Resolvable Ambiguity

ANONYMOUS AUTHOR(S)

Text of abstract . . . .

Additional Key Words and Phrases: keyword1, keyword2, keyword3

1 INTRODUCTION

When constructing domain-specific programming languages there is often a desire to reuse work, to compose smaller language fragments. However, how to make such compositions well-behaved is not always obvious. In particular, composing multiple syntaxes is likely to produce an ambiguous grammar, even if all of the original grammars are unambiguous. 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 words. i.e., ambiguous programs. The former is well explored in the form of heuristics for ambiguity detection (e.g. [Axelsson et al. 2008; Basten 2011; Brabrand et al. 2007]) or restrictions on the grammars to be composed (e.g. [Kaminski and Van Wyk 2013]), while the latter has received relatively little scrutiny. This seems to stem from the view that ambiguity has no place in programming language syntax, which is highly prevalent (e.g. [Aho et al. 2006; Cooper and Torczon 2011; Ginsburg and Ullian 1966; Sudkamp 1997; Webber 2003]).

Handling ambiguity at parse-time on the other hand, beyond just using a general parsing algorithm (e.g. [Earley 1970; Scott and Johnstone 2010; Younger 1967]), seems limited to merely presenting each possible interpretation as an abstract syntax tree of some form [Danielsson and Norell 2011; Palmkvist and Broman 2019]. However, an abstract syntax tree is essentially an implementation detail, and might not provide any useful clues for an end-user to resolve the ambiguity. Another possibility, mentioned by Palmkvist and Broman [2019] but not yet explored, is to automatically generate several unambiguous programs, one for each posssible interpretation, thereby directly showing a user how to resolve the ambiguity, regardless of their desired interpretation. For example, in a language without precedence, ”$1 + 2 * 3$” is ambiguous, but the two possible interpretations can be written unambiguously as ”$(1 + 2) * 3$” and ”$1 + (2 * 3)$” respectively.

In exploring this approach, we find that this is not always possible; some ambiguities have one or more interpretations for which no unambiguous program is exists. For example, in a language with semi-colon for both sequential composition and element separation in a list, ”$[1; 2]$” is ambiguous: it is either a list of two elements, or a list of one element, namely the sequential composition of 1 and 2. The former can be written unambiguously as ”$[(1; 2)]$”, but the latter has no unambiguous alternative. This presents a problem, since such an ambiguity is an error an end-user might encounter, yet it can only be solved by the language designer, since it requires a change to the language grammar. We thus have a concept of ambiguities that can be resolved by an end-user, and those that cannot. We call an instance of the former a *resolvable ambiguity*, and the latter an *unresolvable ambiguity*. This paper introduces this concept in the context of language theory.

Resolvable ambiguity, like classical ambiguity, gives rise to two questions: can we determine if a particular word is unresolvably ambiguous (dynamic resolvability), and can we determine if a language has any unresolvably ambiguous words (static resolvability)? For classical ambiguity in context-free grammars, the former problem is solved by any general parser, while the latter is undecidable [Cantor 1962]. For resolvable ambiguity, in a formalism that associates trees with

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

EBNF-grammars, the dynamic problem is similarly solvable, and the static problem is similarly difficult. We do not yet know if the latter is undecidable in general, but we have found a subclass of languages for which it is decidable.

More specifically, we claim the following contributions:

- A formal definition of *resolvable ambiguity* in terms of languages where words have associated interpretations, as well as formulations of the *static* and *dynamic* resolvability problems, i.e., determine if a grammar (resp. word) is resolvably ambiguous (Section 4).
- A syntax definition formalism based on EBNF, where ambiguities can be resolved with grouping parentheses (Section 5). For this formalism we also provide:
  - A formulation of three versions of the static resolvability problem of increasing difficulty, and a decidable algorithm that is sound and complete for the first version, and sound for the second (Section 6).
  - A decidable algorithm that solves the dynamic resolvability problem for languages that have only well-balanced parentheses (Section 7).
- Two case studies of programming languages and the errors produced when we allow (resolvable) ambiguity:
  - A large subset of OCaml where some ambiguities have been reintroduced (Section 8.1).
  - A domain-specific language for cyber-physical systems (Section 8.2).

