Resolvable Ambiguity

Viktor Palmkvist*
Position1
Department1
KTH Royal Institute of Technology
Stockholm, State1, Sweden
vipa@kth.se

First2 Last2†
Position2a
Department2a
Institution2a
City2a, State2a, Country2a
first2.last2@inst2a.com
Position2b
Department2b
Institution2b
City2b, State2b, Country2b
first2.last2@inst2b.org

Abstract

Text of abstract ....

Keywords keyword1, keyword2, keyword3

1 Introduction

Text of paper ...

1.1 Motivating Ambiguity in Programming Languages

Consider the following nested match expression in OCaml:

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

The OCaml compiler, when presented with this code, will give a type error for the last line:

```
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 fix is simple; put parentheses around the inner match:

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

The connection between the error message and the fix is not a clear one however; adding parentheses around an expression does not change the type of anything.

To come up with an alternative error to present in this case 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*⟩ '->' ⟨*expr*⟩)+

Note that ⟨*pattern-matching*⟩ is slightly simplified, the original grammar supports **when** guards and makes the first '|' optional. 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.

We instead argue that the grammar should be left ambiguous for this sort of corner cases that are likely to trip a user, allowing the compiler to present an ambiguity error, which lets the user select the intended alternative.

1.2 Unresolvable Ambiguity

Unfortunately, not all ambiguities can be resolved by adding parentheses. Again, looking to the informal OCaml grammar:

⟨*expr*⟩ ::= ⟨*expr*⟩ ';' ⟨*expr*⟩
| '[' ⟨*expr*⟩ (';' ⟨*expr*⟩)* ';'? ']'
| ⟨*constant*⟩

The first production is sequential composition, the second is lists (the empty list is under ⟨*constant*⟩). Now consider the following expression: "`[1; 2]`".

We find that it is ambiguous with two alternatives:

1. A list with two elements.
2. A list with one element, namely a sequential composition.

*with author1 note
†with author2 note

PL'18, January 01–03, 2018, New York, NY, USA
2018.

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

Terminals	$t \in \mathbb{T}$
Non-terminals	$N \in \mathbb{N}$
Identifiers	$i \in \mathbb{I}$
Regular expressions	$r ::= t \mid N \mid r \cdot r \mid r + r \mid \epsilon \mid r^*$
Productions	$i : N \to r$

Figure 1. Context-free EBNF grammars

We can select the second option by putting parentheses around "1 ; 2", but there is no way to select the first. If the user intended the first option we have a problem: we can present an accurate error message, but there is no way for an end-user to solve it; it requires changes to the grammar itself.

To prevent the possibility of an end-user encountering such an error we must ensure that the grammar cannot give rise to an unresolvable ambiguity. It is worth mentioning here that statically checking if a context-free grammar is ambiguous has long been known to be undecidable. Unresolvable ambiguity, however, turns out to be decidable[2].

Note that, as for ambiguity, the shape of the grammar is important, since the property considers parse trees rather than merely words. For this paper, we consider context-free grammars with EBNF operators.

1.3 Contributions

- Building on [2], which merely isolates ambiguities, an algorithm that suggests solutions to ambiguity errors.
- A formalization of the unresolvable ambiguity property for context-free EBNF grammars.
- An algorithm for deciding if a grammar is unresolvably ambiguous or not.

2 Preliminaries

> **NOTE:** The text hereafter is written a bit later than the text before, and is ever so slightly inconsistent with it.

