Resolvable Ambiguity

ANONYMOUS AUTHOR(S)

A common standpoint when designing the syntax of programming languages is that the grammar definition has to be unambiguous. However, requiring up front unambiguous grammars can force language designers to make more or less arbitrary choices to disambiguate the language. In this paper, we depart from the traditional view of unambiguous grammar design, and enable the detection of ambiguities to be delayed until parse time, allowing the user of the language to perform the disambiguation. A natural decision problem follows: given a language definition, can a user always disambiguate an ambiguous program? We introduce and formalize this fundamental problem—called the *resolvable ambiguity problem*—and divide it into separate static and dynamic resolvability problems. We provide solutions to the static problem for two restricted classes of languages and sketch proofs of soundness and, depending on the restrictions, completeness. We also provide a sound and complete solution to the dynamic problem for a much less restricted class of languages. The approach is evaluated through two separate case studies, covering both a large existing programming language, and the composability of domain-specific languages.

Additional Key Words and Phrases: Language Composition, Context Free Grammars, Domain-Specific Languages

1 INTRODUCTION

Ever since the early 60s, it has been known that determining whether a context-free grammar is ambiguous is undecidable [Cantor 1962]. As a consequence, a large number of restricted grammars have been developed to guarantee that language definitions are unambiguous. This traditional view of only allowing unambiguous grammars has been taken for granted as the only truth: *the* way of how the syntax of a language must be defined [Aho et al. 2006; Cooper and Torczon 2011; Ginsburg and Ullian 1966; Sudkamp 1997; Webber 2003]. However, recent work by Palmkvist and Broman [2019] shows why carefully designed ambiguous syntax definitions are in some cases preferable compared to completely unambiguous definitions. Specifically, instead of forcing language designers to make more or less arbitrary choices to disambiguate the language, a more natural solution may be to leave the decision up to the programmer.

Another area where ambiguous grammars naturally arise is in domain-specific language development. An important objective when designing domain-specific languages is to be able construct languages by composing and extending existing languages. Unfortunately, the composition of grammars easily produces an ambiguous grammar, even if the original grammars are all unambiguous on their own. Approaches for solving this problem can be broadly split into two categories: those that handle ambiguities on the grammar-level (detection, prevention, etc.), and those that work on particular examples of ambiguous programs. The former is well explored in the form of heuristics for ambiguity detection [Axelsson et al. 2008; Basten 2011; Brabrand et al. 2007] or restrictions on the composed grammars [Kaminski and Van Wyk 2013], while the latter has received relatively little attention.

This paper focuses on the problems that arise if a grammar definition is not guaranteed to be unambiguous. Concretely, if parsing a program could result in several correct parse trees, the grammar is obviously ambiguous. The programmer then needs to disambiguate the program by, for instance, inserting extra parentheses. We say that a program is *resolvably ambiguous* if the ambiguity can be resolved by modifying the program, such that each of the possible parse trees can be selected using different modifications. We can divide the problem of resolvable ambiguity into two different

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

decision problems. The *dynamic resolvable ambiguity problem* is as follows: given a parse function for a language, and a specific program that is parsed into a set of parse trees, decide whether it is possible to rewrite the program, such that each parse tree can be generated unambiguously. This problem can be generalized into the *static resolvable ambiguity problem*: given a parse function, decide whether all parsable programs are resolvably ambiguous. Static resolvability means that a language designer can be confident that the end-user (the programmer) can always resolve an ambiguity by modifying the program. Identifying such ambiguities for a *specific* program and suggesting disambiguations is enabled by dynamic resolvability. Static resolvability gives formally verified guarantees on a syntax definition, whereas dynamic resolvability enables the language designer to test the resolvability of the syntax definition, program by program.

In this paper, we solve the static resolvability problem for certain classes of languages, and demonstrate practically how dynamic resolvability may be used when defining language syntax. More specifically, we make the following contributions:

- We formally define the term *resolvable ambiguity*, and provide precise formalizations of the static and dynamic *resolvable ambiguity problems* (Section 4).
- As part of the formalization of the dynamic and static resolvability problems, we define a syntax definition formalism based on Extended Backus–Naur Form (EBNF), where ambiguities can be resolved with grouping parentheses (Section 5).
- We consider versions of the *static resolvability problem* with increasing difficulty by considering two differently restricted classes of languages. We formalize a decidable algorithm that is sound and complete for the first version, and sound for the second (Section 6).
- We devise a decidable algorithm that solves the *dynamic resolvability problem* for general context-free languages with well-balanced parentheses (Section 7).
- We evaluate the dynamic resolvability methodology in the context of two different case studies using a tool and a small DSL for defining syntax definitions (Section 8): (i) a domain-specific language for orchestrating parallel computations, where we investigate the effects of composing language fragments, and (ii) an implementation and evaluation of a large subset of OCaml's syntax, where we study the effect of implementing an under-specified language definition.

Before presenting the above contributions, the paper starts with motivating examples (Section 2), as well as preliminaries (Section 3).

2 MOTIVATING EXAMPLE

In this section, we motivate the need for ambiguous language definitions, where the decision of how to disambiguate a program is taken by the end-user (the programmer) and not the language designer. We motivate our new methodology both for engineering of new domain-specific languages, as well as for the design of existing general-purpose programming languages.

2.1 Domain-Specific Modeling of Components

Suppose we are defining a new textual modeling language, where components are composed in series using an infix operator `--`, and in parallel using an infix operator `||`. For instance, an expression `C1 -- C2` puts components `C1` and `C2` in series, whereas `C1 || C2` composes them in parallel. In such a case, what is then the meaning of the following expression?

```
C1 -- C2 || C3 || C4
```

Are there natural associativity and precedence rules for these new operators? If there are no predefined rules of how to disambiguate this expression within the language definition, it is an ambiguous expression, and a parser generates a set of parse trees. Consider Fig 1, which depicts four

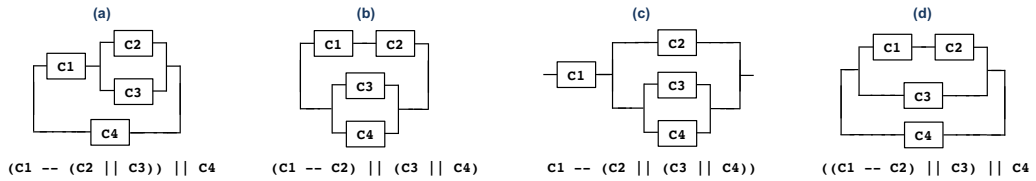


Fig. 1. The figure shows four different alternatives for disambiguating the expression `C1 -- C2 || C3 || C4`. Note that there is a fifth alternative `C1 -- ((C2 || C3) || C4)`. However, the meaning of this expression is the same as (c), presupposed that the parallel operator is associative. In such a case, this expression and the expression in (c) both means that components `C2`, `C3`, and `C4` are composed in parallel.

different alternatives, each with a different meaning, depending on how the ambiguity has been resolved. Clearly, the expression has totally different meanings depending on how the end-user places the parenthesis. However, if a language designer is forced to make the grammar of the syntax definition unambiguous, a specific choice has to be made for precedence and associativity. For instance, assume that the designer makes the arbitrary choice that serial composition has higher precedence than parallel composition, and that both operators are left associative. In such a case, the expression without parenthesis is parsed as Figure 1(d). The question is why such an arbitrary choice—which is forced by the traditional design of unambiguous grammars—is the correct way to interpret a domain-specific expression. The alternative, as argued for in this paper, is to postpone the decision, and instead give an error message to the end-user (programmer or modeler), and expose different alternatives that disambiguate the expression.

2.2 Match Cases in OCaml

Explicit ambiguity is highly relevant also for the design of new and existing general purpose programming languages. The following example shows how an ambiguity error can be clearer than what an existing compiler produces today with the traditional approach.

Consider the following OCaml example of nested `match` expressions, as stated by [Palmkvist and Broman 2019]).

```
1  match 1 with
2    | 1 -> match "one" with
3           | str -> str
4    | 2 -> "two"
```

```
File "./nestmatch.ml", line 4, characters 4-5:
Error: This pattern matches values of type int
       but a pattern was expected which
       matches values of type string
```

The OCaml compiler output is listed to the right. The compiler sees the last line as belonging to the inner **`match`** rather than the outer, as was intended. The solution is simple; we put parentheses around the inner match:

```
1  match 1 with
2    | 1 -> (match "one" with
3             | str -> str)
4    | 2 -> "two"
```

However, the connection between the error message and the solution is not particularly clear; surrounding an expression with parentheses does not change its type. The OCaml compiler makes an arbitrary choice to remove the ambiguity, which may or may not be the alternative the user intended. With a parser that is aware of possible ambiguities, the disambiguation can be left to the end-user, with the alternatives listed as part of an error message.

2.3 Resolvable Ambiguity

The two motivating examples above motivate briefly why grammar and language definitions do not always need to be unambiguous. However, an ambiguous program that cannot be resolved by the end-user is clearly undesirable. In the rest of the paper, we formalize the approach of resolvable ambiguity, and show how we can give guarantees for resolvability.

3 PRELIMINARIES

This section briefly describes the theoretical foundations we build upon.

3.1 Context-Free Grammars

A context-free grammar (CFG) G is a 4-tuple (V, Σ, P, S) where V is a set of non-terminals; Σ a set of terminals, disjoint from V; P a finite subset of $V \times (V \cup \Sigma)^*$, i.e., a set of productions; and $S \in V$ the starting non-terminal.

A word $w \in \Sigma^*$ is recognized by G if there is a sequence of steps starting with S and ending with w, where each step replaces a single non-terminal using a production in P. Such a sequence is called a *derivation*. The set of words recognized by G is written $L(G)$

The standard definition of ambiguity, given a context-free grammar G, is expressed in terms of *leftmost derivations*. A leftmost derivation is a derivation where the non-terminal being replaced is always the leftmost one.

Definition 3.1. A word $w \in L(G)$ is ambiguous if there are two distinct leftmost derivations of that word.

3.2 Automata

A nondeterministic finite automaton (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states; Σ a finite set of terminals; δ a transition function from $Q \times \Sigma$ to finite subsets of Q ; $q_0 \in Q$ an initial state; and $F \subseteq Q$ a set of final states.

A successful run is a sequence of states $r_0, \ldots, r_n$ and a word $a_0 \cdots a_n$ such that:

- $r_0 = q_0$.
- $\forall i \in \{0, 1, \ldots, n-1\}.\ r_{i+1} \in \delta(r_i, a_i)$.
- $r_n \in F$.

We say that the automaton accepts the word $a_0 a_1 \cdots a_n$ iff there is such a succesful run.

A deterministic finite automaton (DFA) has the same definition, except $\delta : Q \times \Sigma \to Q$, i.e., given a state and a symbol there is always a single state we can transition to. NFAs and DFAs have the same expressive power as regular expressions, i.e., for every regular expression there is an NFA and a DFA both reconizing the same language, and vice-versa.

A pushdown automaton extends a finite automaton with a stack from which transitions can push or pop symbols. Formally, a nondeterministic pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where Q is a finite set of states; Σ a finite set of input symbols, i.e., an input alphabet; Γ a finite set of stack symbols, i.e., a stack alphabet; δ a transition function from $Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\}$ to finite subsets of $Q \times (\Gamma \cup \{\lambda\}$; $q_0 \in Q$ the initial state; and $F \subseteq Q$ a set of final states. λ essentially means "ignore", i.e., $\delta(q_1, \lambda, \lambda) = \{(q_2, \lambda)\}$ means: transition from state q_1 without consuming an input symbol (the first λ) and without examining or popping from the current stack (the second λ), to state q_2 without pushing a new symbol on the stack (the third λ).