2 MOTIVATING EXAMPLE

This section walks through an example from OCaml where an ambiguity error can be clearer than what the actual compiler produces, suggesting that (resolvable) ambiguity has merit even in the context of a general purpose programming language.

Consider the following nested match expression (from [Palmkvist and Broman 2019]), and the resulting error message:

```
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 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 error message and solution is not particularly clear; surrounding an expression with parentheses does not change its type.

Instead, we look to the OCaml manual for inspiration. It contains an informal description of the syntax of the language[1], in the form of an EBNF-like grammar. Below is an excerpt of the productions for expressions, written in a more standard variant of EBNF:

⟨*expr*⟩ ::= 'match' ⟨*expr*⟩ 'with' ⟨*pattern-matching*⟩

⟨*pattern-matching*⟩ ::= '|'? ⟨*pattern*⟩ ('when' ⟨*expr*⟩)?
('|' ⟨*pattern*⟩ ('when' ⟨*expr*⟩)? '->' ⟨*expr*⟩)*

[1] https://caml.inria.fr/pub/docs/manual-ocaml/language.html

If we use this grammar to parse the nested match we find an ambiguity: the last match arm can belong to either the inner match or the outer match. The OCaml compiler makes an arbitrary choice to remove the ambiguity, which may or may not be the alternative the user intended. Presenting the two alternatives as parse trees would here be informative, and useful to a language designer, but would also expose implementation details to an end-user. Instead, we can present the alternatives in the form of modified code that parses unambiguously as each corresponding alternative:

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

3 PRELIMINARIES

This section briefly describes the theoretical foundations we build upon. Sections 3.3 and 3.5 describe context-free grammars and various forms of automata, the latter with some non-standard notation to make later sections easier to read. Section 3.6 then describes visibly pushdown languages, which enable the analyses described in Sections 6 and 7. Finally, Sections 3.7 and 3.4 describe trees and ambiguity, the former with some non-standard notation and the latter with a slightly wider definition than normal.

3.1 Bags

A *bag* (alternatively known as a *multiset*) is a generalization of a set, it is an unordered collection where each element may appear multiple times. The number of times an element a appears in a bag B is called the *multiplicity* of a in B and is written $m_B(a)$. A bag A is included in another bag B, written $A \subseteq B$, iff $\forall x.\ m_A(x) \leq m_B(x)$. The underlying set of elements of a bag B is called the *support* of B, and is given by $\mathit{Supp}(B) = \{x \mid m_B(x) > 0\}$.

3.2 Regular Expressions

A regular expression r over alphabet Σ is defined inductively:

$$r ::= \epsilon \mid a \mid r \cdot r \mid r + r \mid r^*$$

where $a \in \Sigma$. The language of a regular expression r is given by $L(r)$:

$$\begin{aligned} L(t) &= \{a\} \\ L(\epsilon) &= \{\epsilon\} \\ L(r_1 \cdot r_2) &= \{w_1 \cdot w_2 \mid w_1 \in L(r_1), w_2 \in L(r_2) \\ L(r_1 + r_2) &= L(r_1) \cup L(r_2) \\ L(r^*) &= L(\epsilon + r \cdot r^*) \end{aligned}$$

We will denote the set of regular expressions with alphabet Σ as $\mathit{Reg}(\Sigma)$.

3.3 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 $V \times (V \cup \Sigma)^*$[2], i.e., a set of productions; and $S \in V$ the starting non-terminal.

[2] Where * is Kleene-star.

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 $L(G)$[3].

3.4 Ambiguity

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

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

3.5 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 $a_0 a_1 \ldots a_n$ iff there is such a succesful run.

