Resolvable Ambiguity[⋆]

Viktor Palmkvist[1], Elias Castegren[1], Philipp Haller[1], and David Broman[1]

KTH Royal Institute of Technology, 100 44 Stockholm, Sweden
{vipa,eliasca,dbro,phaller}@kth.se

Abstract. 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 a restricted language class and sketch proofs of soundness and 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.

Keywords: Syntax · Ambiguity · Grammars · Parsing · 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 [7]. 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 [2,10,15,29,32]. However, our recent work [23] suggests that carefully designed ambiguous syntax definitions can be preferable compared to completely unambiguous definitions.

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 to construct languages by composing and extending existing languages. Unfortunately, the composition of grammars easily produces an

[⋆] This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032)

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 [4–6] or restrictions on the composed grammars [17], 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 an end-user 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 decision problems. For a particular language, the *dynamic resolvable ambiguity problem* asks the following question: can every parse tree of *one particular* program be written unambiguously, while the *static resolvability problem* asks if every parse tree of *every* program can be written unambiguously. In particular, the latter problem is difficult, since it in general involves examining an infinite set (one per parse tree) of infinite sets (words that can parse as that parse tree) to find one without a unique element (i.e., an unambiguous word). Furthermore we must be able to produce an unambiguous program for a given parse tree to be able to give good error messages to an end-user encountering an ambiguity.

In this paper we describe a syntax definition formalism where ambiguities can be resolved by grouping parentheses, and precedence and associativity is specified through *marks* (c.f. Section 5). As a delimitation we only consider languages with balanced parentheses. Our approach builds on the following key insights: i) the language of a parse tree is visibly pushdown [3], ii) these languages can be encoded as words, and iii) the language of the word encodings of all trees for a language is also visibly pushdown. The last point allows the construction of a visibly pushdown automaton (VPDA) that we examine to solve the static problem for a particular language subclass. Our solution to the dynamic problem, which builds on the first point, is general.

A language designer can thus either use our solution to the static problem (if their language is in the appropriate subclass) or use an approach similar to unit testing through our solution to the dynamic problem. Our solution to the dynamic problem also gives suggested fixes to end-users encountering ambiguities.

More concretely, 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 describe the language of a single, arbitrary parse tree and present a novel linear encoding of the same (Section 5.1).
- We devise a decidable algorithm that solves the *dynamic resolvability problem* for languages with balanced parentheses (Section 6). Additionally, this algorithm also produces a minimal unambiguous word for each resolvable parse tree.
- We solve the *static resolvability problem* for a language subclass. We further show that a resolvable result can be preserved through certain modifications that bring the language into a larger subclass (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 underspecified language definition.

Before presenting the above contributions, the paper starts with motivating examples (Section 2), as well as preliminaries (Section 3). Finally, following a discussion of related work (Section 9), the paper provides some conclusions (Section 10).

2 Motivating Examples

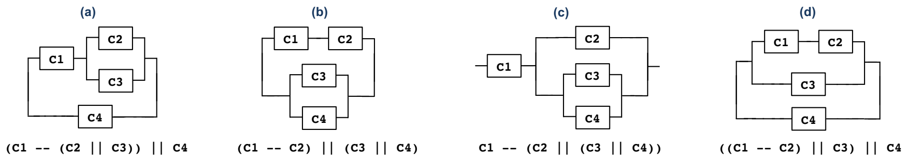
In this section, we motivate the need for ambiguous language definitions, where the decision of how to disambiguate a program is taken by the end-user (the programmer) and not the language designer. We motivate our new methodology both for engineering of new domain-specific languages, as well as for 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 Figure 1 which depicts four different alternatives, each



(C1 -- (C2 || C3)) || C4

(C1 -- C2) || (C3 || C4)

C1 -- (C2 || (C3 || C4))

((C1 -- C2) || C3) || C4

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) assuming the parallel operator is associative. In such a case, this expression and the expression in (c) both mean that components `C2`, `C3`, and `C4` are composed in parallel.

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 parentheses. 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 parentheses 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 this OCaml example of nested `match` expressions, as stated by [23]:

```
1  match 1 with                 | File "./nest.ml", line 4, characters 4-5:
2    | 1 -> match "one" with    | Error: This pattern matches values of type
3             | str -> str      |        int but a pattern was expected which
4    | 2 -> "two"               |        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:

```
match 1 with
  | 1 -> (match "one" with
          | str -> str)
  | 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 Language Composition

The possibility of composing different languages is a prevalent idea in research [11, 22, 30]. Particularly relevant to this paper is the behavior of composed *syntax*; composing unambiguous syntaxes does not necessarily produce an unambiguous composed syntax. Pre-existing systems commonly solve this either by strict requirements on the individual languages [30] or by largely ignoring the problem and merely presenting the multiple syntax trees in case of an ambiguity [11, 22]. The former case gives good guarantees for the end-user experience (no ambiguities) while the latter gives more freedom to the language designer. Resolvable ambiguity, as presented in this paper, provides a middle ground with more language designer freedom than the former and a different guarantee for the end-user: no *unresolvable* ambiguities.

Language composition has limited prevalence outside of research, but it does exist in a simplified form: custom operators in libraries. For example, Haskell allows a programmer to define new operators, with new syntax and custom precedence level and associativity. A user can then import these operators from any number of libraries and use them together in one program. Precedence in Haskell is defined as a number between 0 and 9, meaning that operators from different libraries have a defined relative precedence. If the libraries are designed by different authors then this precedence is unlikely to have been considered and may or may not be sensible. Leaving the relative precedence undefined would not affect expressivity, every possible combination of operators can be written by simply adding parentheses, and would avoid surprising interpretations. This does in fact happen in some cases in Haskell, namely when two operators have the same precedence but incompatible associativity. Both `>>=` and `=<<` have precedence 1, but the former is left-associative while the latter is right-associative. `a >>= b =<< c` then produces the following error:

```
Precedence parsing error
  cannot mix '>>=' [infixl 1] and '=<<' [infixr 1]
  in the same infix expression
```

The error message shows the problem but suggests no solution. Our approach can suggest solutions, allow arbitrary operators to have undefined relative precedence (intransitive precedence), and is general; it handles non-operator ambiguities as well.