A context-free EBNF grammar is a tuple $(S, P, \mathbb{N}, \mathbb{T}, \mathbb{I})$. $S \in \mathbb{N}$ is the starting symbol, P is a set of productions, as given in Figure 1, $\mathbb{N}$, $\mathbb{T}$, and $\mathbb{I}$ are sets of non-terminals, terminals, and identifiers, respectively. Additionally, we require that $\mathbb{N} \cap \mathbb{T} = \emptyset$, and that the identifiers uniquely identify each production, i.e., there are no two distinct productions in P with the same identifier.

The (word) language of a given non-terminal N in a grammar $G = (S, P, \mathbb{N}, \mathbb{T}, \mathbb{I})$ is given by $L_G(N)$:

$$
\begin{aligned}
L_G(t) &= \{t\} \\
L_G(N) &= \bigcup \{L_G(r) \mid (N \to r) \in P\} \\
L_G(r_1 \cdot r_2) &= \{w_1 \cdot w_2 \mid w_1 \in L_G(r_1), w_2 \in L_g(r_2)\} \\
L_G(r_1 + r_2) &= L_G(r_1) \cup L_G(r_2) \\
L_G(\epsilon) &= \{\epsilon\} \\
L_G(r^*) &= \{\epsilon\} \cup L_G(r) \cup L_G(r \cdot r) \cup \ldots
\end{aligned}
$$

The word language of a grammar G, written $L(G)$, is thus $L_G(S)$. We will omit the subscript whenever the intended grammar is clear from context.

Note that the right hand side of a production is a regular expression, which could potentially be ambiguous in and of itself. For this work we assume the rhs regular expressions to either be unambigous, or that any remaining ambiguity is unimportant. This can be achieved in a number of ways, for example using the ambiguity checking of Brabrand and Thomsen [1].

As such, for a given grammar $G = (S, P, \mathbb{N}, \mathbb{T}, \mathbb{I})$ we construct a linear representation of its parse trees as another grammar $T_G = (S, P', \mathbb{N}, \mathbb{T}', \mathbb{I})$ where:

$$
\begin{aligned}
P' &= \{i : N \to [_i r]_i \mid (i : N \to r) \in P\} \\
\mathbb{T}' &= \mathbb{T} \sqcup \textstyle\bigcup_{(i:N \to r) \in P} \{[_i,]_i\}
\end{aligned}
$$

where $\sqcup$ denotes disjoint union. Intuitively, we surround the right hand side of each production with a unique pair of brackets, signifying the production that was used for the parse. Again, we will omit the subscript whenever the intended grammar is clear from context.

The *yield* of a parse tree is the word it parsed, i.e., $yield : L(T) \to L(G)$. Intuitively, *yield* removes the brackets introduced when constructing T.

A given word $w \in L(G)$ is ambiguous iff:

$$\exists t_1, t_2 \in L(T).\ yield(t_1) = w \wedge yield(t_2) = w \wedge t_1 \neq t_2$$

A grammar is ambiguous iff it contains at least one ambiguous word.

3 Parse-time Disambiguation

We begin this section with some motivation, and then list our definition of *resolvable ambiguity*.

We can divide the productions present in a programming language grammar in two groups: those that are semantically important, and those that are semantically *un*important. The former group covers most productions, while the latter contains, e.g., parentheses used for explicit grouping. If programs were written directly as syntax trees then the latter would be unnecessary; two syntax trees that differ only by parentheses are semantically the same.

Thus we wish the output of parsing to be a syntax tree consisting entirely of semantically important productions. This distinction is useful to make, because it allows us to decouple *what* we want to be expressible and *how* it is to

[2] With some caveats, I'll talk more about this during the meeting.

be expressed. As an example, in Figure 2, 2a is the *what* and 2b is the *how*. The latter grammar is a modification of the former that adds precedence, associativity, and parentheses, yielding an unambiguous grammar with at least one way to express each semantically distinct tree.

Our definition thus refers to four grammars in total:

- The semantic grammar G, containing only the semantically important productions (e.g., Figure 2a).
- T_G (generally abbreviated as T), the parse trees of G, representing the trees that must be expressible (Figure 2d is in this language).