A deterministic finite automaton (DFA) has the same definition, except $\delta : Q \times \Sigma \longrightarrow Q$, 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] We will always name a regular expression r (possibly with a subscript) and a CFG G (possibly with a subscript), to lessen the risk of confusion

3.6 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 $s \in \Gamma$.
- $p \xrightarrow{i} q$, where $i \in \Sigma_i$.
- $p \xrightarrow{r,-s} q$, 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 partition names stem from their original use in program analysis, c is for *call*, i for *internal*, and r for *return*. We will instead primarily use Σ_c and Σ_r for balanced parentheses.

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. 2013], 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.

3.7 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 $N \longrightarrow x(r)$.

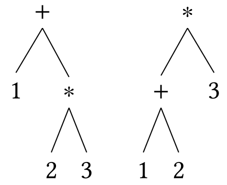


Fig. 1. The two allowable interpretations of $1 + 2 * 3$ in an arithmetic language without precedence.

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 $L(T)$[4].

Finally, $\mathit{yield} : L(T) \longrightarrow \Sigma^*$ is the sequence of terminals $a \in \Sigma$ obtained by a left-to-right[5] 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 some more standard concepts in formal languages.

Normally, we define a language as a set of words, i.e., a subset of Σ^* for some alphabet Σ, but here we additionally require the existence of a *meaning* of each word, in some sense. We will call this meaning an *interpretation* of a word. Note that a single word may have multiple interpretations. As such, we will define a language as:

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

For example, consider a simple arithmetic language without precedence and parentheses, where the interpretations are abstract syntax trees. In such a language, $\mathit{parse}(1 + 2 * 3)$ would produce a set containing the two ASTs in Figure 1:

We further classify each word $w \in \Sigma^*$ depending on the result of *parse*:

Definition 4.1. A word $w \in \Sigma^*$ is:

- not in the language if $\mathit{parse}(w) = \emptyset$.
- unambiguous if $\exists t \in T.\ \mathit{parse}(w) = \{t\}$.
- ambiguous if $\exists t_1, t_2 \in T.\ t_1 \neq t_2 \wedge \{t_1, t_2\} \subseteq \mathit{parse}(w)$.

We can connect these cases to the classical definition of a language:

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

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

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

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

Definition 4.2. A word $w \in \Sigma^*$ in a language given by $parse : \Sigma^* \rightarrow 2^T$ is resolvably ambiguous if $\forall t \in parse(w).\ \exists w' \in \Sigma^*.\ parse(w') = \{t\}$.

The example in Figure 1 is unresolvably ambiguous, since neither tree can be written in any other way. However, if we add grouping parentheses to the language we find that $1 + (2 * 3)$ and $(1 + 2) * 3$ are unambiguously interpreted as the first and second tree, respectively.

Additionally, we define a *language* to be resolvably ambiguous if all its words are resolvably ambiguous, or more formally:

Definition 4.3. A language given by $parse : \Sigma^* \rightarrow 2^T$ is resolvably ambiguous if $\forall w \in \Sigma^*.\ \forall t \in parse(w).\ \exists w' \in \Sigma^*.\ parse(w') = \{t\}$.

We can now state a few things:

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

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 with a particular formalism, introduced in Section 5, that gives us some decidable properties.

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

Static resolvability. Given a language defined by *parse*, determine if it is resolvably ambiguous.
Dynamic resolvability. Given a language defined by *parse* and a word w such that $parse(w) \neq \emptyset$, determine if w is resolvably ambiguous.