3 Notation

This section briefly explains the notations used in this paper. Appendix A contains a more detailed description of the preliminaries (for reviewers).

Grammars Given a context-free grammar G, we write $L(G)$ to denote the set of all words recognized by G. The standard definition of ambiguity states that a word $w \in L(G)$ is ambiguous if there are two distinct leftmost derivations of that word.

Automata An automaton 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 (or single states of Q for a deterministic automaton); $q_0 \in Q$ an initial state; and $F \subseteq Q$ a set of final states. A pushdown automata further adds a set of stack symbols Γ and equips the transition function δ with information on which symbols are pushed, popped or just read from the stack.

Unranked Regular Tree Grammars Trees generalize words by allowing each terminal to have multiple ordered successors, instead of just zero or one. In this paper, we are only concerned with *unranked* trees, where the arity of a terminal is not fixed. The allowed sequence of ordered successors is described by a regular language [9]. For example, $a(n(\texttt{'1'})\ \texttt{'+'}\ n(\texttt{'2'}))$ represents the parse tree of the string "$1 + 2$". Finally, $\mathit{yield} : L(T) \to \Sigma^*$ is the sequence of terminals obtained by a left-to-right traversal of the leaves 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 1. *An abstract parser P is a triple $(\Sigma, T, \mathit{parse})$ consisting of*

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

Note that we do not require each tree to have a unique word, i.e., there may exist two distinct w_1 and w_2 such that $t \in \mathit{parse}(w_1)$ and $t \in \mathit{parse}(w_2)$. 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 2. *Given a word $w \in \Sigma^*$ and an abstract parser $P = (\Sigma, T, \mathit{parse})$, the set of words contained in the* parse language $L(P)$ *is defined as follows:*
$w \in L(P)$ *iff* $\mathit{parse}(w) \neq \emptyset$.

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

Definition 3. *Given an abstract parser $P = (\Sigma, T, \mathit{parse})$, a word $w \in L(P)$ is ambiguous, written $\mathit{amb}_P(w)$, iff $\exists t_1, t_2 \in T.\ t_1 \neq t_2 \wedge \{t_1, t_2\} \subseteq \mathit{parse}(w)$*

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

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 $\{w \mid \mathit{parse}(w) \neq \emptyset\}$.
- 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.

For *resolvable ambiguity* the definition instead centers around parse trees:

Definition 4. *Given an abstract parser $P = (\Sigma, T, \mathit{parse})$, a tree $t \in T$ is* resolvably ambiguous*, written $\rho_P(t)$, iff*
$\exists w \in \Sigma^*.\ \mathit{parse}(w') = \{t\}$.

A word is then resolvably ambiguous if all its parse trees are resolvably ambiguous:

Definition 5. *Given an abstract parser $P = (\Sigma, T, \mathit{parse})$, a word $w \in L(P)$ is* resolvably ambiguous*, written $\rho_P(w)$, iff*
$\forall t \in \mathit{parse}(w).\ \rho_P(t)$.

Additionally, we define an *abstract parser* to be resolvably ambiguous if all its parse trees are resolvably ambiguous, formally:

Definition 6. *An abstract parser P is resolvably ambiguous, written $\rho(P)$, iff*
$\forall t \in T.\ \rho_P(t)$.

We can now make the following observations:

- 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, $\forall t \in T.\ |\{w \mid t \in \mathit{parse}(w)\}| = 1$ 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 ambiguous program.

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 7 (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*

Definition 8 (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 9 (Soundness of Dynamic Resolvability). *A decision procedure f, solving the dynamic resolvability problem, is sound iff*
$f(P, w) \implies \rho_P(w)$ *for all $w \in \Sigma^*$ and abstract parsers $P = (\Sigma, T, \mathit{parse})$*

Definition 10 (Completeness of Dynamic Resolvability). *A decision procedure f, solving the dynamic resolvability problem, is complete iff*
$\rho_P(w) \implies f(P, w)$ *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. Furthermore, Section 4 suggests that disambiguation is made possible by letting *parse* discard information. We use unranked trees as parse trees and have *parse* 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.

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 $L(G'_D)$.

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	$N \to l : r$

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

$$\begin{array}{lllll}
E & \to & l & : & \texttt{'['}\ (E\ (\texttt{';'}\ E)^* + \epsilon)\ \texttt{']'} \\
E & \to & a & : & E\ \texttt{'+'}\ E \\
E & \to & m & : & E_{\{a\}}\ \texttt{'*'}\ E_{\{a\}} \\
E & \to & n & : & I
\end{array}$$

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 I matches a numeric terminal.

E	$\rightarrow$	$l\ (\ \texttt{'['}\ (\epsilon + E(\texttt{';'}\ E)^*)\ \texttt{']'}\ )$
E	$\rightarrow$	$a\ (\ E\ \texttt{'+'}\ E\ )$
E	$\rightarrow$	$m(\ E\ \texttt{'*'}\ E\ )$
E	$\rightarrow$	$n\ (\ I\ )$

(a) T_D, the parse trees of G_D.