A successful run is now a sequence of *configurations*, elements of $Q \times \Gamma^*$, starting with (q_0, ϵ), ending with (f, γ) for some $f \in F$ and $\gamma \in \Gamma^*$.

However, in this paper we will only consider pushdown automata with relatively limited stack manipulation, and will thus use some convenient shorthand:

- $p \xrightarrow{a} q$, a transition that recognizes the terminal a and does not interact with the stack at all, i.e., $\delta(p, a, \lambda) \supseteq \{(q, \lambda)\}$.
- $p \xrightarrow{a,+g} q$, a transition that recognizes the terminal a and pushes the symbol g on the stack, i.e., $\delta(p, a, \lambda) \supseteq \{(q, g)\}$.
- $p \xrightarrow{a,-g} q$, a transition that recognizes the terminal a and pops the symbol g from the stack, i.e., $\delta(p, a, g) \supseteq \{(q, \lambda)\}$.

3.3 Visibly Pushdown Languages

A visibly pushdown language [Alur and Madhusudan 2004] is a language that can be recognized by a visibly pushdown automaton. A visibly pushdown automaton is a pushdown automaton where the input alphabet Σ can be partitioned into three disjoint sets Σ_c, Σ_i, and Σ_r, such that all transitions in the automaton has one of the following three forms:

- $p \xrightarrow{c,+s} q$, where $c \in \Sigma_c$ and $s \in \Gamma$; or
- $p \xrightarrow{i} q$, where $i \in \Sigma_i$; or
- $p \xrightarrow{r,-s} q$, where $r \in \Sigma_r$ and $s \in \Gamma$,

i.e., the terminal recognized by a transition fully determines the change to the stack height. The names of the partitions stem from their original use in program analysis, c is for *call*, i for *internal*, and r for *return*. This partitioning gives us some useful properties. Of particular relevance to this paper are the following two points:

- Visibly pushdown languages with the same input partitions are closed under intersection, complement, and union [Alur and Madhusudan 2004]. Intersection in particular is given by a product automaton, i.e., given a pair of VPDAs $(Q_1, \Sigma, \delta_1, q_0, F_1)$ and $(Q_2, \Sigma, \delta_2, q_0', F_2)$ their product automaton has the form $(Q_1 \times Q_2, \Sigma, \delta', (q_0, q_0'), F_1 \times F_2)$ where:

$$\begin{aligned}
\delta'((p_1, p_2), c, \lambda) &= \{((q_1, q_2), (g_1, g_2)) \mid (q_1, g_1) \in \delta_1(p_1, c, \lambda), (q_2, g_2) \in \delta_2(p_2, c, \lambda)\} && \text{where } c \in \Sigma_c \\
\delta'((p_1, p_2), i, \lambda) &= \{((q_1, q_2), \lambda) \mid (q_1, \lambda) \in \delta_1(p_1, i, \lambda), (q_2, \lambda) \in \delta_2(p_2, i, \lambda)\} && \text{where } i \in \Sigma_i \\
\delta'((p_1, p_2), r, (g_1, g_2)) &= \{((q_1, q_2), \lambda) \mid (q_1, \lambda) \in \delta_1(p_1, r, \lambda), (q_2, \lambda) \in \delta_2(p_2, r, \lambda)\} && \text{where } r \in \Sigma_r
\end{aligned}$$

- A visibly pushdown automaton can be trimmed [Caralp et al. 2015], i.e., modified in such a way that all remaining states and transitions are part of at least one successful run; none are redundant. Furthermore, a successful run in the trimmed automaton corresponds to exactly one successful run in the original automaton, and vice-versa.

3.4 Unranked Regular Tree Grammars

Trees generalize words by allowing each terminal to have multiple ordered successors, instead of just zero or one. Most literature considers *ranked* tree languages, where each terminal has a fixed arity, i.e., the same terminal must always have the same number of successors. This is as opposed to *unranked* tree languages, where the arity of a terminal is not fixed. The sequence of successors to a single terminal in an unranked tree tends to be described by a word language (referred to as a horizontal language in [Comon et al. 2007]), often a regular language.

The results and properties presented in this paper are more naturally described through unranked trees, thus all further references to trees are to unranked trees, despite ranked being more common in the literature. We further distinguish terminals used solely as leaves from terminals that may be either nodes or leaves. Since we will use unranked trees to represent parse trees, the former will represent terminals from the parsed word, while the latter represent terminals introduced as internal nodes.

An unranked tree grammar T is a tuple (V, Σ, X, P, S) where:

- V is a set of (zero-arity) non-terminals.
- Σ is a set of zero-arity terminals, used as leaves.
- X is a set of terminals without fixed arity, used as inner nodes or leaves.
- P is a set of productions, a finite subset of $V \times X \times Reg(\Sigma \cup X)$. We will write a production (N, x, r) as $N \longrightarrow x(r)$.

A tree t (containing only terminals from Σ and X) is recognized by T if there is a sequence of steps starting with S and ending with t, where each step either replaces a single non-terminal using a production in P, or replaces a regular expression r with a sequence in $L(r)$. The set of trees recognized by T is written $L(T)$[1].

Finally, $yield : L(T) \longrightarrow \Sigma^*$ is the sequence of terminals $a \in \Sigma$ obtained by a left-to-right[2] traversal of a tree. Informally, it is the flattening of a tree after all internal nodes have been removed.

4 RESOLVABLE AMBIGUITY

This section introduces our definition of *resolvable ambiguity*, and then relates it to standard concepts in formal languages.

A formal language is defined as a set of words, i.e., a subset of Σ^* for some alphabet Σ. To be able to define *resolvable ambiguity*, we additionally have to consider the results of parsing words. In order to stay as general as possible, we first define the notion of an *abstract parser*.

Definition 4.1. An abstract parser P is a triple $(\Sigma, T, parse)$ consisting of

- an alphabet Σ,
- a set of parse trees T, and
- a function $parse : \Sigma^* \longrightarrow 2^T$ that relates words to their parse trees, where 2^T denotes the powerset of T. Additionally, we require that $T = \bigcup_{w \in \Sigma^*} parse(w)$.

This notion of an abstract parser enables the introduction of a particular class of formal languages that we will use throughout the rest of the paper; we call members of this class *parse languages:*

Definition 4.2. Given a word $w \in \Sigma^*$ and an abstract parser $P = (\Sigma, T, parse)$, the set of words contained in the *parse language* $L(P)$ is defined as follows:
$w \in L(P)$ iff $parse(w) \neq \emptyset$.

For example, consider a simple arithmetic language without precedence and parentheses. In such a language, $parse(1 + 2 \cdot 3)$ would produce a set containing two parse trees. The ambiguity of a word $w \in \Sigma^*$ is defined in terms of *parse*:

Definition 4.3. Given an abstract parser $P = (\Sigma, T, parse)$, a word $w \in L(P)$ is ambiguous, written $amb_P(w)$, iff $\exists t_1, t_2 \in T.\ t_1 \neq t_2 \wedge \{t_1, t_2\} \subseteq parse(w)$

Note that the above definition implies that a word $w \in L(P)$, where $P = (\Sigma, T, parse)$, is not ambiguous, or *unambiguous*, if $\exists t \in T.\ parse(w) = \{t\}$.

We can connect the above definition of a parse language to the classical definition of a (formal) language as follows:

- Given a parse language $L(P)$, the corresponding classical language (i.e., set of words) is given by $\{w \mid parse(w) \neq \emptyset\}$.
- If we select a *parse* function that relates words to their leftmost derivations in a given context-free grammar, then our definition of ambiguity corresponds exactly to the classical definition of ambiguity.

[1] Again, to distinguish from regular expressions and context-free languages, all trees will be named T, possibly with a subscript.
[2] Preorder, postorder, or inorder does not matter since terminals in Σ only appear as leaves

A *resolvably ambiguous* word is a word where all its parse trees can be written in an unambiguous way, formally:

Definition 4.4. Given an abstract parser $P = (\Sigma, T, \mathit{parse})$, a word $w \in L(P)$ is *resolvably ambiguous*, written $\rho_P(w)$, iff
$\forall t \in \mathit{parse}(w).\ \exists w' \in \Sigma^*.\ \mathit{parse}(w') = \{t\}$.

The example $\mathit{parse}(1 + 2 \cdot 3)$ is not resolvably ambiguous, since neither of the two resulting trees can be written in any other way. However, if we add grouping parentheses to the language, we find that $1 + (2 \cdot 3)$ and $(1 + 2) \cdot 3$ are unambiguously parsed as the first and second tree, respectively.

Additionally, we define an *abstract parser* to be resolvably ambiguous if all words in its corresponding parse language $L(P)$ are resolvably ambiguous, formally:

Definition 4.5. An abstract parser P is resolvably ambiguous, written $\rho(P)$, iff
$\forall w \in L(P).\ \rho_P(w)$.

We can now state a few things:

- An unambigous word w is trivially resolvably ambiguous, since its only parse tree t can be written unambiguously with w itself ($\mathit{parse}(w) = \{t\}$). The set of resolvably ambiguous words is thus a superset of the unambiguous words.
- If a given parse tree t has only one word w such that $t \in \mathit{parse}(w)$, then w is resolvably ambiguous iff it is unambiguous. In general, $\forall t \in T.\ |\{w \mid t \in \mathit{parse}(w)\}| \leq 1$ implies that the set of resolvably ambiguous words is exactly the set of unambiguous words.

The second point suggests that resolvable ambiguity is only an interesting property if an element of T does not uniquely identify an element of Σ^*. Intuitively, this only happens if *parse* discards some information present in its argument when constructing an individual parse tree. Fortunately, this is generally true for parsing in commonly used programming languages; they tend to discard, e.g., grouping parentheses and whitespace. In general, whatever information *parse* discards can be used by an end-user to disambiguate an ambiguity encountered at parse-time, without changing the parse tree.

We thus propose to loosen the common "no ambiguity" restriction on programming language grammars, and instead only require them to be resolvably ambiguous. However, merely having an arbitrary function *parse* gives us very little to work with, and no way to decide whether the language it defines is resolvably ambiguous or not. The remainder of this paper will thus consider *parse* functions defined using a particular formalism, introduced in Section 5, which gives us some decidable properties.

Before introducing this formalism, however, we introduce the two central problems we consider in this paper:

Static resolvability. Given an abstract parser P, determine whether $\rho(P)$.
Dynamic resolvability. Given an abstract parser P and a word $w \in L(P)$, determine whether $\rho_P(w)$.

Our main concern is producing decision procedures for these problems. First, we define soundness and completeness for the static problem.

Definition 4.6 (Soundness of Static Resolvability). A decision procedure f, solving the static resolvability problem, is sound iff
$f(P) \implies \rho(P)$ for all abstract parsers P

In practice, we may additionally be interested in a decision procedure g such that $g(P) \implies \neg\rho(P)$ for all abstract parsers P. It turns out, we can give g for the parse languages studied in the following sections.

Definition 4.7 (Completeness of Static Resolvability). A decision procedure f, solving the static resolvability problem, is *complete* iff

$\rho(P) \implies f(P)$ for all abstract parsers P

Similarly, we define soundness and completeness for the dynamic problem.

Definition 4.8 (Soundness of Dynamic Resolvability). A decision procedure f, solving the dynamic resolvability problem, is sound iff

$f(\mathit{parse}, w) \implies \rho_P(w)$ for all $w \in \Sigma^*$ and abstract parsers $P = (\Sigma, T, \mathit{parse})$

Analogous to the static case, for the parse languages studied in the following sections, we can also provide a decision procedure g such that $g(\mathit{parse}, w) \implies \neg\rho_P(w)$ for all $w \in \Sigma^*$ and abstract parsers $P = (\Sigma, T, \mathit{parse})$.