- The parse grammar G', a modification of G meant to actually be used for parsing (e.g., Figure 2b). The details of these modifications will be covered later.
- $T_{G'}$ (generally abbreviated as T'), the parse trees of G' (Figure 2c is in this language).

We also require a function $semantic : L(T') \to L(T)$ that removes the semantically unimportant productions from a parse tree. The relation between the four grammars can be seen in Figure 3. Finally, we define the function $parse : L(G') \to 2^{L(T)}$:

$$parse(w') = \{semantic(t') \mid t' \in L(T') \wedge yield(t') = w'\}$$

Definition 3.1. A language defined by the semantic grammar G and the parse grammar G' is resolvable if:

$$\forall t \in L(T).\ \exists w' \in L(G').\ parse(w') = \{t\}$$

Intuitively, a language is resolvably ambiguous if there is at least one (unambiguous[3]) word for each semantically distinct tree. Note that neither G nor G' necessarily need to be unambiguous for this to hold.

4 Static Resolvability Check

This section describes a decision algorithm for detecting unresolvable ambiguities, starting with a version with several limitations, most of which are later lifted.

4.1 Basic Algorithm

Our initial limitations / assumptions are as follows:

1. G contains no parentheses, i.e., given $G = (S, P, \mathbb{N}, \mathbb{T}, \mathbb{I})$ we require $\mathbb{T} \cap \{\text{'('}, \text{')'}\} = \emptyset$.
2. G' is constructed by adding parentheses to all non-terminals of G, i.e., $G' = (S, P', \mathbb{N}, \mathbb{T} \cup \{\text{'('}, \text{')'}\}, \mathbb{I} \cup \mathbb{N})$ where $P' = P \cup \{N : N \to \text{'('} N \text{')'} \mid N \in \mathbb{N}\}$.
3. No production has a right-hand side that matches a single non-terminal, i.e., $\forall (i : N \to r) \in P.\ \mathbb{N} \cap R(r) = \emptyset$[4].

[3] For a slightly different meaning of unambiguous, it's here relating G' to T, instead of G' to T'.

[4] R is here the more traditional language of a regular expression, i.e., $R(r) \subseteq (\mathbb{T} \cup \mathbb{N})^*$. It should be written more explicitly somewhere, but I'm lazy at the moment.

⟨*expr*⟩ ::= Sum: ⟨*expr*⟩ '+' ⟨*expr*⟩
| Product: ⟨*expr*⟩ '*' ⟨*expr*⟩
| Number: ⟨*number*⟩

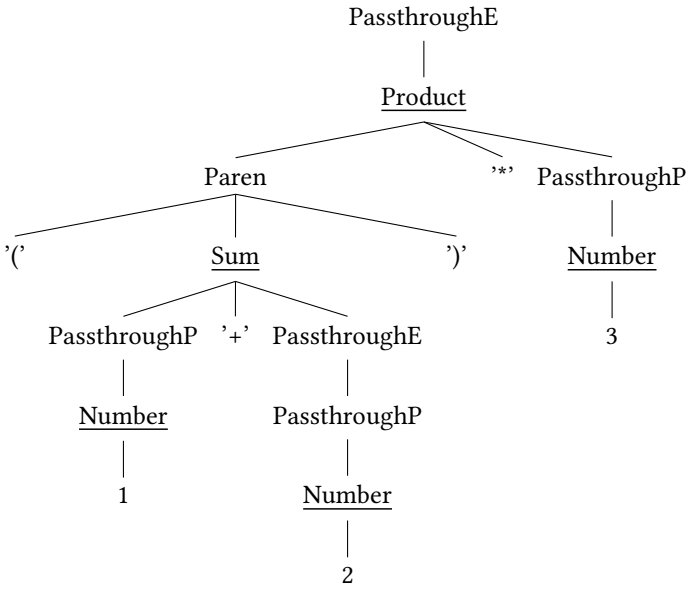
(a) The (ambiguous) intuitive grammar without parentheses.

⟨*expr*⟩ ::= Sum: ⟨*product*⟩ '+' ⟨*expr*⟩
| PassthroughE: ⟨*product*⟩

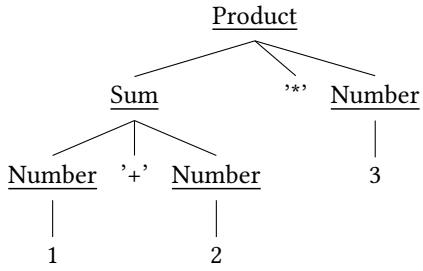
⟨*product*⟩ ::= Product: ⟨*atom*⟩ '*' ⟨*product*⟩
| PassthroughP: ⟨*atom*⟩

⟨*atom*⟩ ::= Paren: '(' ⟨*expr*⟩ ')'
| Number: ⟨*number*⟩

(b) The (unambiguous) grammar with parentheses.



(c) Parse tree for $(1 + 2) * 3$. The underlined nodes are semantically important.



(d) The same parse tree, after removing the semantically unimportant nodes.

Figure 2. A basic expression grammar in two variations, and two example syntax trees.

$$\begin{array}{ccc} L(G) & \xleftarrow[yield]{} & L(T) \\ & & \uparrow{\scriptstyle semantic} \\ L(G') & \xleftarrow[yield]{} & L(T') \end{array}$$

Figure 3. The grammars considered, and their relation to each other. G is provided by a user of the system, along with instructions how to modify G to construct G', while T and T' are automatically derived.

We are looking for a counterexample to the resolvable property, i.e., a $t \in L(T)$ for which $\neg\exists w' \in L(G')$. $parse(w') = \{t\}$. By the construction of G', we have $t \in parse(yield(t))$, thus there is at least one (potentialy ambiguous) word w' that can be parsed as t. Our task is thus to find a tree t such that $\forall w' \in L(G')$. $t \in parse(w') \to \{t\} \subset parse(w')$, i.e., a tree that can only be parsed with ambiguous words.

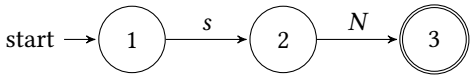