E	$\rightarrow$	$l\ (\ \texttt{'['}\ (\epsilon + E(\texttt{';'}\ E)^*)\ \texttt{']'}\ )$
E	$\rightarrow$	$a\ (\ E\ \texttt{'+'}\ E\ )$
E	$\rightarrow$	$m(\ E_{\{a\}}\ \texttt{'*'}\ E_{\{a\}}\ )$
E	$\rightarrow$	$n\ (\ I\ )$
E	$\rightarrow$	$g\ (\ \texttt{'('}\ E\ \texttt{')'}\ )$
$E_{\{a\}}$	$\rightarrow$	$l\ (\ \texttt{'['}\ (\epsilon + E(\texttt{';'}\ E)^*)\ \texttt{']'}\ )$
$E_{\{a\}}$	$\rightarrow$	$m(\ E_{\{a\}}\ \texttt{'*'}\ E_{\{a\}}\ )$
$E_{\{a\}}$	$\rightarrow$	$n\ (\ I\ )$
$E_{\{a\}}$	$\rightarrow$	$g\ (\ \texttt{'('}\ E\ \texttt{')'}\ )$

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

E	$\rightarrow$	$\texttt{'['}\ E_{l1}\ \texttt{']'}$
E	$\rightarrow$	$E\ \texttt{'+'}\ E$
E	$\rightarrow$	$E\ \texttt{'*'}\ E$
E	$\rightarrow$	I
E_{l1}	$\rightarrow$	ϵ
E_{l1}	$\rightarrow$	$E\ E_{l2}$
E_{l2}	$\rightarrow$	ϵ
E_{l2}	$\rightarrow$	$\texttt{';'}\ E\ E_{l2}$

(c) G_D, the generated abstract syntax.

E	$\rightarrow$	$\texttt{'['}\ E_{l1}\ \texttt{']'}$
E	$\rightarrow$	$E\ \texttt{'+'}\ E$
E	$\rightarrow$	$E_{\{a\}}\ \texttt{'*'}\ E_{\{a\}}$
E	$\rightarrow$	I
E	$\rightarrow$	$\texttt{'('}\ E\ \texttt{')'}$
$E_{\{a\}}$	$\rightarrow$	$\texttt{'['}\ E_{l1}\ \texttt{']'}$
$E_{\{a\}}$	$\rightarrow$	$E_{\{a\}}\ \texttt{'*'}\ E_{\{a\}}$
$E_{\{a\}}$	$\rightarrow$	I
$E_{\{a\}}$	$\rightarrow$	$\texttt{'('}\ E\ \texttt{')'}$
E_{l1}	$\rightarrow$	ϵ
E_{l1}	$\rightarrow$	$E\ E_{l2}$
E_{l2}	$\rightarrow$	ϵ
E_{l2}	$\rightarrow$	$\texttt{';'}\ E\ E_{l2}$

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

Fig. 4: The generated grammars.

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 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. Each element corresponds to the word "$(1+2)*3$" in $L(G'_D)$. Note that 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.

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

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

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

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

$1 + 2 * 3$

(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.

$$\begin{tikzcd}
L(T_D) \arrow[d, "yield"] & L(T'_D) \arrow[l, "unparen"'] \arrow[d, "yield"] \\
L(G_D) & L(G'_D)
\end{tikzcd}$$

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

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 $\mathit{unparen} : L(T'_D) \to L(T_D)$ 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 $\mathit{parse} : L(G'_D) \to 2^{L(T_D)}$, along with its inverse $\mathit{words} : L(T_D) \to 2^{L(G'_D)}$:

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

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 $\mathit{parse}($`'1 + 2 + 3'`$) = \{t_1, t_2\}$ where

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

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

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

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 $\mathit{parse}($`'[1 ; 2]'`$) = \{t_3\}$.

5.1 The Word Language of a Parse Tree

Central to our approach is the shape of $\mathit{words}(t)$. Consider the tree corresponding to `'(1 + 2) * 3'`. Each node in the tree may be surrounded by zero or more parentheses, except `'1 + 2'`, which requires at least one pair. Its language is thus represented by $\{(^{n_1}(^{n_2}(^{n_3}1)^{n_3}+(^{n_4}2)^{n_4})^{n_2}*(^{n_5}2)^{n_5})^{n_1} \mid n_i \in \{0,1,\ldots\}, n_2 \geq 1\}$. To have something more manageable we introduce an alternative representation: a linear encoding as a word. This representation is convenient for Section 6 and essential for Section 7.

Continuing with the example above, we can encode this language as a word, if we take "()" to mean "exactly one pair of parentheses" and "[]" to mean "zero or more pairs of parentheses": "[[([1] + [2])] ∗ [3]]". This encoding lets us reduce comparisons between languages to comparisons between words, with one caveat: the encoding is not unique. For example, "([])" encodes the same language as "[()]", and "[[]]" encodes the same language as "[]". We rectify this by repeatedly swapping "([. . .])" with "[(. . .)]" and "[[. . .]]" with "[. . .]" until we reach a fixpoint. We call the result the *canonical linear encoding* of the (word) language of a tree in $L(T_D)$, which *is* unique.

We are now ready to construct decision procedures for the static and dynamic resolvability problems, as given in Definitions 7, 8, 9, and 10. Section 7 solves the static problem for a language subclass, while Section 6 fully solves the dynamic problem.

6 Dynamic Resolvability Analysis

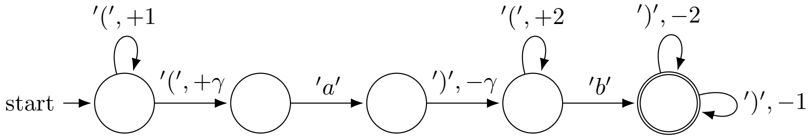
The dynamic resolvability problem is as follows: for a given $w' \in L(G'_D)$ determine whether $\forall t \in \mathit{parse}(w').\ \exists w'_2.\ \mathit{parse}(w'_2) = \{t\}$. Furthermore, for practical reasons, if the word is resolvably ambiguous we wish to produce a (minimal) witness for each tree. We place two restriction on languages D we consider:

1. Stemming from our initial delimitation, 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. Additionally, if we already know statically that D is resolvably ambiguous[1] we can drop the second requirement.

[1] While the *decision* part of this procedure is uninteresting in this case (it will always answer "resolvable"), the minimal witness *is* interesting: it shows the user how to resolve the ambiguity.