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

Definition 4.4. A decision procedure f (with possible outputs *resolvable*, *unresolvable*, and *unresolvable*) solving the static resolvability problem is:

- Sound if $f(parse) = resolvable$ implies that the language given by *parse* is resolvably ambiguous, and $f(parse) = unresolvable$ implies that the language given by *parse* is not resolvably ambiguous.
- Complete if $f(parse) \neq unknown$ for all possible inputs *parse*.

Similarly, for the dynamic problem:

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.

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

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.

Definition 4.5. A decision procedure f (with possible outputs *resolvable*, *unresolvable*, and *unresolvable*) solving the dynamic resolvability problem is:

- Sound if $f(parse, w) = resolvable$ implies that w is resolvably ambiguous, and $f(parse, w) = unresolvable$ implies w is not resolvably ambiguous.
- Complete if $f(parse) \neq unknown$ for all possible inputs *parse* and w.

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 interpretations to each word. The interpretations will be unranked trees, intended to be somewhat reminiscent of the abstract syntax trees used in most compilers. Section 4 suggests that *parse* discard some information, to enable the resolution of some ambiguities; we here choose to discard grouping parentheses.

With that in mind, we define a language definition D as a set of labelled productions, as described in Figure 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 here 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. This can be checked using, e.g., the approach described by Brabrand and Thomsen [2010]. In practice, this is likely desireable regardless, since a user is likely to attach different semantic meaning to each occurrence of a non-terminal.

Each non-terminal appearing in the regular expression of a production carry a *mark* m, which is a set of labels whose productions may *not* replace that non-terminal. To lessen clutter, we will write $E_\emptyset$ as E. As an example, consider the language definition in Figure 3, which will be used 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 interpretations, 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 parentheses and other forms of disambiguation (i.e., the result of marks).

$$\begin{array}{lll}
\hline
E & \longrightarrow & l\,(\,\texttt{'['}\ (\epsilon + E(\texttt{';'}\ E)^{*})\ \texttt{']'}\,) \\
E & \longrightarrow & a\,(\,E\ \texttt{'+'}\ E\,) \\
E & \longrightarrow & m(\,E\ \texttt{'*'}\ E\,) \\
E & \longrightarrow & n\,(\,N\,) \\
\hline
\end{array}$$

(a) T_D, the parse trees of G_D.

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

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

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

(c) G_D, the generated abstract syntax.

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

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

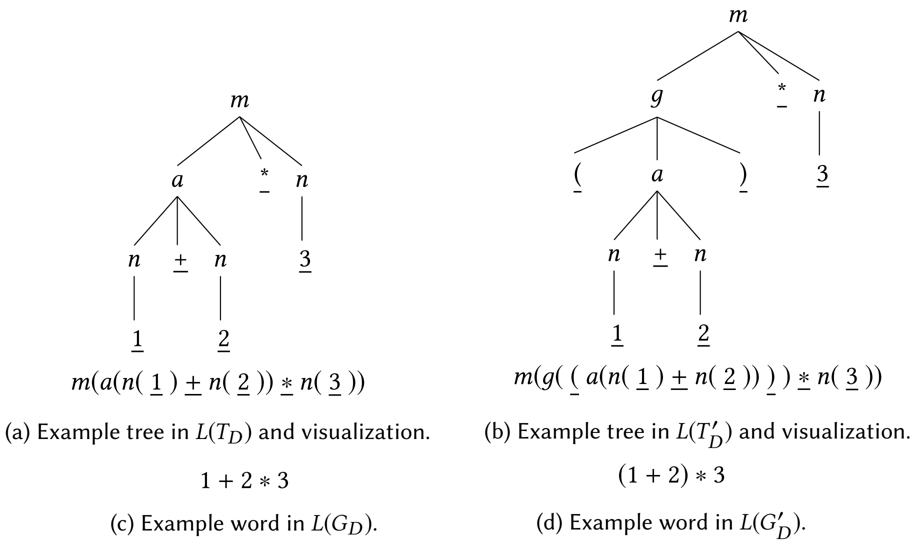
Fig. 4. The generated grammars.

- T'_D represents a tree language describing the parse trees of words in $L(G'_D)$.

Figure 4 contains the four grammars generated from our running example in Figure 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 Figure 4b has no production corresponding to the a production in Figure 3.

Examples of elements in each of these four languages can be seen in Figure 5, along with visualizations of the syntax trees. 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 prime 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 from D. We will generally refer to elements of $L(G_D)$ as w, $L(G'_D)$ as w', $L(T_D)$ as t, and $L(T'_D)$ as t'.