Definition 4.9 (Completeness of Dynamic Resolvability). A decision procedure f, solving the dynamic resolvability problem, is complete iff

$\rho_P(w) \implies f(\mathit{parse}, w)$ for all $w \in \Sigma^*$ and abstract parsers $P = (\Sigma, T, \mathit{parse})$

5 PARSE-TIME DISAMBIGUATION

This section describes our chosen language definition formalism, and motivates its design.

The primary purpose of this formalism is, as described in the previous section, to produce a *parse* function, i.e., to describe a word language and assign one or more parse trees to each word. Parse trees are unranked trees; Section 4 suggests that *parse* discards some information, to enable the resolution of some ambiguities; here, we choose to discard grouping parentheses.

With that in mind, we define a language definition D as a set of labelled productions, as described in Fig. 2. Note that we require the labels to uniquely identify the production, i.e., there can be no two distinct productions in D that share the same label. Also note that the right-hand side of a production is a regular expression, rather than the theoretically simpler sequence used in a context-free grammar. For proof-technical reasons (see Section 6.3) we require each regular expression to be unambiguous, in the classical sense, when marks are removed. This can be checked using, e.g., the approach described by Brabrand and Thomsen [2010]. In practice, this is likely desirable regardless, since a user is likely to attach different semantic meanings to each occurrence of a non-terminal.

Each non-terminal appearing in the regular expression of a production carries a *mark* m which is a set of labels whose productions may *not* replace that non-terminal. To lessen clutter, we write $E_\emptyset$ as E. Consider the language definition shown in Fig. 3 which we use as a running example. In the production describing multiplication (m) both non-terminals are marked with $\{a\}$, which thus forbids addition from being a direct child of a multiplication. By "direct child" we mean "without an intermediate node", most commonly grouping parentheses; thus, this enforces conventional precedence.

From D we then generate four grammars: G_D, T_D, G'_D, and T'_D. Technically, only G'_D and T_D are required, G'_D is used as the defined word language and T_D as the parse trees, but the remaining two grammars help the presentation.

- G_D represents a word language describing all semantically distinct programs.
- T_D represents a tree language describing the parse trees of words in $L(G_D)$.
- G'_D is essentially a modified version of G_D, e.g., adding grouping parentheses and other forms of disambiguation (i.e., the result of marks).
- T'_D represents a tree language describing the parse trees of words in $L(G'_D)$.

Fig. 4 shows the four grammars generated from our running example in Fig. 3. The context-free grammars are produced by a rather standard translation from regular expressions to CFGs, while

Terminals	$t \in \Sigma$
Non-terminals	$N \in V$
Labels	$l \in L$
Σ, V, and L disjoint	
Marks	$m \subseteq L$
Regular expressions	$r ::= t \mid N_m \mid r \cdot r \mid r + r \mid \epsilon \mid r^*$
Labelled productions	$N \longrightarrow l : r$

Fig. 2. The abstract syntax of a language definition.

$$
\begin{array}{lcl}
E & \longrightarrow & l : \text{'['}\ (E\ (\text{';'}\ E)^* + \epsilon)\ \text{']'} \\
E & \longrightarrow & a : E\ \text{'+'}\ E \\
E & \longrightarrow & m : E_{\{a\}}\ \text{'*'}\ E_{\{a\}} \\
E & \longrightarrow & n : N
\end{array}
$$

Fig. 3. The input language definition used as a running example, an expression language with lists, addition, and multiplication, with precedence defined, but not associativity. Assumes that N matches a numeric terminal.

$$
\begin{array}{lcl}
E & \longrightarrow & l\,(\ \text{'['}\ (\epsilon + E(\text{';'}\ E)^*)\ \text{']'}\ ) \\
E & \longrightarrow & a\,(\ E\ \text{'+'}\ E\ ) \\
E & \longrightarrow & m(\ E_{\{a\}}\ \text{'*'}\ E_{\{a\}}\ ) \\
E & \longrightarrow & n(\ N\ ) \\
E & \longrightarrow & g(\ \text{'('}\ E\ \text{')'}\ )
\end{array}
$$

(a) T_D, the parse trees of G_D.

$$
\begin{array}{lcl}
E & \longrightarrow & l\,(\ \text{'['}\ (\epsilon + E(\text{';'}\ E)^*)\ \text{']'}\ ) \\
E & \longrightarrow & m(\ E_{\{a\}}\ \text{'*'}\ E_{\{a\}}\ ) \\
E & \longrightarrow & n(\ N\ ) \\
E & \longrightarrow & g(\ \text{'('}\ E\ \text{')'}\ )
\end{array}
$$

(b) T'_D, the parse trees of G'_D.

$$
\begin{array}{lcl}
E & \longrightarrow & \text{'['}\ E_{l1}\ \text{']'} \\
E & \longrightarrow & E\ \text{'+'}\ E \\
E & \longrightarrow & E\ \text{'*'}\ E \\
E & \longrightarrow & N \\
\hline
E_{l1} & \longrightarrow & \epsilon \\
E_{l1} & \longrightarrow & E\ E_{l2} \\
\hline
E_{l2} & \longrightarrow & \epsilon \\
E_{l2} & \longrightarrow & \text{';'}\ E\ E_{l2}
\end{array}
$$

(c) G_D, the generated abstract syntax.

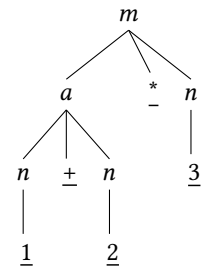
$$
\begin{array}{lcl}
E & \longrightarrow & \text{'['}\ E_{l1}\ \text{']'} \\
E & \longrightarrow & E\ \text{'+'}\ E \\
E & \longrightarrow & E_{\{a\}}\ \text{'*'}\ E_{\{a\}} \\
E & \longrightarrow & N \\
E & \longrightarrow & \text{'('}\ E\ \text{')'} \\
\hline
E_{\{a\}} & \longrightarrow & \text{'['}\ E_{l1}\ \text{']'} \\
E_{\{a\}} & \longrightarrow & E_{\{a\}}\ \text{'*'}\ E_{\{a\}} \\
E_{\{a\}} & \longrightarrow & N \\
E_{\{a\}} & \longrightarrow & \text{'('}\ E\ \text{')'} \\
\hline
E_{l1} & \longrightarrow & \epsilon \\
E_{l1} & \longrightarrow & E\ E_{l2} \\
\hline
E_{l2} & \longrightarrow & \epsilon \\
E_{l2} & \longrightarrow & \text{';'}\ E\ E_{l2}
\end{array}
$$

(d) G'_D, the generated concrete syntax.

Fig. 4. The generated grammars.

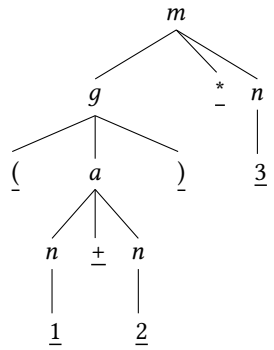
the primed grammars get a new non-terminal per distinctly marked non-terminal in D, where each new non-terminal only has the productions whose label is not in the mark. For example, the non-terminal $E_{\{a\}}$ in Fig. 4b has no production corresponding to the a production in Fig. 3.

Examples of elements in each of these four languages can be seen in Fig. 5, along with visualizations of the syntax trees. Each element corresponds to the word "$(1 + 2) * 3$" in $L(G'_D)$. Note that



$m(a(n(\underline{1}) \underline{+} n(\underline{2})) \underline{*} n(\underline{3}))$

(a) Example tree in $L(T_D)$ and visualization.



$m(g(\underline{(}\, a(n(\underline{1}) \underline{+} n(\underline{2}))\, \underline{)}) \underline{*} n(\underline{3}))$

(b) Example tree in $L(T'_D)$ and visualization.

$1 + 2 * 3$

(c) Example word in $L(G_D)$.

$(1 + 2) * 3$

(d) Example word in $L(G'_D)$.

Fig. 5. Example with elements from each generated language that correspond to each other. The leaf terminals in the tree languages appear underlined to distinguish the two kinds of parentheses.

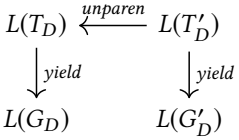


Fig. 6. The generated grammars, and their relation to each other.

the word in $L(G_D)$ is ambiguous, and that there are other words in $L(G'_D)$ that correspond to the same element in $L(T_D)$, e.g., "((1 + 2)) ∗ 3" and "(1 + 2) ∗ (3)". As a memory aid, the primed versions (G'_D and T'_D) contain disambiguation (grouping parentheses, precedence, associativity, etc.) while the unprimed versions (G_D and T_D) are the (likely ambiguous) straightforward translations (i.e., ignoring marks) from D.

At this point we also note that the shape of D determines where the final concrete syntax permits grouping parentheses; they are allowed exactly where they would surround a complete production. For example, G_D in Fig. 4c can be seen as a valid language definition (if we generate new unique labels for each of the productions). However, starting with that language definition would allow the expression "[1(; 2)]", which makes no intuitive sense; grouping parentheses should only be allowed around complete expressions, but "; 2" is not a valid expression.

Finally, we require a function $unparen : L(T'_D) \to L(T_D)$ that removes grouping parentheses from a parse tree, i.e., it replaces every subtree $g(\ '('\ t\ ')'\ )$ with t. The relation between the four grammars in terms of *yield* and *unparen* can be seen in Fig. 6. With this we can define $parse : L(G'_D) \to 2^{L(T_D)}$, along with its inverse $words : L(T_D) \to 2^{L(G'_D)}$:

$$\begin{aligned} parse(w) &= \{unparen(t) \mid t \in L(T'_D) \wedge yield(t) = w\} \\ words(t) &= \{w \mid t \in parse(w)\} \end{aligned}$$

The latter is mostly useful in later sections, but *parse* allows us to consider some concrete examples of resolvable and unresolvable ambiguities. For example, in our running example (Fig. 3), the word `'1 + 2 + 3'` is ambiguous, since $parse(\texttt{'1 + 2 + 3'}) = \{t_1, t_2\}$ where

$$t_1 = a(\quad n(\texttt{'1'})\ \texttt{'+'}\ a(\ n(\texttt{'2'})\quad \texttt{'+'}\ n(\texttt{'3'})\ )\ )$$
$$t_2 = a(\ a(\ n(\texttt{'1'})\ \texttt{'+'}\quad n(\texttt{'2'})\ )\ \texttt{'+'}\ n(\texttt{'3'})\quad)$$

This is a resolvable ambiguity, since $parse(\texttt{'1 + (2 + 3)'}) = \{t_1\}$ and $parse(\texttt{'(1 + 2) + 3'}) = \{t_2\}$. To demonstrate the unresolvable case, we add the production $E \longrightarrow s : E\ \texttt{';'}\ E$, at which point we find that the word `'[1 ; 2]'` is unresolvably ambiguous; $parse(\texttt{'[1 ; 2]'}) = \{t_3, t_4\}$ where:

$$t_3 = l(\ \texttt{'['}\quad n(\texttt{'1'})\ \texttt{';'}\ n(\texttt{'2'})\quad \texttt{']'}\ )$$
$$t_4 = l(\ \texttt{'['}\ s(\ n(\texttt{'1'})\ \texttt{';'}\ n(\texttt{'2'})\ )\ \texttt{']'}\ )$$

In this case, t_4 has an unambigous word (namely `'[(1 ; 2)]'`), but t_3 does not. The solution is to forbid list elements from being sequences by modifying the language definition in Fig. 3 so that both non-terminals in the production l are marked with s (i.e., they look like $E_{\{s\}}$), at which point $parse(\texttt{'[1 ; 2]'}) = \{t_3\}$.