Our approach centers around the construction of a VPDA recognizing $\mathit{words}(t)$ for any particular $t \in L(T_D)$. We can easily construct this automaton via the canonical linear encoding: let every pair "()" push and pop the same stack symbol (call it γ) and let every "[]" push and pop a unique stack symbol. For example, "$[(a)[b]]$" produces the following automaton:



This is useful to us since VPDAs are closed under difference. Given a pair of trees t_1 and t_2 we can thus produce a pair of automata recognizing $\mathit{words}(t_1)$ and $\mathit{words}(t_2)$ respectively, then construct a pair of automata recognizing $\mathit{words}(t_1) \setminus \mathit{words}(t_2)$ and $\mathit{words}(t_2) \setminus \mathit{words}(t_1)$, respectively. These new automata recognize the words that are not ambiguous between these two particular trees. Our algorithm can thus be seen as Algorithm 1.

Algorithm 1 The Dynamic Algorithm

```
procedure DYNAMIC($T$)
    $R \leftarrow \emptyset$
    $U \leftarrow \emptyset$
    for all $t \in T$ do
        $T' \leftarrow T \setminus \{t\}$
        $A \leftarrow \mathit{words}(t)$                                   ▷ $A$ is a VPDA.
        loop
            $A \leftarrow A \setminus \bigcup_{t' \in T'} \mathit{words}(t')$   ▷ VPDAs closed under difference and union.
            if $L(A) = \emptyset$ then
                $U \leftarrow U \cup \{t\}$
                break loop
            if $\mathit{parse}(\mathit{shortest}(A)) = \{t\}$ then        ▷ $\mathit{shortest}(A)$ denotes a shortest
                $R \leftarrow R \cup \{(t, \mathit{shortest}(A))\}$              ▷ word recognized by $A$.
                break loop
            $T' \leftarrow \mathit{parse}(\mathit{shortest}(A)) \setminus \{t\}$
    return($R, U$)
```

Note that the shortest word we produce might not be unique, in the general case there may be more than one shortest word that disambiguates a tree.

We require two things to ensure termination: that T' is finite, and that the inner loop terminates. T' is finite if no words are infinitely ambiguous, which is ensured by requirement 2. It is also ensured if D is resolvably ambiguous, since an infinite ambiguity requires that G_D contains a cycle $N \Rightarrow^+ N$, which would imply that any parse tree containing a production from N only has (infinitely) ambiguous words, i.e., contradiction. Requirement 2 also ensures that the number of trees that share an ambiguous word with t is finite (since each tree corresponds to a left-most derivation in G_D and must share exactly the same non-parenthesis terminals).

7 Static Resolvability Analysis

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 $\mathit{parse}(w) = \{t\}$, 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.

Our approach finds this counterexample by finding a pair of trees t_1 and t_2 such that $\mathit{words}(t_1) \subseteq \mathit{words}(t_2)$. Such a pair trivially implies that t_1 has no unambiguous words, since $\forall w.\ t_1 \in \mathit{parse}(w) \rightarrow t_2 \in \mathit{parse}(w)$, which thus implies that the examined language is unresolvably ambiguous. The converse is less obvious, in fact, the absence of such a pair does not guarantee a resolvably ambiguous language in general. We shall however show it to be sufficient for a limited class of languages.

We now divide the possible languages along two axes: whether a language has (non-grouping) parentheses, and whether a language has marks. The parentheses present in the canonical encoding will have the following shapes ($\mathbb{N} = \{0, 1, \ldots\}$):

	Parentheses	No parentheses
Marks	$[^1(^{\mathbb{N}}$ or $[^0(^{\mathbb{Z}_+}$	$[^1(^{\mathbb{N}}$
No marks	$[^1(^{\mathbb{N}}$ or $[^0(^{\mathbb{Z}_+}$	$[^1(^0$

The absence of both non-grouping parentheses and marks mean that every occurrence of a pair of parentheses must permit zero or more pairs. Adding marks adds required pairs for some trees, turning each occurrence to one of k or more pairs, for some non-negative integer k determined by the tree. The left column is more complicated since non-grouping parentheses can introduce occurrences that require an exact number of parentheses, as well as all the complications of marks.

The remainder of this section will be devoted to first showing that $\neg\exists t_1, t_2 \in L(T_D).\ \mathit{words}(t_1) \subseteq \mathit{words}(t_2)$ implies that D is resolvably ambiguous if D is in the lower-right quadrant, i.e., if it has no marks and no non-grouping parentheses (in Section 7.1), and then describing the automaton that forms the basis for our algorithm (Section 7.2). Section 7.1 additionally shows that adding marks to a resolvably ambiguous language in the lower-right quadrant preserves resolvability, which gives us a conservative approach for languages in the top-right quadrant. Intuitively, if a language has marks, but they are not required for resolvability, then we can still detect that the language is resolvable. For example, most expression languages fall in this category (we can write any expression unambiguously by adding parentheses everywhere, even without any precedence or associativity), while lists in OCaml do not (no amount of parentheses will make `'[1;2]'` look like a list of two elements without a mark).

7.1 The Road to Correctness

This section shows that $\neg\exists t_1, t_2 \in L(T_D).\ \mathit{words}(t_1) \subseteq \mathit{words}(t_2)$ implies that D is resolvably ambiguous if D has no non-grouping parentheses and no marks. These requirements can be written more formally as follows:

1. $\Sigma \cap \{\texttt{'('}, \texttt{')'}\} = \emptyset$.
2. Every non-terminal N_m on the right hand side of a productin in D has $m = \emptyset$.

Together, these imply that the canonical linear encoding of a tree has no "()" (required parentheses), only "[]" (optional parentheses).

Consider an arbitrary $t \in L(T_D)$ with canonical linear encoding c. Now replace each pair "[]" with exactly one pair of parentheses, calling the result $w_\top$. We have $w_\top \in \mathit{words}(t)$ by construction. Now consider the two possibilities of ambiguity for $w_\top$:

$w_\top$ **is ambiguous.** In this case, $\exists t'.\ t \neq t' \wedge t' \in \mathit{parse}(w_\top)$. This means that we can take the canonical encoding of t' (call it c') and replace every "[]" with either one or zero parentheses and produce $w_\top$. But that also means that every word we construct from c (by choosing some non-negative integer of parentheses for each "[]") can also be constructed from c' (by choosing the same number for corresponding parentheses, and zero for the others), i.e., $\mathit{words}(t) \subseteq \mathit{words}(t')$.