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 Figure 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



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

(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.

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

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

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 $unparen : L(T'_D) \longrightarrow L(T_D)$ 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 Figure 6. With this we can define $parse : L(G'_D) \longrightarrow 2^{L(T_D)}$, along with its inverse $words : L(T_D) \longrightarrow 2^{L(G'_D)}$:

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

The latter is mostly useful in later sections, but *parse* allow us to consider some concrete examples of resolvable and unresolvable ambiguities. For example, in our running example (Figure 3), the word `'1 + 2 + 3'` is ambiguous, since $parse($`'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 $parse($`'1 + (2 + 3)'`$) = \{t_1\}$ and $parse($`'(1 + 2) + 3'`$) = \{t_2\}$. To demonstrate the unresolvable case, we add the production $E \longrightarrow s : E$ `';'` E, at which point we find that the word `'[1 ; 2]'` is unresolvably ambiguous; $parse($`'[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 modify the language definition in Figure 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*(`'[1 ; 2]'`) $= \{t_3\}$.

We are now ready to construct decision procedures for the static and dynamic resolvability problems, as given in Definitions 4.4 and 4.5. Section 6 gives a partial solution to the former problem, while Section 7 fully solves the latter 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 interpretations instead of words:

Theorem 6.1. *A language given by parse* : $\Sigma^* \longrightarrow 2^T$ *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 $\Sigma \cap \{$ `'('`, `')'` $\} = \emptyset$ 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 $\Sigma \cap \{$ `'('`, `')'` $\} = \emptyset$.
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*(`'((1 + 2)) * 3'`) = *parse*(`'(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[6] 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. As such, going forward, we will assume no zero-length grouping parentheses.

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. *$\forall w'_1, w'_2 \in words(t).\ basic(w'_1) = basic(w'_2)$, i.e., all w' 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., two 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. *$\exists R$ such that $\exists w' \in words(t).\ rangebag(w') = R$ and $\forall w' \in words(t).\ R \subseteq Supp(rangebag(w'))$, i.e., some parentheses are required.*

For example, removing the parentheses in ”$(1+2)*3$” changes the interpretations produced. Each required range is a direct consequence of a mark on a non-terminal in D. We will write the word implied by the first quantified expression as $w'_\bot$. It is unique, since its basic word is fixed by t, and its rangebag is a set.

Lemma 6.4. *$\exists R$ such that $\exists w' \in words(t).\ rangebag(w') = R$ and $\forall w' \in words(t).$ $Supp(rangebag(w')) \subseteq R$, i.e., 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 the word implied by the first quantified expression $w'_\top$.

Lemma 6.5. *$\forall w'_1, w'_2 \in words(t).\ Supp(rangebag(w'_1)) = Supp(rangebag(w'_2)) \Rightarrow parse(w'_1) = parse(w'_2)$, i.e., duplicated parentheses do not matter.*

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

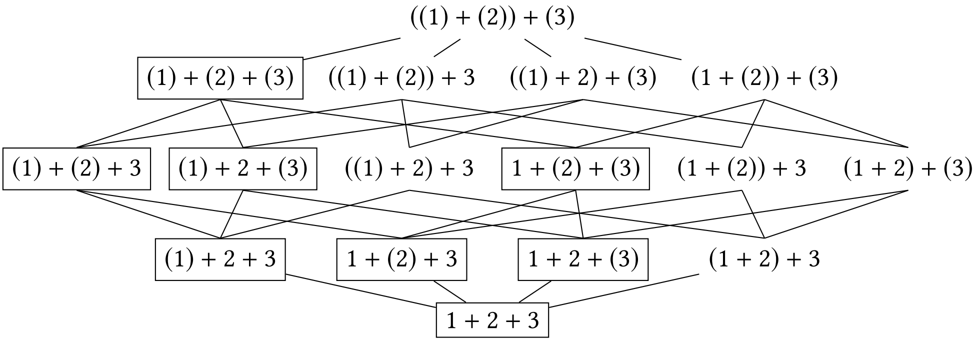


Fig. 7. The lattice of words for the tree $a(a(n(1) + n(2)) + n(3))$. The boxed words are shared with the lattice in Figure 8.

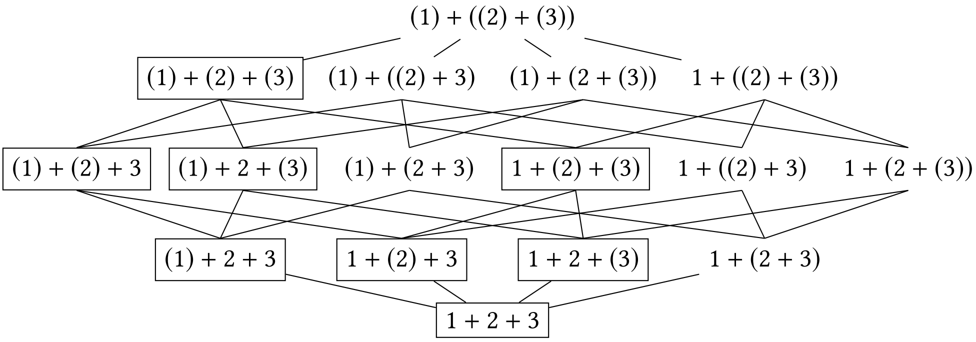


Fig. 8. The lattice of words for the tree $a(n(1) + a(n(2) + n(3)))$. The boxed words are shared with the lattice in Figure 7.