We are now ready to construct decision procedures for the static and dynamic resolvability problems, as given in Definitions 4.6, 4.7, 4.8, and 4.9. Section 6 gives a partial solution to the static problem, while Section 7 fully solves the dynamic problem with one caveat: we only consider languages with balanced parentheses.

6 STATIC RESOLVABILITY ANALYSIS

For this section, we will use an alternative formulation of resolvable ambiguity for languages, stated in terms of parse trees instead of words:

Theorem 6.1. *Given an abstract parser $P = (\Sigma, T, parse)$, the language $L(P)$ is resolvably ambiguous iff* $\forall t \in T.\ \exists w \in \Sigma^*.\ parse(w) = \{t\}$.

Proof. The quantifier $\forall t \in T$ is equivalent with $\forall w \in \Sigma^*.\forall t \in parse(w)$, since $T = \bigcup_{w \in \Sigma^*} parse(w)$ (by definition). □

To determine if a given language definition D is resolvably ambiguous we attempt to find a counterexample: a tree $t \in L(T_D)$ such that there is no $w \in L(G'_D)$ for which $parse(w) = \{t\}$, or prove that no such tree exists. Or, more briefly put: find a tree that has only ambiguous words or show that no such tree exists.

Before an actual algorithm, we split the problem into three different versions of increasing difficulty. We then lay the groundwork for our correctness proofs, eventually culminating in the algorithm, which we show to be sound and complete for version 1 and sound for version 2.

The three versions are as follows:

Version 1 $\Sigma \cap \{\ \texttt{'('}, \texttt{')'}\ \} = \emptyset$ and no non-terminals appear marked, i.e., for all non-terminals N_m appearing on the right-hand side of the labelled productions in D, we have $m = \emptyset$.

Version 2 $\Sigma \cap \{\ \texttt{'('}, \texttt{')'}\ \} = \emptyset$.

Version 3 There are no constraints on D.

The first restriction in version 1 states that D cannot contain parentheses. This implies that all parentheses present in G'_D are grouping parentheses. The second restriction implies that no parentheses are required (as a counterexample, assuming normal precedence, the parentheses in "$(1 + 2) * 3$" are required and removing them would produce a different interpretation).

The reason for this split stems from the following insight: double grouping parentheses do not matter, in the sense that they do not change the interpretation of the word, e.g., $parse(\texttt{'((1 + 2)) * 3'}) = parse(\texttt{'(1 + 2) * 3'})$. Versions 1 and 2 have only grouping parentheses, meaning that no double

parentheses matter. What follows is a brief outline of the remainder of this section, which details our static analysis for versions 1 and 2, building from that observation.

- We can consider an alternate representation of words where parentheses are not explicitly part of the string of terminals, but are represented by an accompanying bag of ranges denoting which terminals are covered by parentheses. (Section 6.1).
- This alternate representation gives rise to a lattice per tree in $L(T_D)$, where a tree has only ambiguous words iff its lattice is entirely covered by the lattices of other trees (Section 6.2).
- These lattices can be encoded as words, leading to the construction of a visibly pushdown automaton that we can examine to determine the existence of a tree whose lattice is covered by the lattice of a *single* other tree, which is sufficient for completeness in version 1 (Section 6.3).

6.1 An Alternative Word View

This section introduces an alternative (isomorphic) definition of a word, heavily used in sections 6.2 and 6.3. The method by which we generate G_D and G'_D limits the possible differences between them significantly. In particular, no new terminals are introduced, except '(' and ')', and they are always introduced in a well-balanced fashion.

If we thus delimit ourselves to only consider languages where words have no unbalanced parentheses[3] we can give the following alternative definition of a word: a word is a two-tuple containing a sequence of non-parenthesis terminals and a bag (or multiset) of ranges covered by parentheses. For example, the word "`(1 + 2) * 3`" is equivalent to ("`1 + 2 * 3`", $\{0{-}3\}$), while "`((1 + 2)) * 3`" is equivalent to ("`1 + 2 * 3`", $\{0{-}3, 0{-}3\}$). We will refer to the first component of the tuple as the *basic word* and the second as the *range bag*.

However, there is a case where this alternate representation is not fully isomorphic; namely when a pair of parentheses would yield a zero-length range. As an example, consider the two words "`(())`" and "`()()`". Both of these have an alternate representation of ("", $\{0{-}0, 0{-}0\}$). In versions 1 and 2 this corresponds to grouping parentheses surrounding zero-length productions. We can avoid this case by finding every production $N \longrightarrow l : r$ where $\epsilon \in L(r)$, replacing it with $N \longrightarrow l : r'$ where $L(r') = L(r) \setminus \{\epsilon\}$, and replacing every use of N with $N + \epsilon$. The one exception is if the start symbol has a nullable production, but that can be trivially special cased. Thus, we will assume no zero-length grouping parentheses in the following.

We denote the functions producing the two components by *basic* and *rangebag*, i.e., *basic*("`((1 + 2)) * 3`") = "`1 + 2 * 3`" and *rangebag*("`((1 + 2)) * 3`") = $\{0{-}3, 0{-}3\}$.

The next four lemmas, all concerning the words in $words(t)$ for some given $t \in L(G_t)$, form the basis of the lattices mentioned at the end of the previous section.

Lemma 6.2. $\forall w_1, w_2 \in words(t).\ basic(w_1) = basic(w_2)$.

More informally, all $w \in words(t)$ have the same basic word. Intuitively, *words* first applies the "inverse" of *unparen*, i.e., adding some number of grouping parentheses nodes between pre-existing nodes, then *yield*, which flattens the tree to a word. The only terminals changed in this process are parentheses, i.e., different w can only differ in their range bags. This also implies that two trees that share a word (i.e., trees t_1 and t_2 that have a word w such that $parse(w) \supseteq \{t_1, t_2\}$) must also share a basic word.

Lemma 6.3. *There exists a set* R *such that* $\exists w \in words(t).\ rangebag(w) = R$ *and* $\forall w \in words(t).\ R \subseteq Supp(rangebag(w))$

Recall that $Supp(B)$ is the support of the bag B, i.e., the underlying set of elements in B. More informally, some parentheses are required. For example, removing the parentheses in "`(1 + 2) * 3`"

[3] I.e., the vast, vast majority of programming languages currently in use.

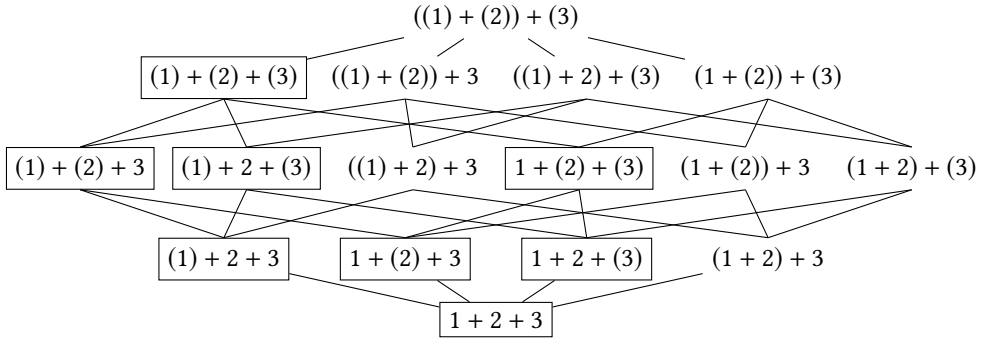


Fig. 7. The lattice of words for the tree $a(a(n(1) + n(2)) + n(3))$. The boxed words are shared with the lattice in Fig. 8.

changes the produced parse trees. Each required range is a direct consequence of a mark on a non-terminal in D. We will write $\bot_w(t)$ for the word implied by the first quantified expression. It is unique, since its basic word is fixed by t, and its rangebag is a set.

Lemma 6.4. *There exists a set R such that $\exists w \in words(t).\ rangebag(w) = R$ and $\forall w \in words(t).\ Supp(rangebag(w)) \subseteq R$.*

More informally, there is a finite set of possible parentheses. As mentioned in Section 5, grouping parentheses can only be added if they exactly cover a production, which amounts to an internal node in t, and each tree has a finite amount of nodes. We will write $\top_w(t)$ for the word implied by the first quantified expression, which is also unique.

Lemma 6.5. $\forall w_1, w_2 \in words(t).\ Supp(rangebag(w_1)) = Supp(rangebag(w_2)) \implies parse(w_1) = parse(w_2)$.

More informally, duplicated parentheses do not matter. We first note that in version 1 and 2 there are no parentheses in D, i.e., all parentheses in G'_D are introduced as grouping parentheses. Each pair of parentheses thus corresponds to a single grouping node g in a tree in $L(T'_D)$. Duplicated grouping parentheses correspond to nested grouping nodes, all of which are removed by *unparen*, thus producing identical sets of trees in $L(T_D)$. For example, `((1 + 2)) * 3` has the same interpretations as `(1 + 2) * 3`.

6.2 A Lattice of Word Partitions

Lemma 6.5 suggests a partition of the words in $words(t)$ for any given $t \in L(T_D)$; group words w by their *rangeset*, where $rangeset(w) = Supp(rangebag(w))$. These partitions can be partially ordered by subset on the rangeset, resulting in a lattice of word partitions per tree. This lattice is bounded, with top and bottom elements $\top_w(t)$ and $\bot_w(t)$. We denote this lattice by $lattice(t)$. Lemma 6.2 further states that all words in $words(t)$ share the same basic word. For example, Fig. 7 contains the lattice for the tree $a(a(n(1) + n(2)) + n(3))$ (from our running example, defined in Fig. 3 on page 9). Each partition is represented by the word whose rangebag is a set. To reduce clutter, we do not draw the partitions that have grouping parentheses around the entire word. This outermost possible pair is uninteresting since it is always allowed, and would double the size of the figure if it was included.

To show the connection between resolvable ambiguity and these lattices, consider the word "`1 + 2 + 3`". It has two interpretations in our running example, $a(a(n(1) + n(2)) + n(3))$, which we

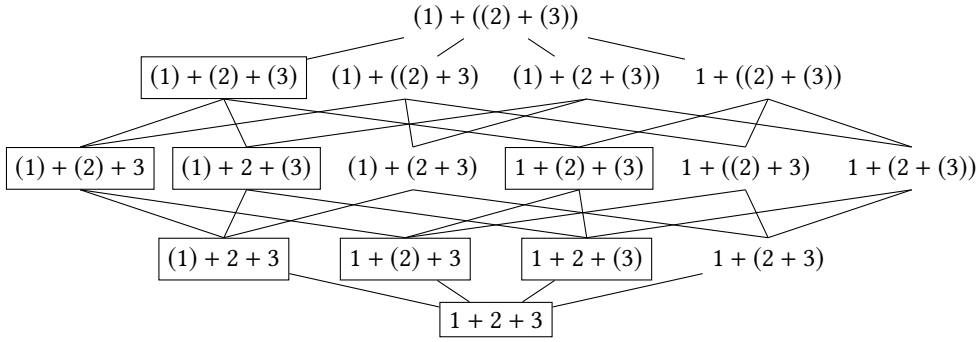


Fig. 8. The lattice of words for the tree $a(n(1) + a(n(2) + n(3)))$. The boxed words are shared with the lattice in Fig. 7.

would normally write as `(1 + 2) + 3`, and $a(n(1) + a(n(2) + n(3)))$, which we would normally write `1 + (2 + 3)`. The lattices for these two trees are given in Figures 7 and 8 respectively. The partitions that appear in both lattices are represented as boxed words, the others are unboxed. These shared partitions represent words that are ambiguous between these particular trees. Finding an unambiguous word is thus the same as finding a partition that is not shared with any other tree. In this particular case, there is no ambiguity with any other tree at all, and so the unboxed words are all valid resolutions of the ambiguity.

