explorer 7. CESE [Majumdar and Xu 2007] combines exhaustive enumeration of valid inputs with symbolic execution. In comparison, our problem is to find an ambiguous sentence of the grammar instead of just a random sentence accepted by the grammar.

11 CONCLUSION

We present Lay-it-out, a framework for layout-sensitive grammar design via user interaction. Our SMT-based bounded ambiguity checker produces the shortest ambiguous sentences that help a grammar designer to recognize the ambiguity during the design phase. With a polynomial-space encoding, the checker scales to the complete SASS grammar. Through user interactions, Lay-it-out recommends candidate layout constraints for the grammar designer to select, which relieves the user from manually specifying layout rules. A user study on 8 participants reveals usability of Lay-it-out, and users gain more benefits than the traditional manual approach of designing layout-sensitive grammars.

REFERENCES

Michael D. Adams. 2013. Principled Parsing for Indentation-sensitive Languages: Revisiting Landin's Offside Rule. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 511–522. https://doi.org/10.1145/2429069.2429129

Ali Afroozeh and Anastasia Izmaylova. 2015. One Parser to Rule Them All. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Pittsburgh, PA, USA) *(Onward! 2015)*. ACM, New York, NY, USA, 151–170. https://doi.org/10.1145/2814228.2814242

Ali Afroozeh and Anastasia Izmaylova. 2016. Iguana: A Practical Data-dependent Parsing Framework. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. ACM, New York, NY, USA, 267–268. https://doi.org/10.1145/2892208.2892234

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers: Principles, Techniques, and Tools. *Addison wesley* 7, 8 (1986), 9.

Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. 2018. Declarative Specification of Indentation Rules: A Tooling Perspective on Parsing and Pretty-printing Layout-sensitive Languages. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering* (Boston, MA, USA) *(SLE 2018)*. ACM, New York, NY, USA, 3–15. https://doi.org/10.1145/3276604.3276607

Roland Axelsson, Keijo Heljanko, and Martin Lange. 2008. Analyzing Context-Free Grammars Using an Incremental SAT Solver. In *Automata, Languages and Programming*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 410–422.

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 95–110. https://doi.org/10.1145/3062341.3062349

Dirk Beyer, Matthias Dangl, and Philipp Wendler. 2018. A unifying view on SMT-based software verification. *Journal of automated reasoning* 60, 3 (2018), 299–335.

Nikolaj Bjørner and Leonardo de Moura. 2014. Applications of SMT solvers to program verification. *Notes for the Summer School on Formal Techniques* (2014).

Jonathan Immanuel Brachthäuser, Tillmann Rendel, and Klaus Ostermann. 2016. Parsing with First-class Derivatives. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, New York, NY, USA, 588–606. https://doi.org/10.1145/2983990.2984026

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

N Chomsky and MP Schützenberger. 1963. The Algebraic Theory of Context-Free Languages. In *Studies in Logic and the Foundations of Mathematics*. Vol. 35. Elsevier, 118–161.

Chris Coyier. 2012. https://css-tricks.com/poll-results-popularity-of-css-preprocessors/

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2013. Layout-Sensitive Generalized Parsing. In *Software Language Engineering*, Krzysztof Czarnecki and Görel Hedin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 244–263.

Clark C Evans et al. 2014. Yaml: Yaml ain't markup language.

Marco Gario and Andrea Micheli. 2015. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT workshop*, Vol. 2015.

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

John Gruber. 2012. Markdown: Syntax. *URL http://daringfireball.net/projects/markdown/syntax. Retrieved on June* 24 (2012), 640.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, 317–330. https://doi.org/10.1145/1926385.1926423

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, Berkeley, CA, USA, 38–38. http://dl.acm.org/citation.cfm?id=2362793.2362831

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

Trevor Jim, Yitzhak Mandelbaum, and David Walker. 2010. Semantics and Algorithms for Data-dependent Grammars. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. ACM, New York, NY, USA, 417–430. https://doi.org/10.1145/1706299.1706347

Ian Johnson. 2009. Formal Verification with SMT Solvers: Why and How. In *ACL2 Theorem Proving Seminar at the University of Texas, Autin.*

Paul Klint and Eelco Visser. 1994. Using filters for the disambiguation of context-free grammars. In *Proc. ASMICS Workshop on Parsing Theory*. Citeseer, 1–20.

Donald E Knuth. 1965. On the translation of languages from left to right. *Information and control* 8, 6 (1965), 607–639.

P. J. Landin. 1966. The Next 700 Programming Languages. *Commun. ACM* 9, 3 (March 1966), 157–166. https://doi.org/10.1145/365230.365257

Martin Lange and Hans Leiß. 2009. To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Informatica Didactica* 8, 2009 (2009), 1–21.

Rustan Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *16th International Conference, LPAR-16, Dakar, Senegal* (16th international conference, lpar-16, dakar, senegal ed.). Springer Berlin Heidelberg, 348–370. https://www.microsoft.com/en-us/research/publication/dafny-automatic-program-verifier-functional-correctness-2/

Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive Parser Synthesis by Example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 565–574. https://doi.org/10.1145/2737924.2738002

John Levine. 2009. *Flex & Bison: Text Processing Tools.* " O'Reilly Media, Inc.".

John R Levine, John Mason, John R Levine, John R Levine, Paul Levine, Tony Mason, and Doug Brown. 1992. *Lex & yacc.* " O'Reilly Media, Inc.".

Christian Lindig. 2005. Random Testing of C Calling Conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging* (Monterey, California, USA) *(AADEBUG'05)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/1085130.1085132

Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. 2015. Automating Grammar Comparison. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 183–200. https://doi.org/10.1145/2814270.2814304

Rupak Majumdar and Ru-Gang Xu. 2007. Directed Test Generation Using Symbolic Grammars. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. ACM, New York, NY, USA, 134–143. https://doi.org/10.1145/1321631.1321653

Simon Marlow et al. 2010. Haskell 2010 language report. *Available online http://www.haskell.org/(May 2011)* (2010).

Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed Fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 548–560. https://doi.org/10.1145/3314221.3314651

Marjan Mernik, Goran Gerlič, Viljem Žumer, and Barrett R. Bryant. 2003. Can a Parser Be Generated from Examples?. In *Proceedings of the 2003 ACM Symposium on Applied Computing* (Melbourne, Florida) *(SAC '03)*. ACM, New York, NY, USA, 1063–1067. https://doi.org/10.1145/952532.952740

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 297–310. https://doi.org/10.1145/2980024.2872387

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices* 50, 10 (2015), 107–126.

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 198–216.

Don Syme, Luke Hoban, Tao Liu, Dmitry Lomov, James Margetson, Brian McNamara, Joe Pamer, Penny Orwick, Daniel Quirk, Chris Smith, et al. 2010. The F# 2.0 language specification. *Microsoft, August* (2010).

Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

Mikkel Thorup. 1996. Disambiguating grammars by exclusion of sub-parse trees. *Acta Informatica* 33, 6 (1996), 511–522.

Guido Van Rossum and Fred L Drake. 2011. *The python language reference manual.* Network Theory Ltd.

Eelco Visser et al. 1997. *Syntax definition for language prototyping.* Eelco Visser.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Daniel H Younger. 1967. Recognition and parsing of context-free languages in time n3. *Information and control* 10, 2 (1967), 189–208.