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 $\mathit{words}(t)$ for any given $t \in L(T_D)$; group words w' by their *rangeset*, where $\mathit{rangeset}(w') = \mathit{Supp}(\mathit{rangebag}(w'))$. 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 given by lemmas 6.4 and 6.3 respectively. Lemma 6.2 further states that all words in $\mathit{words}(t)$ share the same basic word. For example, Figure 7 contains the lattice for the tree $a(a(n(1) + n(2)) + n(3))$ (from our running example, defined in Figure 3 on page 8). 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 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* $t \in L(T_D)$, $w'_\top, w'_\bot \in words(t)$ *such that* $\forall w'.\ w' \in words(t) \Rightarrow rangebag(w'_\bot) \subseteq rangeset(w') \subseteq rangebag(w'_\top)$ *the following holds:*
$\forall w'.\ rangebag(w'_\bot) \subseteq rangeset(w') \subseteq rangebag(w'_\top) \Rightarrow w' \in words(t)$.

Lemmas 6.4 and 6.3 guarantee the existence of two such words $w'_\top$ and $w'_\bot$.

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,* $rangeset(w'_\bot) = \emptyset$ *for all trees, where* $w'_\bot$ *is given by Lemma 6.3.*

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* $t \in L(T_D)$ *with a lattice determined by* $w_\bot$ *and* $w_\top$, $\exists t' \in L(T_D).\ basic(w_\bot) = basic(w'_\bot) \land rangeset(w_\bot) \subseteq rangeset(w'_\bot) \land rangeset(w'_\top) \subseteq rangeset(w_\top) \Rightarrow \neg\exists w.\ parse(w) = \{t\}$.

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

Theorem 6.9. *In version 1, given a* $t \in L(T_D)$ *with a lattice determined by* $w_\bot$ *and* $w_\top$, $\exists t' \in L(T_D).\ basic(w_\bot) = basic(w'_\bot) \land rangeset(w'_\bot) \subseteq rangeset(w_\bot) \land rangeset(w_\top) \subseteq rangeset(w'_\top) \Longleftrightarrow \neg\exists w.\ parse(w) = \{t\}$.

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

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

6.3 A Lattice as a Word, and an Algorithm

This section introduces a linear encoding of the lattices of the previous chapter 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 Figure 8 is represented by ”[[1] + [[2] + [3]]]”, while the tree for ”(1 + 2) ∗ 3” has a lattice represented by ”[([1] + [2]) ∗ [3]]”. 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, e.g., ”[[1]]” is uniquely represented as ”[1]” while ”([2])” is uniquely represented as ”(2)”.

Lattice equality is now equvialent 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 strict superset) by adding a pair of possible parentheses, and move the bottom ”down” (new bottom rangeset is a strict 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.
- 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.

We will now walk through the construction of this automaton, using the following language definition:

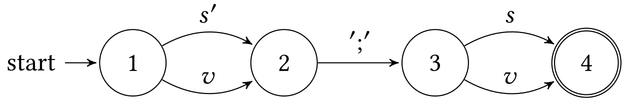
$$\begin{array}{llll} E & \longrightarrow & s: & E_{\{s\}}\ \texttt{';'}\ E \\ E & \longrightarrow & v: & \texttt{'e'} \end{array}$$

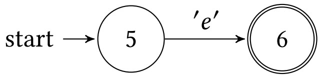
We first present a simplified construction that assumes no production can match a single non-terminal, i.e., for all right-hand side regular expressions r, $\neg\exists N_m.\ N_m \in L(r)$, to make the base idea easier to follow. Such cases introduce duplicated parentheses if handled in a naive way, 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 child 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 in m is distinguished by a prime, e.g., $E_{\{s\}}$ is replaced with $s' + v$.

$$\begin{array}{llll} E & \longrightarrow & s: & (s' + v)\ \texttt{';'}\ (s + v) \\ E & \longrightarrow & v: & \texttt{'e'} \end{array}$$

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.



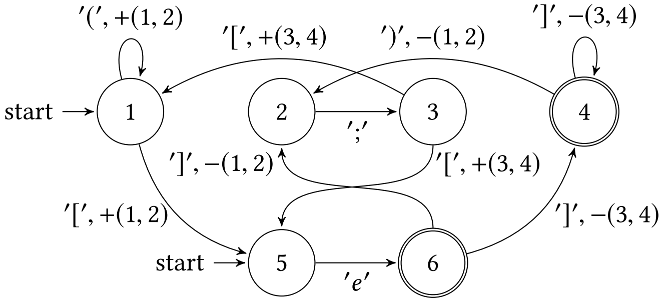