At this point it is clear that a given tree has an unambiguous word iff its lattice has at least one partition that is not shared with any other tree.

Next, we note that each lattice is uniquely determined by its top and bottom elements; it contains all elements between them:

Lemma 6.6. *Given $t \in L(T_D)$, we have that:*
$\forall w.\ rangebag(\bot_w(t)) \subseteq rangeset(w) \subseteq rangebag(\top_w(t)) \implies w \in words(t)$.

Recall that $rangebag = rangeset$ for $\bot_w(t)$ and $\top_w(t)$. Finally, we show the difference between versions 1 and 2 in the lattice setting: in version 1, the rangeset of the bottom word is the empty set:

Lemma 6.7. *In version 1, $\forall t \in L(T_D).\ rangeset(\bot_w(t)) = \emptyset$.*

Since version 1 has no markings there are no required parentheses, thus the bottom word has no parentheses.

Our approach centers around finding a pair of lattices, such that one is entirely contained in the other. Since the former shares *all* partitions with the latter, it has only ambiguous words, thus any of those words will be unresolvably ambiguous.

Theorem 6.8. *In versions 1 and 2, given a $t \in L(T_D)$, $w_\bot = \bot_w(t)$, and $w_\top = \top_w(t)$, the existence of a tree $t' \in L(T_D)$ such that $basic(w_\bot) = basic(\bot_w(t')) \wedge rangeset(w_\bot) \subseteq rangeset(\bot_w(t')) \wedge rangeset(\top_w(t')) \subseteq rangeset(w_\top)$ implies $\neg\exists w.\ parse(w) = \{t\}$.*

For version 1, *every* unresolvable ambiguity has this form:

Theorem 6.9. *In version 1, given a $t \in L(T_D)$, $w_\bot = \bot_w(t)$, and $w_\top = \top_w(t)$, $\exists t' \in L(T_D).\ basic(w_\bot) = basic(\bot_w(t')) \wedge rangeset(\bot_w(t')) \subseteq rangeset(w_\bot) \wedge rangeset(w_\top) \subseteq rangeset(\top_w(t')) \iff \neg\exists w.\ parse(w) = \{t\}$.*

Proof. The left-to-right implication is already given in Theorem 6.8, and Lemma 6.7 together with $basic(w_\bot) = basic(\bot_w(t'))$ implies that $w_\bot = \bot_w(t')$. We thus need to show that $\neg\exists w.\ parse(w) = \{t\} \implies \exists t' \in L(T_D).\ basic(w_\bot) = basic(\bot_w(t')) \wedge rangeset(\top_w(t')) \subseteq rangeset(w_\top)$.

If all words in $words(t)$ are ambiguous, then $\exists t' \in L(T_D).\ t' \neq t \wedge t' \in parse(w_\top)$. Lemma 6.2 gives $basic(w_\bot) = basic(\bot_w(t'))$. The definition of $\top_w(t')$ (Lemma 6.4) gives $rangeset(w_\top) \subseteq rangebag(\top_w(t'))$, but since the rangebag of $\top_w(t')$ is a set, this also implies that $rangeset(w_\top) \subseteq rangeset(\top_w(t'))$. □

6.3 A Lattice as a Word, and an Algorithm

This section introduces a linear encoding of the lattices of the previous section as words. Lemmas 6.2 and 6.6 imply that a lattice encoding requires three things: a basic word, a set of *required* parentheses, and a set of *possible* parentheses. We encode required parentheses with "()" and possible parentheses with "()". For example, the lattice in Fig. 8 is represented by "((1) + ((2) + (3)))", while the tree for "(1 + 2) ∗ 3" has a lattice represented by "(((1) + (2)) ∗ (3))". We denote the lattice encoded by such a word by $lattice(w)$. With this encoding we can apply the formidable body of knowledge available for word languages, albeit with some extra care; the linear encoding of a lattice is not unique.

For example, "((1))" and "(1)" represent the same lattice, as do "((2))" and "(2)". To gain uniqueness we forbid duplicated parentheses and prioritize required parentheses over possible parentheses. For example, "((1))" is uniquely represented as "(1)" while "((2))" is uniquely represented as "(2)". We define a linear encoding of this form to be *well-formed*.

Lattice equality is now equivalent with equality of the linear encoding. To determine if a lattice is entirely contained in another, we make the following observation: we can move the top of the lattice "up" (new top rangeset is a superset) by adding a pair of possible parentheses, and move the bottom "down" (new bottom rangeset is a subset) by replacing a required pair with a possible pair. For example, "(1)2" is entirely contained in "(1)(2)", which is entirely contained in "(1)(2)".

The centerpiece of our algorithm is a *visibly pushdown automaton* constructed in such a way that

- there is a bijection between trees in $L(T_D)$ and successful runs; and
- the word recognized by a successful run is the linear encoding of the corresponding tree's lattice.

Two distinct successful runs through this automaton that recognize the same word thus imply two distinct trees that have exactly the same lattice. To detect lattices contained in each other, we create a modified copy, where the copy may add arbitrary (balanced, well-nested) possible parentheses, and replace any required pair of parentheses with a possible pair. The copy maintains much of the structure of the original, and in particular, keeps the connection to trees in $L(T_D)$ (though it is no longer a bijection: there are multiple successful runs per tree since there are multiple larger lattices). Two distinct successful runs, one in each automaton, that recognize the same word then imply two distinct trees where the lattice of one is entirely contained in the other.

6.3.1 Example for Simplified Construction. We will now walk through the construction of this automaton, using the following language definition:

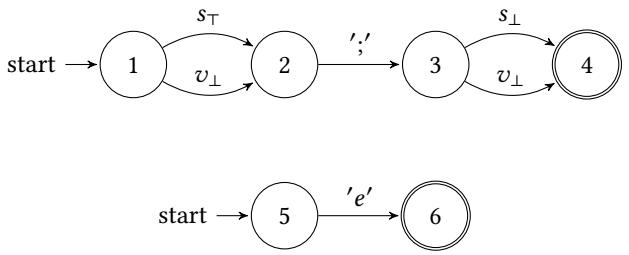
$$\begin{array}{lcll} E & \rightarrow & s: & E_{\{s\}}\ \texttt{';'}\ E \\ E & \rightarrow & v: & \texttt{'e'} \end{array}$$

We first present a simplified construction that assumes no production can match a single non-terminal, i.e., for all right-hand side regular expressions r, $\neg\exists N_m.\ N_m \in L(r)$, to make the base idea easier to follow. Such cases introduce duplicated parentheses if handled in a naive way, and the extra book-keeping required to correctly handle them complicate this presentation, and will so be introduced after the more naive method.

Conceptually, each non-terminal represents a choice of which production should replace it, and whether that production requires parentheses around it. We now make this explicit, to ensure that standard operations on finite automata produce the correct result. We replace each N_m with a sum of the labels of productions in N, where each label has a subscript $\top$ or $\bot$ if parentheses are required or possible, respectively. For example, $E_{\{s\}}$ is replaced with $s_\top + v_\bot$.

E	$\rightarrow$	s :	$(s_\top + v_\bot)$ ';' $(s_\bot + v_\bot)$
E	$\rightarrow$	v :	'e'

Next we construct a DFA per production. This can be done in the standard way by constructing an NFA, then determinizing it, and optionally minimizing it.

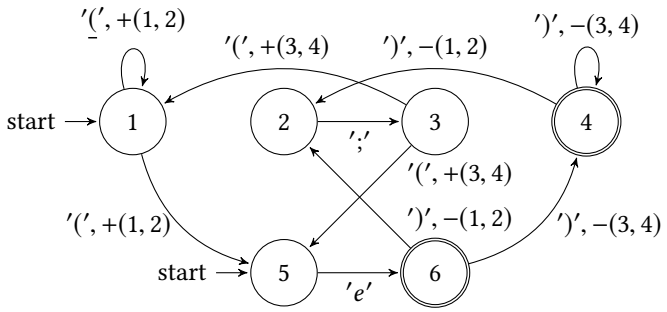


We then combine the DFAs into a single visibly pushdown automaton. Every time we transition from a production to a child we push a symbol to the stack, which we then pop when we return. We use $Q \times Q$ as our stack alphabet, to record the transition we are replacing in the parent. We thus replace every edge $p \xrightarrow{l_\top} q$, where $l \in L$, with:

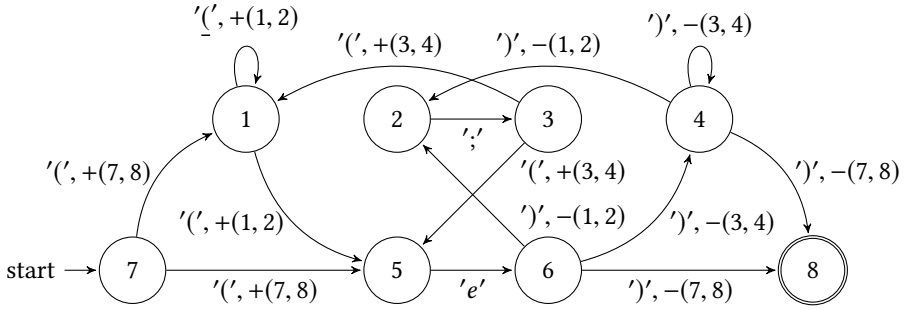
- an edge $p \xrightarrow{'(',+(p,q)} p'$, where p' is the initial state in the DFA corresponding to the production with label l, and
- an edge $q' \xrightarrow{')',-(p,q)} q$ for every final state q' in the DFA corresponding to the production with label l.

Similarly, for every edge $p \xrightarrow{l_\bot} q$, where $l \in L$, we add edges $p \xrightarrow{'(',+(p,q)} p'$ and $q' \xrightarrow{')',-(p,q)} q$. Note the difference: for $l_\top$ we add a required parenthesis "(", while $l_\bot$ adds a possible parenthesis "(".

Intuitively, we parse a child node with parentheses around it, then return. This is where our simplification is used, without it we might introduce double parentheses here.

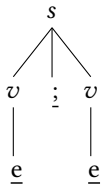


Finally, we add a new initial state and a new final state, and connect them to the initial and final states belonging to the starting non-terminal:



For example, the linearly encoded lattice $((e);(e))$ is recognized by the run

$$
\begin{aligned}
(7,\epsilon) &\xrightarrow{(} (1, && (7,8)) \xrightarrow{(} (5, (1,2)\cdot(7,8)) \xrightarrow{e} (6, (1,2)\cdot(7,8))\\
&\xrightarrow{)} (2, && (7,8)) \xrightarrow{;} (3, && (7,8)) \xrightarrow{(} (5, (3,4)\cdot(7,8))\\
&\xrightarrow{e} (6, (3,4)\cdot(7,8)) \xrightarrow{)} (4, && (7,8)) \xrightarrow{)} (8, && \epsilon)
\end{aligned}
$$

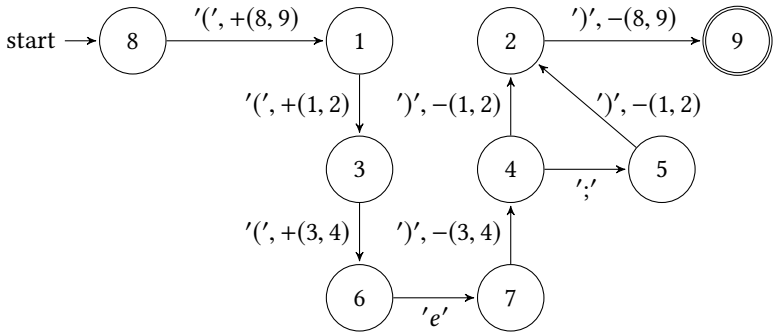


and corresponds to the tree on the right.