We begin by noting that parentheses cannot be added just anywhere; they must correspond to a node in the parse tree. For example, for the tree in Figure 2d (which could be parsed from "1 + 2 ∗ 3") it would be valid to add parentheses around "1 + 2" but not "2 ∗ 3". As a consequence, adding parentheses restricts the possible parse trees, i.e., given two words w_1' and w_2' where the latter has added some parentheses we have $parse(w_1') \supseteq parse(w_2')$. We also note that adding double parentheses imposes no additional restriction, e.g., $parse$("((1 + 2)) ∗ 3") = $parse$("(1 + 2) ∗ 3").

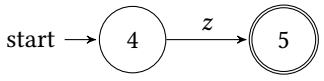
There is thus a "most restrictive word" for every tree. If this word is ambiguous, then we have our counterexample.

We will now construct a pushdown automaton that recognizes these most restrictive words in such a way that there is a bijection between successful runs and trees in $L(T)$. As a running example, we will use the following (very simple) grammar G:

⟨N⟩ ::= succ: 's' ⟨N⟩
| zero: 'z'

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

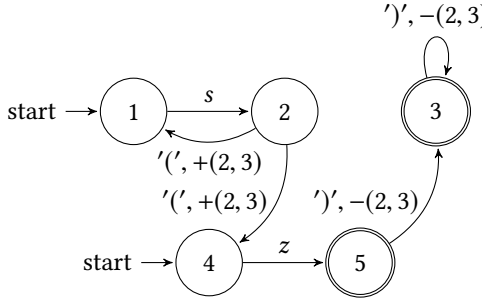




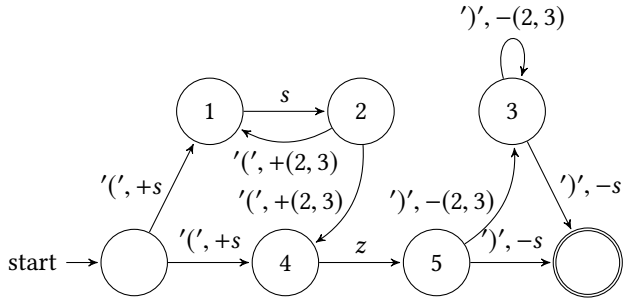
We then combine them into a single pushdown automata by replacing each edge with a non-terminal label $p \xrightarrow{N} q$ with:

- an edge $p \xrightarrow{'(',+(p,q)} p'$ for every initial state p' in some DFA belonging to non-terminal N, and
- an edge $q' \xrightarrow{')',-(p,q)} q$ for every final state q' in some DFA belonging to non-terminal N,

were $+\gamma$ means "push γ", while $-\gamma$ means "pop γ". Intuitively, parse a child node, but put parentheses around it.

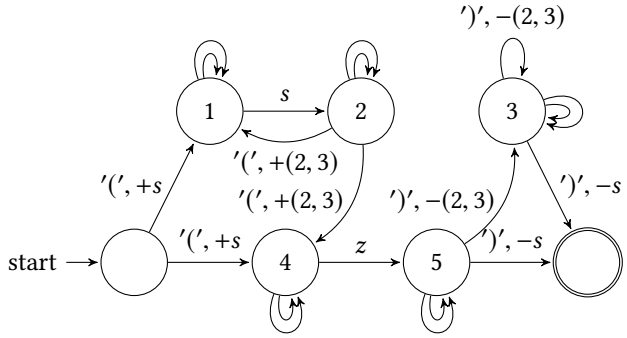


Finally, we add a new initial state and a new final state, and connect them with the initial and final states belonging to the starting non-terminal:



The resulting pushdown automaton has only a single source of non-determinism: the edges labelled `'('`. Each one corresponds to one of the allowable child productions at that point in the parse tree.

We now have a pushdown automaton that recognizes the "most restrictive word" for each tree in $L(T)$, which we will call $A_($. Next we need to determine if it is a valid word for some other tree as well. To do this we create a copy of $A_($ with one modification: for every state s in $A_($ (except the initial and final states), add two transitions $s \xrightarrow{'(',+p} s$ and $s \xrightarrow{')',-p}$. To avoid cluttering the graph too much, these transitions are shown unlabeled below:



We will call this automaton $A_('$. Intuitively, this automaton recognizes a most restrictive word, but allows the addition

of arbitrary well-nested parentheses. Successful runs in this automaton have a surjection to runs in $A_($ (ignore transitions along the newly added edges), and thus also have a surjection to parse trees in $L(T)$. We can also note that every sucessful run in $A_($ is also a successful run in $A'_($, since the latter has all states and transitions of the former. Furthermore, two distinct runs, one in $A_($ (call it p) and one in $A'_($ (call it p'), that both recognize the same word must represent different parse trees in $L(T)$. To see why, we consider two cases:

1. p' only uses transitions present in $A_($. This is a successful run in $A_($, and distinct from p. But there is a bijection between runs in $A_($ and parse trees in $L(T)$, thus p' represents a different parse tree.
2. p' uses at least one transition added in $A'_($. Using the surjection between runs in $A'_($ and $A_($ we find a new successful run that produces a different word (at least one fewer pair of parentheses). Since this run produces a different word, it must be distinct from p, and thus represent a different parse tree.

References

[1] Claus Brabrand and Jakob G. Thomsen. [n. d.]. 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* (2010) *(PPDP '10)*. ACM, 243–254. https://doi.org/10.1145/1836089.1836120

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