We then combine them into a single visibly pushdown automaton. Every time we transition from a production to a child we push a symbol to the stack, which we then pop when we go back. We use $Q \times Q \times \{"()", "[]"\}$ as our stack alphabet, where the first two components are states in the parent production automaton and the last component records whether the parenthesis pair is required or merely possible. We replace every edge $p \xrightarrow{l} q$, where $l \in L$, with:

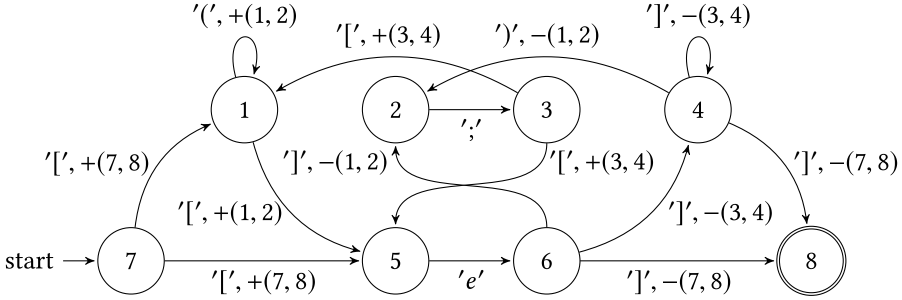
- an edge $p \xrightarrow{'[', +(p,q,"[]")} p'$, where p' is the initial state in the DFA corresponding to the production with label l, and
- an edge $q' \xrightarrow{']', -(p,q,"[]")} q$ for every final state q' in the DFA corresponding to the production with label l.

Similarly, for every edge $p \xrightarrow{l'} q$, where $l \in L$, we add edges $p \xrightarrow{'(', +(p,q,"()")} p'$ and $q' \xrightarrow{')', -(p,q,"()")} q$.

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. To lessen clutter in the figure below we omit the third component of the stack symbol, since it is fully determined by the input symbol, e.g., if an edge reads '[' then the third component of the stack symbol pushed must be "[]".



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



The resulting pushdown automaton has only two sources of non-determinism: the edges labelled $'['$ and $'('$. Each one corresponds to one of the allowable child productions at that point in the tree, thus there is one run per tree.

UNFINISHED

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. Additionally, while Section 6 requires a language definition D to not contain parentheses, this section merely requires parentheses to be balanced.

Our approach centers around around *words*(t), which, being a set of words, can be considered a language in the classical sense. Each such language turns out to be a visibly pushdown language. Given a set of trees in $L(T_D)$ we can construct a corresponding set of non-overlapping visibly pushdown automata (i.e., each automaton only accepts words not accepted by any other automaton) since VPLs are closed under complement and intersection [Alur and Madhusudan 2004]. These automata can then be examined to determine if they accept the empty language (which implies that the corresponding tree only has ambiguous words), or otherwise produce a witness, a word accepted only by that automaton.

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

UNFINISHED

8 EVALUATION

8.1 OCaml

8.2 Cyphym

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 discussed in Section 8.1.

UNFINISHED

10 CONCLUSION

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 edition 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.* Theses. 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. 2013. Trimming Visibly Pushdown Automata. In *Implementation and Application of Automata (Lecture Notes in Computer Science)*, Stavros Konstantinidis (Ed.). Springer Berlin Heidelberg, 84–96.

H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 2007. Tree Automata Techniques and Applications. Available on: http://www.grappa.univ-lille3.fr/tata. (2007). release October, 12th 2007.

Keith Cooper and Linda Torczon. 2011. *Engineering a Compiler* (2n edition 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

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.

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.

Elizabeth Scott and Adrian Johnstone. 2010. GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (Sept. 2010), 177–189. https://doi.org/10.1016/j.entcs.2010.08.041

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

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

Daniel H. Younger. 1967. Recognition and Parsing of Context-Free Languages in Time N3. *Information and Control* 10, 2 (Feb. 1967), 189–208. https://doi.org/10.1016/S0019-9958(67)80007-X