6.3.2 Example for Full Construction. We now illustrate the issue that our previous simplification avoids, and then modify the construction such that the simplification is no longer required. Consider the following language definition:

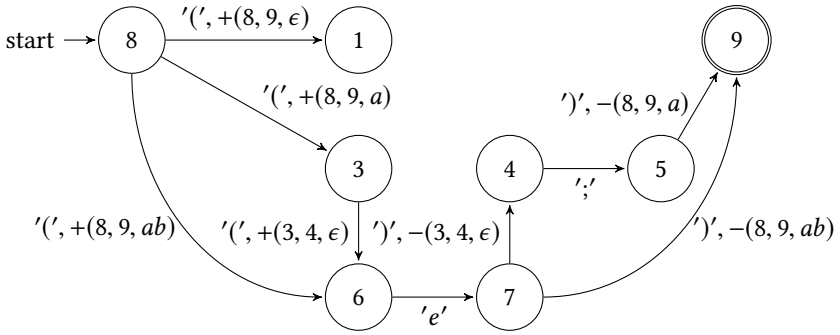
A	$\longrightarrow$	$a:$	B
B	$\longrightarrow$	$b:$	C (' ; ' + ϵ)
C	$\longrightarrow$	$c:$	'e'

Following the simplified construction we produce the following automaton:



Unfortunately, this automaton recognizes the word $(((e)))$, which is not the unique linear encoding we require; it should be (e). The reason for this mismatch is that each of the productions with labels a and b introduces an additional pair of parentheses in the word recognized by the automaton. To ensure that the automaton instead recognizes the required unique linear encoding (e), we extend the construction of the automaton such that "passing through" a non-terminal without recognizing any terminals does *not* add parentheses. The idea is to skip the DFAs of such non-terminals, but still keep track of the skipped production(s) in the symbols pushed to the stack; thus, runs which differ in skipped non-terminals are still distinguished.

For our example, the resulting automaton looks as follows:



Note that the transition from state 8 to state 6 pushes the symbol $(8, 9, ab)$ to the stack. This means (a) that we started the "replacement" for the transition from 8 to 9, and (b) that we passed through the productions with labels a and b without consuming terminals, thus ending up in the starting state of the DFA corresponding to production c.

6.3.3 Formalization. Given a language definition D, with terminals Σ, non-terminals V, labels L, and starting non-terminal S. We use $nt(l)$ and $regex(l)$ as the non-terminal and regex, respectively, of the production labelled l. We define the set $T : L \times L \times \{\top, \bot\} \times L^*$ of transitions between productions inductively through:

- $(l, l, \bot, \epsilon) \in T$ for all $l \in L$.
- $(l, l', l' \in m, l) \in T$ for all $N_m \in L(regex(l))$ where $nt(l') = N$.
- $(l_1, l_3, r_1 \vee r_2, w_1 \cdot w_2) \in T$ for all $(l_1, l_2, r_1, w_1), (l_2, l_3, r_2, w_2) \in T$.

The third component of each element denotes whether that transition introduces a required pair of parentheses ($\top$) or a possible pair ($\bot$). T is finite if D has no cycles $N \Rightarrow^+ N$[4]. Such a cycle is easy to detect (depth-first search), and means that any tree that contains a production from the non-terminal N only ever produces infinitely ambiguous words, i.e., all words for such a tree are unresolvably ambiguous.

We define the translation from production to DFA through a function *dfa* from a label to a DFA with the alphabet $\Sigma \cup (L \times \{\top, \bot\})$:

dfa(l): Compute the regular language $L' := L(regex(l)) \setminus \{N'_m \mid N' \in V, m \subseteq L\}$. Then replace every marked non-terminal N_m with the regular expression $(l_1, l_1 \in m) + (l_2, l_2 \in m) + \ldots + (l_n, l_n \in m)$, where l_i are all the labels of productions $N \longrightarrow l_i : r_i$. For example, $E_{\{a\}}$ in the running example would be replaced by $(l, \bot) + (a, \top) + (m, \bot) + (n, \bot)$. Then construct a DFA for this language.

We now define the visibly pushdown automaton $A_{()}$. Given $dfa(l) = (Q_l, \Sigma \cup (L \times \{\top, \bot\}), \delta_l, s_l, F_l)$ for all $l \in Labels$, $A_{()} = (Q, \Sigma', \Gamma, \delta, s, \{f\})$ where:

- s and f are two new distinct states.
- $Q = \{s, f\} \cup \bigsqcup_l Q_l$.
- $\Sigma' = \Sigma \cup \{(,)\}$.
- $\Gamma = Q \times Q \times L^*$.

[4]This is notation from CFGs, which is semi-obvious what it means, but technically not defined here.

- δ is defined by the following equations (unspecified cases produce $\emptyset$):

$$\begin{aligned}
\delta(p, a, \lambda) &= \{(\delta_l(p, a), \lambda)\} && \text{where } p \in Q_l \text{ and } a \in \Sigma \\
\delta(p, \text{“(”}, \lambda) &= \{(s_{l_3}, (p, q, w)) \mid p \in Q_{l_1}, \delta_{l_1}(p, (l_2, r_1)) = q, (l_2, l_3, r_2, w) \in P, r_1 \vee r_2\} \\
\delta(p, \text{“(”}, \lambda) &= \{(s_{l_3}, (p, q, w)) \mid p \in Q_{l_1}, \delta_{l_1}(p, (l_2, r_1)) = q, (l_2, l_3, r_2, w) \in P, \neg(r_1 \vee r_2)\} \\
\delta(f_{l_3}, \text{“)”}, (p, q, w)) &= \{(q, \lambda) && \mid p \in Q_{l_1}, \delta_{l_1}(p, (l_2, r_1)) = q, (l_2, l_3, r_2, w) \in P, f_{l_3} \in F_{l_3}\} \\
\delta(s, \text{“(”}, \lambda) &= \{(s_{l_2}, (s, f, w)) \mid nt(l_1) = S, (l_1, l_2, \top, w) \in P\} \\
\delta(s, \text{“(”}, \lambda) &= \{(s_{l_2}, (s, f, w)) \mid nt(l_1) = S, (l_1, l_2, \bot, w) \in P\} \\
\delta(f_{l_2}, \text{“)”}, (s, f, w)) &= \{(f, \lambda) && \mid nt(l_1) = S, (l_1, l_2, r, w) \in P, f_{l_2} \in F_{l_2}\}
\end{aligned}$$

We also construct a modified VPDA $A'_{()} = (Q, \Sigma', \Gamma \cup \{\gamma\}, \delta', s, \{f\})$ where:

- γ is a new distinct stack symbol.
- $\delta'(p, a, g) = \delta(p, a, g) \cup \delta''(p, a, g)$, where δ'' is given by:

$$\begin{aligned}
\delta''(p, \text{“(”}, \lambda) &= \{(p, \gamma)\} \\
\delta''(p, \text{“)”}, \gamma) &= \{(p, \lambda)\} \\
\delta''(p, \text{“(”}, \lambda) &= \delta(p, \text{“}\underline{(}\text{”}, \lambda)
\end{aligned}$$

To find two distinct successful runs, one in each automaton, that recognize the same word we construct the product automaton $A_{()} \times A'_{()} = (Q \times Q, \Sigma', \Gamma \times (\Gamma \cup \{\gamma\}), \delta_\times, (s, s), \{(f, f)\})$, where $\delta_\times$ is given in Section 3.3 (using the partitions $\Sigma_c = \{\underline{(}, ()\}$, $\Sigma_i = \Sigma$, and $\Sigma_r = \{)\}$). If the product automaton has a successful run that passes through at least one configuration $((p, p'), (g, g') \cdot w)$ such that $p \neq p' \vee g \neq g'$ (distinct states, or distinct stack symbols, respectively), then that run corresponds to two distinct runs in $A_{()}$ and $A'_{()}$. We check for the existence of such a run by trimming the product automaton and looking for a transition that pushes (g, g') where $g \neq g'$, or transitions to (q, q') where $q \neq q'$.

Performing this procedure directly on a language definition D gives a sound decision procedure where the presence of two distinct runs implies unresolvably ambiguous. If we instead do it on a modified D with no marks on the right hand side of any production (i.e., all non-terminals have the form $N_\emptyset$) we get a sound decision procedure where the absence of two distinct runs implies resolvably ambiguous.

There are thus cases where we can answer resolvably ambiguous or unresolvably ambiguous with certainty, and cases where we cannot answer either way with certainty.

Lemma 6.10. *There is a bijection between successful runs in $A_{()}$ and trees $t \in L(T_D)$.*

Proof. By defining two functions *run* and *tree*, the former from a tree to a run, the latter from run to tree, then showing these two functions to be inverses of each other. □

Lemma 6.11. *All words $w \in L(A_{()})$ are well-formed linear encodings.*

We denote the word recognized by a run R by $word(R)$.

Lemma 6.12. $\forall t \in L(T_D).\ lattice(word(run(t))) = lattice(t)$.

Lemma 6.13. *For all $t \in L(T_D)$ and every well-formed lattice encoding w such that $lattice(t) \subseteq lattice(w)$, $w \in L(A'_{()})$.*

Lemma 6.14. *We can define a function normalrun from runs in $A'_{()}$ to runs in $A_{()}$ such that $\forall R.\ lattice(word(normalrun(R))) \subseteq lattice(word(R))$.*

Proof. By removing every transition that pushes or pops γ and replacing every used transition $p \xrightarrow{\text{'(',+g}} q$ that does not exist in $A_{()}$ with a transition $p \xrightarrow{\text{'}\underline{(}\text{',+g}} q$. □

Theorem 6.15. *The existence of two distinct runs R and R', where R is a successful run in $A_{()}$ and R' in $A'_{()}$, and $word(R) = word(R')$, implies the existence of two distinct trees such that the lattice of one is entirely contained in the other.*

Proof. Assume $normalrun(R') = R$. *normalrun* either leaves its argument unchanged, or changes the recognized word. If R' and R recognize the same word, and $normalrun(R') = R$, then $R' = R$, but that is a contradiction.

Since $normalrun(R') \neq R$ we have two distinct trees $t_1 = tree(normalrun(R'))$ and $t_2 = tree(R)$. Since $lattice(word(normalrun(R'))) \subseteq lattice(word(R'))$ and $lattice(word(R')) = lattice(word(R))$ we have that $lattice(t_1) \subseteq lattice(t_2)$. □

7 DYNAMIC RESOLVABILITY ANALYSIS

The dynamic resolvability problem is as follows: for a given $w' \in L(G'_D)$ determine whether $\forall t \in parse(w').\ \exists w'_2.\ parse(w'_2) = \{t\}$. Furthermore, for practical reasons, if the word is resolvably ambiguous we wish to produce a (minimal) witness for each tree. Similarly to Section 6, we place some restrictions on D, though the restrictions are less severe. While Section 6 requires a language definition D to not contain parentheses, this section merely requires parentheses to be balanced. Concretely, the restrictions are as follows:

(1) Each right-hand side regular expression must only recognize words with balanced parentheses. For example, "`()`" is permissible, but "`(*)*`" is not.
(2) G_D, but with parentheses removed, must not be infinitely ambiguous.

Both of these restrictions can be checked statically.