$w_\top$ **is unambiguous.** In this case $\mathit{words}(t)$ has at least one unambiguous word ($w_\top$) which thus cannot be part of $\mathit{words}(t')$ for any other $t' \in L(T_D)$, thus $\neg\exists t'.\ t \neq t' \wedge \mathit{words}(t) \subseteq \mathit{words}(t')$.

We thus arrive at the first of two central theorems for our approach:

Theorem 1. *Given a language with productions D and terminals Σ, where $\Sigma \cap \{\texttt{'('}, \texttt{')'}\} = \emptyset$ and all non-terminals N_m appearing in a right-hand side of a production in D having $m = \emptyset$, the following holds: $\rho_P(D) \longleftrightarrow \neg\exists t_1, t_2 \in L(T_D).\ t_1 \neq t_2 \wedge \mathit{words}(t_1) \subseteq \mathit{words}(t_2)$.*

Adding marks to such a language preserves resolvability, since:

- Marks can introduce at most one required pair of parentheses per node.
- Adding marks to a language can only ever shrink $\mathit{words}(t)$ for any particular $t \in L(T_D)$.
- Given a tree t, the word produced by putting one pair of parentheses around each node in t (thus potentially adding double parentheses), call it w, is unambiguous in the language without marks. It cannot be excluded from $\mathit{words}(t)$ by adding marks (since it already has a pair of parentheses around each node), and it also cannot be ambiguous, since that would require $\mathit{words}(t')$ for some other tree t' to have grown. Thus, $\mathit{parse}(w) = \{t\}$ even when we add marks to D.

Theorem 2. *Adding marks to a resolvably ambiguous language D preserves resolvability, if D had no parentheses and no marks.*

7.2 The Algorithm

This section describes the construction of a VPDA that we examine to determine if a language has a pair of trees where the language of one entirely contains the other. We begin by outlining the approach and then describe it in more detail.

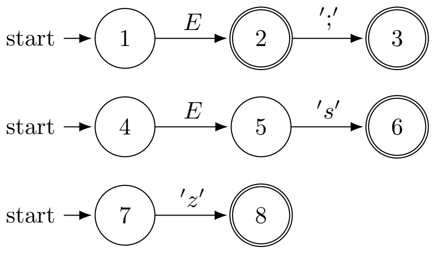
The automaton we examine recognizes canonical linear encodings and has a one-to-one correspondence between runs and trees. Two distinct runs that recognize the same word thus corresponds to two distinct trees that have the *same* word language. This is essentially the question of whether a VPDA is ambiguous, which we can detect by constructing a product automaton and trimming it.

To detect subsumption we make the following observation: given two trees t_1 and t_2 with canonical encodings c_1 and c_2, respectively, we have $\mathit{words}(t_1) \subset \mathit{words}(t_2)$ iff we can add arbitrary well-nested "[]" pairs to c_1, producing a c_1' such that $c_1' = c_2$ (Note that we do not need to consider "()" since no such pairs appear in this language class). We thus make a product automaton of two slightly different automata, where the second may add arbitrary "[]", but is otherwise identical.

Illustration by Example We begin by constructing a VPDA that recognizes *a* linear encoding for each tree, then modify it to only recognize the *canonical* linear encoding. To illustrate the approach we consider the following language:

$$\begin{array}{l} S \to a : E(\texttt{';'} + \epsilon) \\ E \to b : E\texttt{'s'} \\ E \to c : \texttt{'z'} \end{array}$$

This language has no non-grouping parentheses and no marks and is thus in the language class we consider. We begin by constructing a DFA per production (using standard methods, since the right hand side is a regular expression).



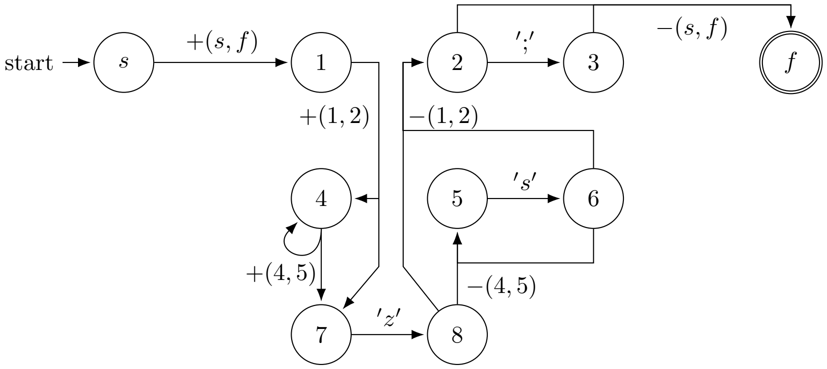
We then combine these separate DFAs to produce a single VPDA in three steps:

1. Introduce two new distinct states (call them s and f) and a transition $s \xrightarrow{S} f$, where S is the starting symbol of the language being examined.
2. Replace each transition $p \xrightarrow{N} q$ with:
   - A transition $p \xrightarrow{'[', +(p,q)} q_0$ for every initial state q_0 in a DFA corresponding to a production with lefthand side N.

  - A transition $q_f \xrightarrow{']',-(p,q)} q$ for every accepting state q_f in a DFA corresponding to a production with lefthand side N.

3. Make s the only initial state, and f the only accepting state.

To reduce clutter we omit $'['$ and $']'$, i.e., we abbreviate $p \xrightarrow{'[',+(p',q')} q$ as $p \xrightarrow{+(p',q')} q$ and $p \xrightarrow{']',-(p',q')} q$ as $p \xrightarrow{-(p',q')} q$.