Our approach centers around $words(t)$, which, being a set of words, can be considered a language in the classical sense. Given restriction (1), each such language turns out to be a visibly pushdown language. VPLs are closed under difference, which enables us to create a visibly pushdown automaton for the language $words(t_1) \setminus words(t_2)$ for any pair of trees $t_1, t_2 \in L(T_D)$. This is useful, since $words(t_1)$ is the set of possible resolutions for t_1 in an ambiguity, and $words(t_1) \cap words(t_2)$ is the set of ambiguous words that can parse as both t_1 and t_2, i.e., words that *cannot* resolve an ambiguity.

Intuitively, we then need to, for each tree $t \in parse(w)$, determine if $words(t) \setminus \bigcup_{t' \in L(T_D) \wedge t' \neq t} words(t)$ is the empty set. Unfortunately, $L(T_D)$ tends to be infinite. However, for any particular t, most trees $t' \in L(T_D)$ are irrelevant: their words do not overlap at all; $words(t) \cap words(t') = \emptyset$. Hence we construct a (finite) subset of $L(T_D)$ such that all other trees are irrelevant. As a bonus, this approach constructs a VPDA per tree that recognizes valid resolutions for that tree; a minimal witness is thus a shortest word recognized by a VPDA, and the problem of finding such a word is a known one.

For this section, we assume the presence of a single language definition D with terminals in Σ, labels in L, and non-terminals in V.

Lemma 7.1. *Given a $t \in L(T_D)$, we can construct a visibly pushdown automaton that accepts exactly $words(t)$.*

We construct this automaton in a bottom-up fashion. A sequence of terminals in Σ is easily recognized by a sequence of transitions that do not interact with the stack. Each subtree allows surrounding grouping parentheses, which can be represented by transitions that push and pop stack symbols (one unique stack symbol per subtree), to ensure that they are balanced. With that, there are two remaining complications: a subtree $l(w)$ recognized from a marked non-terminal N_m where $l \in m$ must have one or more surrounding parentheses instead of zero or more, and we must maintain the visibly pushdown property, i.e., all transitions labelled "(" or ")" must push or pop stack symbols, respectively. The former can be solved by either tracking the subtree's origin when parsing, or reparsing with the right hand side regular expression, while the latter is solved

by introducing a new stack symbol γ that is pushed and popped by all non-grouping-parenthesis terminals.

We now present the function $f : L(T_D) \to VPDA$ that performs this construction. Writing the transition function $\delta : Q \times \Sigma \times (\Gamma \cup \{\lambda\}) \to 2^{Q \times (\Gamma \cup \{\lambda\})}$ as a subset of $Q \times \Sigma \times (\Gamma \cup \{\lambda\}) \times Q \times (\Gamma \cup \{\lambda\})$, and using $\sqcup$ for disjoint union:

$$f(a) = \begin{cases} (\{p,q\}, \Sigma, \emptyset, \{(p,a,\lambda,q,\lambda)\}, p, \{q\}) & \text{if } a \notin \{\text{“(”}, \text{“)”}\} \\ (\{p,q\}, \Sigma, \{\gamma\}, \{(p,\text{“(”},\lambda,q,\gamma)\}, p, \{q\}) & \text{if } a = \text{“(”} \\ (\{p,q\}, \Sigma, \{\gamma\}, \{(p,\text{“)”},\lambda,q,\lambda))\}, p, \{q\}) & \text{if a = “)”} \end{cases}$$

$$f(w_1 \cdot w_2) = (Q_1 \sqcup Q_2, \Sigma, \Gamma', \delta_1 \cup \delta_2 \cup \{(f_1, \lambda, \lambda, s_2, \lambda)\}, s_1, \{f_2\})$$
$$\text{where } f(w_i) = (Q_i, \Sigma, \Gamma_i, \delta_i, s_i, \{f_i\})$$
$$\Gamma' = (\Gamma_1 \setminus \{\gamma\}) \sqcup (\Gamma_2 \setminus \{\gamma\}) \cup \{\gamma\}$$

$$f(l(w)) = \begin{cases} (Q \sqcup \{p,q\}, \Sigma, \Gamma', \delta \cup \delta_1, p, \{q\}) & \text{if } l \text{ is marked in the introducing non-terminal.} \\ (Q, \Sigma, \Gamma', \delta \cup \delta_2, s, \{f\}) & \text{if } l \text{ is unmarked in the introducing non-terminal.} \end{cases}$$
$$\text{where } f(w) = (Q, \Sigma, \Gamma, \delta, s, \{f\})$$
$$\Gamma' = (\Gamma \setminus \{\gamma\}) \sqcup \{g, \gamma\}$$
$$\delta_1 = \{(p, \text{“(”}, \lambda, p, g), (p, \text{“(”}, \lambda, s, g), (f, \text{“)”}, g, q, \lambda), (q, \text{“)”}, g, q, \lambda)\}$$
$$\delta_2 = \{(s, \text{“(”}, \lambda, s, g), (f, \text{“)”}, g, f, \lambda)\}$$

Lemma 7.2. *Given a finite set $\{t_1, t_2, \ldots, t_n\} \subseteq L(T_D)$ we can, in finite time, construct a set of VPDAs $\{A_1, A_2, \ldots, A_n\}$ such that $\forall i.\ L(A_i) \setminus \bigcup_{i \neq i'} words(t_{i'})$.*

Proof. Follows trivially from Lemma 7.1 and VPDAs being closed under union and difference. □

Lemma 7.3. *Given a finite set $\{t_1, t_2, \ldots, t_n\} \subseteq L(T_D)$, the set of trees $t' \in L(T_D)$ such that $words(t) \cap words(t') \neq \emptyset$ is finite.*

Proof. Lemma 6.2, presented in Section 6.1, holds since restriction (1) implies only well-balanced parentheses, thus all t' must share a basic word with all t_i. G_D with parentheses removed recognizes these basic words, and has a one-to-one correspondence between leftmost derivations and trees in $L(T_D)$, by construction. Restriction (2) states that G_D without parentheses is not infinitely ambiguous, thus there must be a finite number of trees that have the same basic word as all t_i. □

Restriction (2) also implies that G'_D is not infinitely ambiguous, whereby $parse(w)$ must be finite. The dynamic analysis thus parses $basic(w)$ using G_D with parentheses removed to produce a set R of relevant trees. For every $t \in parse(w)$, we construct a VPDA recognizing $words(t) \setminus \bigcup_{t' \in R, t \neq t'} words(t')$. If none of these VPDAs recognize the empty language, then the word is resolvably ambiguous, otherwise it is unresolvably ambiguous.

8 CASE STUDIES

In this section, we showcase two possible use-cases of dynamic resolvability analysis: composing separately defined DSLs (Section 8.1), and finding ambiguities in grammars through testing (Section 8.2). We have implemented a tool and a small DSL for syntax definition that generates a language definition in the style of Section 5, which the tool then uses to construct a parser and do dynamic resolvability analysis. The tool is implemented in Haskell and uses an off-the-shelf implementation[5] of the Earley parsing algorithm [Earley 1970].

The syntax definition DSL builds on *syncons*, introduced by Palmkvist and Broman [2019], wherein each production is defined separately from each other (one **syncon** each). The main

[5] http://hackage.haskell.org/package/Earley

difference between the DSL and the formalism in Section 5 is that marks are introduced as separate **forbid** declarations rather than being inlined in productions. This allows supporting high-level convenience constructs, e.g., specifying precedence between previously defined operators instead of manually inserting marks, but is also important for composability as ambiguities can be statically resolved without changing the original definitions. Other convenience constructs include defining prefix, infix, and postfix operators with a given associativity.

The running example used in Section 5 (Fig. 3 on page 9) can be defined as follows:

```
type Exp                                token Integer = "[0-9]+"
grouping "(" Exp ")"                    syncon literal: Exp = n:Integer
precedence {                            syncon list: Exp =
  mul; // higher in precedence list       "[" (head:Exp (";" tail:Exp)*)? "]"
  add; // means higher precedence       infix add: Exp = "+"
}                                       infix mul: Exp = "*"
```

Here, we define a syntax type Exp for expressions, and declare that parentheses can be used to group expressions. We define a lexical token for integers, and a **syncon** for integer literals using this token. A list is defined as a sequence of zero or more expressions separated by semi-colons wrapped in square braces. Finally, addition and multiplication are defined as infix syncons with the expected precedence rules. Note that we could just as well have replaced the precedence list with explicit **forbid** declarations, "**forbid** `mul.`**left** `= add`" and "**forbid** `mul.`**right** `= add`" (cf. the mark $\{a\}$ in the production of m in Fig. 3).

8.1 Composing Language Definitions

In this section, we consider the use case of defining a language and composing it with another previously defined language, showing how we can deal with the resulting ambiguities that arise from the composition. The language being defined is a subset of Orc [Kitchin et al. 2009], a functional programming language which includes a number of special-purpose combinators for coordinating concurrent workflows. On their own, these combinators act as a DSL for concurrency.

In Orc, every expression can "publish" zero or more values. Orc defines four combinators for orchestrating these published values: the parallel (|), sequential (>x>), pruning (<x<), and "otherwise" (;) combinators. The expression e_1 | e_2 runs e_1 and e_2 in parallel, publishing any value published by either of them. The expression e_1 >x> e_2 executes $[x \mapsto v]e_2$ for *each* value v published by e_1, building a concurrent pipeline. The expression e_1 <x< e_2 executes $[x \mapsto v]e_1$ for *first* value v published by e_2 (discarding any remaining values published by e_2). Finally, e_1 ; e_2 runs e_2 only if running e_1 does not publish any values.

The syncon definition of the Orc combinators is very simple (the precedence rules follow the original definition [Kitchin et al. 2009]):

```
type Exp                          token Ident = "[[:lower:]][[:word:]]*"
grouping "(" Exp ")"              infix par:Exp = "|"
precedence {                      infix seq:Exp = ">" x:Ident ">"
  seq; par; prune; otherwise;     infix prune:Exp = "<" x:Ident "<"
}                                 infix otherwise:Exp = ";"
```

Naïvely composing this definition with another, separately defined language is likely to introduce ambiguities. For example, if that language also contains infix operators, the precedence between these and the Orc combinators will be undefined. With support for resolvable ambiguity, however, we can allow this ambiguity and let programmers disambiguate expressions using parentheses. After composing the above definition with a simple language supporting addition, variables, and

function calls (language definition omitted for brevity), parsing the expression "`1 + 2 >x> f(x)`" results in the following error message from our tool:

```
Ambiguity error with 2 alternatives:
  ( 1 + 2 ) > x > f ( x )
  1 + ( 2 > x > f ( x ) )
```

Because there is no precedence specified between + and >x>, disambiguation is required to specify the order of operations. In this case, it is likely that the preferred semantics are to have all operators in the base language bind tighter than the Orc combinators, and these precedence rules can be added after the composition without changing the original definitions. Importantly though, we are not *required* to resolve this ambiguity at time of composition.

Because of how we defined the syntax of the combinators, and since the tool is currently whitespace insensitive, the multi-character combinators (sequencing and pruning) are parsed as three separate lexical tokens (this is visible in how whitespace is inserted in the error message above). This means we can also run into *unresolvable* ambiguities, for example if our base language includes comparison of numbers with < and >. With such a base language (assuming no associativity for >), parsing the expression "`42 >x> f(x)`" will result in the following error message:

```
Unresolvable ambiguity error with 2 alternatives.

  Resolvable alternatives:
    ( 42 > x ) > f ( x )
    42 > ( x > f ( x ) )

  Unresolvable alternatives:
    seq
     - int                  gt.orc:1:1-3
     - call                 gt.orc:1:8-12
```

By adding parentheses, a programmer can disambiguate the expression as two comparisons. However, there is no way for a programmer to specify that what they want is a sequential combinator with left and right children being an integer and a function call, respectively. In this case we have at least three choices to make as language designers:

- We could decide to forbid the case where two comparisons are right next to each other, e.g., `forbid gt.left = gt` and `forbid gt.right = gt` (where `gt` is the syncon for the greater-than operator).
- We could change the definition of the parallel combinators and define separate lexical tokens for the combinators, e.g., `token Seq = ">[[:lower:]][[:word:]]*>"`. This would add just enough whitespace sensitivity to allow separating the different cases.
- We could change the syntax of the combinators to avoid clashes with other operators.