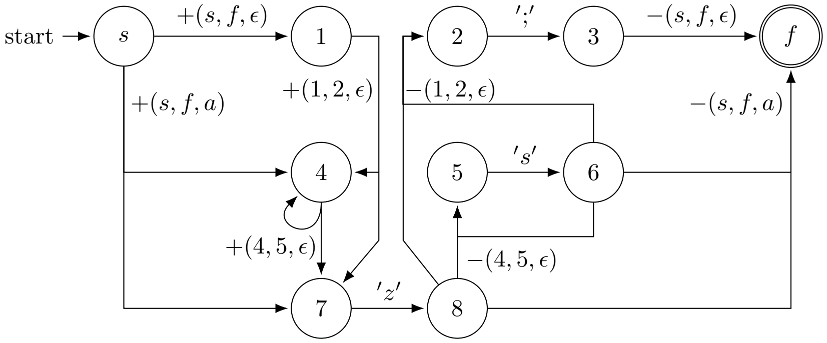
This automaton recognizes a linear encoding by simply putting a pair $'[]'$ around each production. That encoding will be canonical in many cases (e.g. "$[[[z]s];]$"), but not all; "$[[z]]$", which is recognized for the tree $a(c(z))$, is not canonical, it should be "$[z]$". The extra $'[]'$ pair stems from $S \to a : E(\text{';'} + \epsilon)$, its righthand side matches E (i.e., a word consisting of a single non-terminal).

We solve this by first recording which productions have righthand sides that match single non-terminal words (and which non-terminal) and then compute DFAs that do *not* match such words. For our running example we note that the production a matches the word E (we record this as a tuple (S, a, E)) and replace its automaton with one that does not match E. In this case it is sufficient to mark state 2 as non-accepting.

The recorded tuple (S, a, E) can be read as "starting from non-terminal S we can match the word E by choosing the production with label a." If we consider the middle element as a sequence of labels we can build a relation by adding (N, ϵ, N) for all non-terminals N and computing the closure of $(A, w, B) \cdot (B, w', C) = (A, w \cdot w', C)$. Call the resulting set T.

Finally, we change step 2 in our construction to the following:

2. Replace each transition $p \xrightarrow{N} q$ with:
  - A transition $p \xrightarrow{'[',+(p,q,w)} q_0$, for every initial state q_0 in a DFA corresponding to a production with lefthand side N', for every $(N, w, N') \in T$.
  - A transition $q_f \xrightarrow{']',-(p,q,w)} q$, for every accepting state q_f in a DFA corresponding to a production with lefthand side N', for every $(N, w, N') \in T$.



We call this automaton $A_{[]}$. We also construct a modified automaton that recognizes linear encodings of larger word languages. We call this automaton $A'_{[]}$ and obtain it by adding transitions $p \xrightarrow{'[',+\gamma} p$ and $p \xrightarrow{']',-\gamma} p$ for every state p, where γ is a new distinct stack symbol. Note that all these new transition push and pop the *same* stack symbol, meaning that a new pair "[]" can be started in any state and then closed in any state (not just the same state) as long as intermediate transitions leave the stack unchanged, i.e., only introduce zero or more well-balanced pairs.

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

- $(N, \epsilon, N) \in T$ for all $N \in V$.
- $(N, l, N') \in T$ for all $N' \in V \cap L(\mathit{regex}(l))$ where $\mathit{nt}(l) = N$.
- $(N_1, w_1 \cdot w_2, N_3) \in T$ for all $(N_1, w_1, N_2), (N_2, w_2, N_3) \in T$.

T is finite if D has no cycles $N \Rightarrow^+ N$. 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 $\Sigma \cup L$:

$\mathit{dfa}(l)$**:** Compute the regular language $L' := L(\mathit{regex}(l)) \setminus V$, then construct a DFA for this language.

We now define the VPDA $A_{[]}$. Given $\mathit{dfa}(l) = (Q_l, \Sigma \cup V, \delta_l, s_l, F_l)$ for all $l \in \mathit{Labels}$, $A_{[]} = (Q, \Sigma', \Gamma, \delta, s, \{f\})$ where:

- s and f are two new distinct states.
- $Q = \{s, f\} \cup \bigsqcup_l Q_l$.
- $\Sigma' = \Sigma \cup \{'[', ']'\}$.

- $\Gamma = Q \times Q \times L^*$.
- δ is defined by the following equations (unspecified cases produce $\emptyset$):

$$\begin{aligned}
&\delta(p, a, \lambda) = \{(\delta_l(p, a), \lambda)\} \qquad \text{where } p \in Q_l \text{ and } a \in \Sigma \\
&\delta(s, \text{“[”}, \lambda) = \{(s_l, (s, f, w)) \mid (S, w, N) \in T, N = nt(l)\} \\
&\delta(p, \text{“[”}, \lambda) = \\
&\qquad \{(s_{l_3}, (p, q, w)) \mid p \in Q_{l_1}, \delta_{l_1}(p, N) = q, (N, w, N') \in T, N' = nt(l_3)\} \\
&\delta(p, \text{“]”}, (s, f, w)) = \{(f, \lambda) \mid (S, w, N) \in T, N = nt(l), p \in F_l\} \\
&\delta(q', \text{“]”}, (p, q, w)) = \\
&\qquad \{(q, \lambda) \mid q' \in F_{l_1}, nt(l_1) = N', (N, w, N') \in T, p \in Q_{l_2}, \delta_{l_2}(p, N) = q\}
\end{aligned}$$

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:

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

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 described in [3] (using the partitions $\Sigma_c = \{[\}$, $\Sigma_i = \Sigma$, 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 successful runs in $A_{[]}$ and $A'_{[]}$. We check for the existence of such a run by trimming the product automaton (as described in [8]) 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 on a language definition D without non-grouping parentheses and marks gives a sound and complete decision procedure for determining if D is resolvably ambiguous. If we instead take a language definition D with marks but no non-grouping parentheses, then remove the marks and perform the procedure, then the absence of two such runs implies that D is resolvably ambiguous, while their presence only implies the possibility of an unresolvable ambiguity; the marks might remove it.