The change in the first alternative is somewhat ad-hoc, but can be done after composition and would allow us to reuse both language definitions without modifications. In the latter two alternatives, we lose reuse of the definitions of Orc combinators, but place no additional restrictions on the base language. The preferred resolution strategy is likely to differ between different compositions and different languages. For example, another unresolvable ambiguity is going to show up if the base language uses semi-colons, e.g. for sequencing expressions (this would clash with the "otherwise" combinator). In this case, there is no reasonable way to disambiguate "`e1; e2`" without changing the syntax of one of the operations.

The main takeaway from this case study is that resolvable ambiguity makes composition of languages less restrictive than if all ambiguity is completely banned. The approach is strictly more

general since dynamic resolvability analysis allows deferring disambiguation to the programmer, while removing ambiguities from the resulting grammar is also possible.

8.2 Finding Ambiguities with Dynamic Resolvability Analysis

In this section, we investigate how to apply dynamic resolvability analysis to identify ambiguities in the specification of a real-world language. Using syncons, we have specified a substantial subset of OCaml's syntax as described in chapters 7 and 8 (base language and extensions) of the OCaml reference manual [Leroy et al. 2019]. The syncon definition is just under 700 lines of code, and is currently complete enough to parse roughly 75% (1012 out of 1334 files) of the `.ml` files present in the OCaml compiler itself (we discuss limitations of our tool in the end of this section).

By using our implementation of the OCaml syntax to parse real-world OCaml code, we can dynamically identify ambiguities in the grammar presented in the reference manual, instead of silently resolving these in the implementation of an unambiguous parser. For example, an OCaml expression can be a function application *expr* {*argument*}$^+$ or a constructor application *constr expr*. Since an expression can also be a constructor, however, the language definition allows "`Foo 42`" to be parsed both as a function application and as a constructor application (only the latter is well-typed, but both are *syntactically* well-formed). Similarly, the expression "`Foo.f`" can be parsed as a field access targeting either a constructor or a module, since both must start with capital letters (again, only the latter is well-typed). Another example, that was already mentioned in Section 5, is the fact that semi-colons are used both for sequencing and for separating elements in lists, arrays and records. Thus, "`[1; 2]`" can be parsed both as a list of two elements and a singleton list equivalent to "`[(1; 2)]`".

As this case study shows, implementing a parser that reports ambiguities lets us identify ambiguities in a grammar through testing. Even though detecting ambiguities in a grammar is undecidable in general, dynamic resolvability analysis over a large corpus of code lets us find (and fix) many cases of ambiguity in practice. In this case study, out of the 700 lines of syncon code, ~200 lines are additions to conform to how the canonical compiler behaves on cases that are under-specified in the manual, including the examples listed above.

Current Limitations of Syncons. Other than syntactic OCaml constructs that are not yet specified, there are two sources of failure in the parts of the OCaml compiler that we cannot parse. First, our parsing tool does not support specifying "longest match" on a production, which is required to handle the pattern matching constructs correctly. Some cases can be handled using **`forbid`** declarations, but it does not get us all the way. Second, our system uses a different definition of precedence than the OCaml language. Our translation from precedence to implicit **`forbid`** declarations is shallow (it only considers direct children), while the OCaml has deep precedence. For example, addition binds stronger than **`let`** (which syntactically functions as a prefix operator in OCaml), and thus "`1 + let x = 1 in x + 2`" should be parsed as "`1 + (let x = 1 in (x + 2))`". Our tool, however, reports the expression as ambiguous, additionally suggesting the alternative interpretation "`(1 + (let x = 1 in x)) + 2`". This interpretation is indeed valid if we only look at the direct children of each operator, since "`(_ + _) + 2`", "`1 + let _ in _`", and "`let x = 1 in x`" are all individually correct with regards to precedence.

9 RELATED WORK

Afroozeh et al. [2013]'s operator ambiguity removal patterns bear a striking resemblance to the marks presented in this paper. However, in special-casing (what in this paper would be) marks on left and right-recursions in productions they correctly cover the edge case involving deep precedence discussed in Section 8.2. This approach thus suggests an interesting direction for future work: to

integrate it into a system designed around resolvable ambiguity, and extending the algorithms presented in this paper to cover it.

Danielsson and Norell [2011] give a method for specifying grammars for expressions containing mixfix operators. They allow non-transitive, non-total precedence, and "feel that it is overly restrictive to require the grammar to be unambiguous." Similar to our approach, they do not reject ambiguous grammars, only ambiguous parses. They also introduce a concept of *precedence correct* expressions; expressions where direct children must be related by precedence (children must have higher precedence than parents). This is more restrictive than our approach, e.g., in a language where `'+'` and `'-'` have no defined relative precedence they reject `'1 + 2 * 3'` as syntactically invalid, while we parse it as an ambiguous expression.

Silver [Van Wyk et al. 2010], a system for defining extensible languages using attribute grammars, and its associated parser Copper [Van Wyk and Schwerdfeger 2007] have a "Modular Well-Definedness Analysis" [Kaminski and Van Wyk 2013], with which extension language developers can check that their extension will compose well with other extensions, without any knowledge of any particular other extension. For this paper, only the syntactic component [Schwerdfeger and Van Wyk 2009] of this analysis is relevant. This analysis guarantees that the composition of a base language and any number of extensions that have passed the analysis will compose to a grammar in LALR(1). This is a far more restricted class of languages than unambiguous languages, not to mention resolvably ambiguous languages, and thus places far more restrictions on language designers than our approach.

The detection of classical ambiguity in context-free grammars is undecidable in general [Cantor 1962], yet numerous heuristic approaches exist. Examples include linguistic characterizations and regular language approximations [Brabrand et al. 2007], using SAT-solvers [Axelsson et al. 2008], and other conservative approaches [Schmitz 2007], For an overview, and additional approaches, see the PhD thesis of Basten [2011].

Numerous language development frameworks and libraries support syntactic language composition *without* any guarantees on the resulting language (e.g., [Heering et al. 1989]). These systems tend to have some form of general parser, so that they can handle arbitrary context-free grammars, but also assume that the composed language is unambiguous, which in practice precludes the composition of languages constructed independently of each other.

10 CONCLUSION

In this paper, we introduce the concept of *resolvable ambiguity*. A language grammar is resolvably ambiguous if all ambiguities can be resolved by the end-user at parse time. This approach departs from the common standpoint that grammars and syntax definitions of languages must be unambiguous. As part of the new concept, we formalize the fundamental *resolvable ambiguity problem*, divide it into static and dynamic parts, and provide solutions for both variants for a restricted class of languages. Through case studies, we show practical applicability of the approach, both for building new domain-specific languages and for reasoning about existing general-purpose languages.

In future work, we will investigate weakning the restrictions currently needed for the static and dynamic resolvability algorithms to work. We also intend to develop our theory and tools to support handling deep precedence ambiguities and ambiguities based on "longest match". A long term goal is to be able to suggest changes to the grammar of a language, based on ambiguities found through dynamic analysis.

REFERENCES

Ali Afroozeh, Mark van den Brand, Adrian Johnstone, Elizabeth Scott, and Jurgen Vinju. 2013. Safe Specification of Operator Precedence Rules. In *Software Language Engineering (Lecture Notes in Computer Science)*, Martin Erwig, Richard F. Paige,

and Eric Van Wyk (Eds.). Springer International Publishing, 137–156.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison Wesley, Boston.

Rajeev Alur and P. Madhusudan. 2004. Visibly Pushdown Languages. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing (STOC '04)*. ACM, New York, NY, USA, 202–211. https://doi.org/10.1145/1007352.1007390

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

Bas Basten. 2011. *Ambiguity Detection for Programming Language Grammars.* Ph.D. Dissertation. Universiteit van Amsterdam.

Claus Brabrand, Robert Giegerich, and Anders Møller. 2007. Analyzing Ambiguity of Context-Free Grammars. In *Implementation and Application of Automata (Lecture Notes in Computer Science)*, Jan Holub and Jan Žďárek (Eds.). Springer Berlin Heidelberg, 214–225.

Claus Brabrand and Jakob G. Thomsen. 2010. Typed and Unambiguous Pattern Matching on Strings Using Regular Expressions. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '10)*. ACM, New York, NY, USA, 243–254. https://doi.org/10.1145/1836089.1836120

David G. Cantor. 1962. On The Ambiguity Problem of Backus Systems. *J. ACM* 9, 4 (Oct. 1962), 477–479. https://doi.org/10.1145/321138.321145

Mathieu Caralp, Pierre-Alain Reynier, and Jean-Marc Talbot. 2015. Trimming Visibly Pushdown Automata. *Theoretical Computer Science* 578 (May 2015), 13–29. https://doi.org/10.1016/j.tcs.2015.01.018

H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 2007. *Tree Automata Techniques and Applications.* release October, 12th 2007.

Keith Cooper and Linda Torczon. 2011. *Engineering a Compiler* (2nd ed.). Elsevier.

Nils Anders Danielsson and Ulf Norell. 2011. Parsing Mixfix Operators. In *Implementation and Application of Functional Languages (Lecture Notes in Computer Science)*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Springer Berlin Heidelberg, 80–99.

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

Seymour Ginsburg and Joseph Ullian. 1966. Ambiguity in Context Free Languages. *J. ACM* 13, 1 (Jan. 1966), 62–89. https://doi.org/10.1145/321312.321318

J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. 1989. The Syntax Definition Formalism SDF—Reference Manual—. *SIGPLAN Not.* 24, 11 (Nov. 1989), 43–75. https://doi.org/10.1145/71605.71607

Ted Kaminski and Eric Van Wyk. 2013. Modular Well-Definedness Analysis for Attribute Grammars. In *Software Language Engineering (Lecture Notes in Computer Science)*, Krzysztof Czarnecki and Görel Hedin (Eds.). Springer Berlin Heidelberg, 352–371.

David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. 2009. The Orc Programming Language. In *Formal Techniques for Distributed Systems.* Springer, 1–25.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Rémy Didier, and Jérôme Vouillon. 2019. The OCaml system, release 4.08, Documentation and user's manual. https://caml.inria.fr/pub/docs/manual-ocaml.

Viktor Palmkvist and David Broman. 2019. Creating Domain-Specific Languages by Composing Syntactical Constructs. In *Practical Aspects of Declarative Languages (Lecture Notes in Computer Science)*, José Júlio Alferes and Moa Johansson (Eds.). Springer International Publishing, 187–203.

Sylvain Schmitz. 2007. Conservative Ambiguity Detection in Context-Free Grammars. In *Automata, Languages and Programming (Lecture Notes in Computer Science)*, Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki (Eds.). Springer Berlin Heidelberg, 692–703.

August C. Schwerdfeger and Eric R. Van Wyk. 2009. Verifiable Composition of Deterministic Grammars. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1542476.1542499

Thomas A. Sudkamp. 1997. *Languages and Machines: An Introduction to the Theory of Computer Science.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1 (Jan. 2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

Eric R Van Wyk and August C Schwerdfeger. 2007. Context-aware scanning for parsing extensible languages. In *Proceedings of the 6th international conference on Generative programming and component engineering.* ACM, 63–72.

Adam Brooks Webber. 2003. *Modern Programming Languages: A Practical Introduction.* Franklin, Beedle & Associates.