Lemma 1. *There is a bijection between successful runs in $A_{[]}$ and trees $t \in L(T_D)$.*

Proof (sketch). 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.

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

Lemma 2. $\forall t \in L(T_D)$. *$word(run(t))$ is the canonical linear encoding of $words(t)$.*

Lemma 3. *For all $t \in L(T_D)$ and every linear encoding c such that $words(t) \subseteq L(c)$, $c \in L(A'_{[]})$.*

Lemma 4. *We can define a function normalrun from runs in $A'_{[]}$ to runs in $A_{[]}$ such that $\forall R.\ L(word(normalrun(R))) \subseteq L(word(R))$.*

Proof (sketch). By removing every transition that pushes or pops γ.

Theorem 3. *The existence of two distinct runs R and R' such that 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 word language of one is entirely contained in the other.*

Proof (sketch). 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 $L(word(normalrun(R'))) \subseteq L(word(R'))$ and $L(word(R')) = L(word(R))$ we have that $L(t_1) \subseteq L(t_2)$.

We thus have a way to detect the presence or absence of a pair of trees $t_1, t_2 \in L(T_D)$ such that $words(t_1) \subseteq words(t_2)$, which completely determines the resolvability of D when D has no non-grouping parentheses or marks.

8 Case Studies

We have implemented a tool and a small DSL for syntax definition (Section 8.1) 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. In this section, we showcase two possible use-cases of dynamic resolvability analysis using this tool: composing separately defined DSLs (Section 8.2), and finding ambiguities in grammars through testing (Section 8.3).

8.1 A DSL for Syntax Definition

We write our syntax definitions in a DSL building on *syncons*, introduced by Palmkvist and Broman [23], wherein each production is defined separately from each other (one `syncon` each). The tool is implemented in Haskell and uses a custom implementation of the Earley parsing algorithm [12].

The main difference between the DSL and the formalism in Section 5 is that marks are introduced as separate `forbid` declarations instead of being inlined in productions. This allows supporting 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 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 list means   "[" (head:Exp (";" tail:Exp)*)? "]"
  add; // 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 bracketed sequence of zero or more expressions separated by semi-colons. Finally, addition and multiplication are defined as infix syncons with the expected precedence rules. Note that we could also 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).

8.2 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 [19], 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 [19]):

```
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 use parentheses for disambiguation. After composing the above definition with a simple language supporting addition, variables, and function calls (full 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.


                                  Unresolvable alternatives:
   Resolvable alternatives:         seq
     ( 42 > x ) > f ( x )            - int        gt.orc:1:1-3
     42 > ( x > f ( x ) )            - 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.

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.

8.3 Finding Ambiguities with Dynamic Resolvability Analysis

In this section, we investigate how to apply dynamic resolvability analysis to find 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 [21]. The syncon definition is just under 700 lines long, and can currently 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

Our related work falls in three categories: syntax definition formalisms (including subclasses of CFGs), language frameworks, and other approaches to ambiguity.

Afroozeh et al. [1]'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 8.3. This approach thus suggests an interesting direction for future work: extend the algorithms presented in this paper to cover it.

Danielsson and Norell [11] 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 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.

Parsing expression grammars [14] sidestep the issue of ambiguity by not introducing it at all. However, this also loses the potential gains of leaving certain ambiguities. Additionally, since the ordering of productions matter, composition of languages must be ordered, and the interactions between composed languages becomes non-obvious, e.g., merely adding productions may remove previously recognized words from the language, depending on where they are added.

Most commonly used parser generators are based on unambiguous CFG subclasses, e.g., LL(k), LR(k), or LR(*). Others do not fit neatly in the Chomsky hierarchy, but still produce a single parse tree per parse, e.g., LL(*) [24] and ALL(*) [25]. Yet others produce multiple parse trees or other forms of parse forests, e.g., GLR [20], GLL [28], and Earley [12].

Silver [30], a system for defining extensible languages using attribute grammars, and its associated parser Copper [31] have a "Modular Well-Definedness Analysis" [17], the syntactic component of which can be found in [27]. 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 language class is more restrictive than both unambiguous and resolvably ambiguous languages, though somewhat comparable to the subclass our static analysis supports.

The detection of classical ambiguity in context-free grammars is undecidable in general [7], yet numerous heuristic approaches exist. Examples include linguistic characterizations and regular language approximations [6], using SAT-solvers [4], and other conservative approaches [26], For an overview, and additional approaches, see the PhD thesis of Basten [5].

Numerous language development frameworks and libraries support syntactic language composition *without* any guarantees on the resulting language (e.g., [13, 16, 18, 22]). These systems tend to have some form of general parser, so that they can handle arbitrary context-free grammars, but mention no handling of ambiguities encountered by an end-user.

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 weakening 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

1. Afroozeh, A., van den Brand, M., Johnstone, A., Scott, E., Vinju, J.: Safe Specification of Operator Precedence Rules. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) Software Language Engineering. pp. 137–156. Lecture Notes in Computer Science, Springer International Publishing (2013)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley, Boston, 2nd edn. (Sep 2006)
3. Alur, R., Madhusudan, P.: Visibly Pushdown Languages. In: Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing. pp. 202–211. STOC '04, ACM, New York, NY, USA (2004). https://doi.org/10.1145/1007352.1007390

4. Axelsson, R., Heljanko, K., Lange, M.: Analyzing Context-Free Grammars Using an Incremental SAT Solver. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) Automata, Languages and Programming. pp. 410–422. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2008)
5. Basten, B.: Ambiguity Detection for Programming Language Grammars. Ph.D. thesis, Universiteit van Amsterdam (Dec 2011)
6. Brabrand, C., Giegerich, R., Møller, A.: Analyzing Ambiguity of Context-Free Grammars. In: Holub, J., Žďárek, J. (eds.) Implementation and Application of Automata. pp. 214–225. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2007)
7. Cantor, D.G.: On The Ambiguity Problem of Backus Systems. Journal of the ACM **9**(4), 477–479 (Oct 1962). https://doi.org/10.1145/321138.321145
8. Caralp, M., Reynier, P.A., Talbot, J.M.: Trimming visibly pushdown automata. Theoretical Computer Science **578**, 13–29 (May 2015). https://doi.org/10.1016/j.tcs.2015.01.018
9. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (2007), release October, 12th 2007
10. Cooper, K., Torczon, L.: Engineering a Compiler. Elsevier, 2nd edn. (Jan 2011)
11. Danielsson, N.A., Norell, U.: Parsing Mixfix Operators. In: Scholz, S.B., Chitil, O. (eds.) Implementation and Application of Functional Languages. pp. 80–99. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2011)
12. Earley, J.: An Efficient Context-free Parsing Algorithm. Communications of the ACM **13**(2), 94–102 (Feb 1970). https://doi.org/10.1145/362007.362035
13. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: Layout-Sensitive Generalized Parsing. In: Czarnecki, K., Hedin, G. (eds.) Software Language Engineering. pp. 244–263. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2013)
14. Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 111–122. POPL '04, ACM, New York, NY, USA (2004). https://doi.org/10.1145/964001.964011
15. Ginsburg, S., Ullian, J.: Ambiguity in Context Free Languages. Journal of the ACM **13**(1), 62–89 (Jan 1966). https://doi.org/10.1145/321312.321318
16. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The Syntax Definition Formalism SDF—Reference Manual—. SIGPLAN Not. **24**(11), 43–75 (Nov 1989). https://doi.org/10.1145/71605.71607
17. Kaminski, T., Van Wyk, E.: Modular Well-Definedness Analysis for Attribute Grammars. In: Czarnecki, K., Hedin, G. (eds.) Software Language Engineering. pp. 352–371. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2013)
18. Kats, L.C., Visser, E.: The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 444–463. OOPSLA '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1869459.1869497
19. Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc programming language. In: Formal Techniques for Distributed Systems, pp. 1–25. Springer (2009)
20. Lang, B.: Deterministic Techniques for Efficient Non-Deterministic Parsers. In: Loeckx, J. (ed.) Automata, Languages and Programming. pp. 255–269. Lecture Notes in Computer Science, Springer Berlin Heidelberg (1974)

21. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system release 4.07: Documentation and user's manual. Report (Jul 2018)

22. Lorenzen, F., Erdweg, S.: Sound Type-dependent Syntactic Language Extension. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 204–216. POPL '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837644

23. Palmkvist, V., Broman, D.: Creating Domain-Specific Languages by Composing Syntactical Constructs. In: Alferes, J.J., Johansson, M. (eds.) Practical Aspects of Declarative Languages. pp. 187–203. Lecture Notes in Computer Science, Springer International Publishing (2019)

24. Parr, T., Fisher, K.: LL(*): The Foundation of the ANTLR Parser Generator. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 425–436. PLDI '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1993498.1993548

25. Parr, T., Harwell, S., Fisher, K.: Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. pp. 579–598. OOPSLA '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2660193.2660202

26. Schmitz, S.: Conservative Ambiguity Detection in Context-Free Grammars. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) Automata, Languages and Programming. pp. 692–703. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2007)

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

28. Scott, E., Johnstone, A.: GLL Parsing. Electronic Notes in Theoretical Computer Science **253**(7), 177–189 (Sep 2010). https://doi.org/10.1016/j.entcs.2010.08.041

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

30. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Science of Computer Programming **75**(1), 39–54 (Jan 2010). https://doi.org/10.1016/j.scico.2009.07.004

31. Van Wyk, E.R., Schwerdfeger, A.C.: Context-aware Scanning for Parsing Extensible Languages. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering. pp. 63–72. GPCE '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1289971.1289983

32. Webber, A.B.: Modern Programming Languages: A Practical Introduction. Franklin, Beedle & Associates (2003)

A Preliminaries

This appendix briefly describes some of the theoretical foundations we build upon.

A.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 11. *A word $w \in L(G)$ is ambiguous if there are two distinct leftmost derivations of w.*

A.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 $r_0 = q_0$, $\forall i \in \{0, 1, \ldots, n-1\}.\ r_{i+1} \in \delta(r_i, a_i)$ and $r_n \in F$. We say that the automaton accepts the word $a_0 a_1 \cdots a_n$ iff there is such a successful 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 to/from which transitions can push/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 only consider pushdown automata with relatively limited stack manipulation, and thus use some convenient shorthands:

- $p \xrightarrow{a} q$, 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., $\delta(p, a, g) \supseteq \{(q, \lambda)\}$.

A.3 Visibly Pushdown Languages

A visibly pushdown language [3] is a language that can be recognized by a visibly pushdown automaton. A visibly pushdown automaton (VPDA) 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 have 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 the following particularly relevant properties:

- Visibly pushdown languages with the same input partitions are closed under intersection, complement, and union [3]. Intersection is given by a product automaton. 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, when $c \in \Sigma_c$, $i \in \Sigma_i$ and $r \in \Sigma_r$:

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

- A VPDA can be trimmed [8], 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.

A.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 [9]), often a regular language.

The results and properties presented in this paper are more naturally described through unranked trees, thus all references to trees here 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 $V \times X \times \mathit{Reg}(\Sigma \cup X)$. We will write a production (N, x, r) as $N \to x(r)$.

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 $L(T)$[2]. Finally, $\mathit{yield} : L(T) \to \Sigma^*$ is the sequence of terminals $a \in \Sigma$ obtained by a left-to-right[3] traversal of a tree. Informally, it is the flattening of a tree after all internal nodes have been removed.

[2] Again, to distinguish from regular expressions and context-free languages, all trees will be named T, possibly with a subscript.

[3] Preorder, postorder, or inorder does not matter since terminals in Σ only appear as leaves