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"
```

*with author1 note
†with author2 note

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:

$\langle expr \rangle$ ::= 'match' $\langle expr \rangle$ 'with' $\langle pattern\text{-}matching \rangle$

$\langle pattern\text{-}matching \rangle$ ::= ('|' $\langle pattern \rangle$ '->' $\langle expr \rangle$)+

Note that $\langle pattern\text{-}matching \rangle$ 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:

$\langle expr \rangle$ ::= $\langle expr \rangle$ ';' $\langle expr \rangle$
 | '[' $\langle expr \rangle$ (';' $\langle expr \rangle$)* ';'? ']'
 | $\langle constant \rangle$

The first production is sequential composition, the second is lists (the empty list is under $\langle constant \rangle$). 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.

[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	$N \longrightarrow i : r$

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 [4], 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 before this section is largely out of date.

2.1 Context-Free Grammars

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 $\mathbb{N} \cap \mathbb{T} = \emptyset$. Most (though not all) grammars we will consider will have unique identifiers, i.e., there are no two distinct productions in P with the same identifier. As an example, Figure 2a contains a small grammar. The first production listed has the identifier "Sum" and consists of an expression, then a literal "+", then another expression.

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

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

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 [2].

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:

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

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* : $L(T) \longrightarrow L(G)$. 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.

2.2 Automata

A nondeterministic finite automaton (NFA) is a tuple $(Q, \Sigma, \Delta, q_0, F)$:

- A finite set of states Q.
- A finite set of input symbols Σ, i.e., an input alphabet.
- A transition function $\delta : Q \times \Sigma \longrightarrow 2^Q$.
- An initial state $q_0 \in Q$.
- A set of final states $F \subseteq Q$.

A successful run is a sequence of states $r_0, r_1, \dots, r_n$ and a word $a_0 a_1 \dots a_n$ such that:

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

We say that the automaton accepts the word $a_0 a_1 \dots a_n$ iff there is such a succesful run. A particular state $q \in Q$ is *reachable* if there is a (not necessarily successful) run starting

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

in q_0 and ending in q. q is *coreachable* if there is a run starting in q and ending in some $f \in F$.

A deterministic finite automaton (DFA) has the same definition, except $\delta : Q \times \Sigma \to Q$.

A pushdown automaton extends a finite automaton with a stack, and lets each transition push or pop symbols from it. Formally, a pushdown automaton is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$:

- A finite set of states Q.
- A finite set of input symbols Σ, i.e., an input alphabet.
- A finite set of stack symbols Γ, i.e., a stack alphabet.
- A transition function $\delta : Q \times \Sigma \times \Gamma \to 2^{Q \times \Gamma *}$.
- An initial state $q_0 \in Q$.
- A set of final states $F \subseteq Q$.

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 $\gamma \in \Gamma*$. Reachable and coreachable are now defined on configurations, rather than states.

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.
- $p \xrightarrow{a,+g} q$, a transition that recognizes the terminal a and pushes the symbol g on the stack.
- $p \xrightarrow{a,-g} q$, a transition that recognizes the terminal a and pops the symbol g from the stack (i.e., this transition cannot be taken if g is not on top of the stack).

2.3 Visibly Pushdown Languages

A visibly pushdown language [1] 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 $c \in \Sigma_c$.
- $p \xrightarrow{i} q$, where $i \in \Sigma_i$.
- $p \xrightarrow{r,-s} q$, where $r \in \Sigma_r$.

i.e., the terminal recognized by a transition fully determines the change to the stack height.

This gives us some nice properties. Of particular relevance to this paper are the following two points:

- If two visibly pushdown automata have the same partitions Σ_c, Σ_i, and Σ_r, then we can construct a product automaton that simulates two simultaneous runs through both automata. This product automaton has the same input alphabet partitions, each state is a pair of states (one from each automaton), and each stack symbol is a pair of stack symbols (one from each automaton).
- A visibly pushdown automaton can be trimmed [3], i.e., modified such that all reachable configurations are also coreachable.

3 Parse-time Disambiguation

We begin this section with some motivation, then list our definition of *resolvable ambiguity* and what disambiguation we will consider, and finally an alternative definition of a word, which will be useful in later sections.

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 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 identifiers in this grammar need not be unique, but should be a superset of the identifiers in G. The modifications we will consider in this paper are introduced in Section 3.1.
- $T_{G'}$ (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* : $L(G') \to 2^{L(T)}$, and its inverse *words* : $L(T) \to 2^{L(G')}$:

$$\begin{array}{rcl} \mathit{parse}(w') & = & \{\mathit{semantic}(t') \mid t' \in L(T') \wedge \mathit{yield}(t') = w'\} \\ \mathit{words}(t) & = & \{w' \mid t \in \mathit{parse}(w')\} \end{array}$$

The latter will be useful later, while *parse* is used directly in the definition of resolvable ambiguity:

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

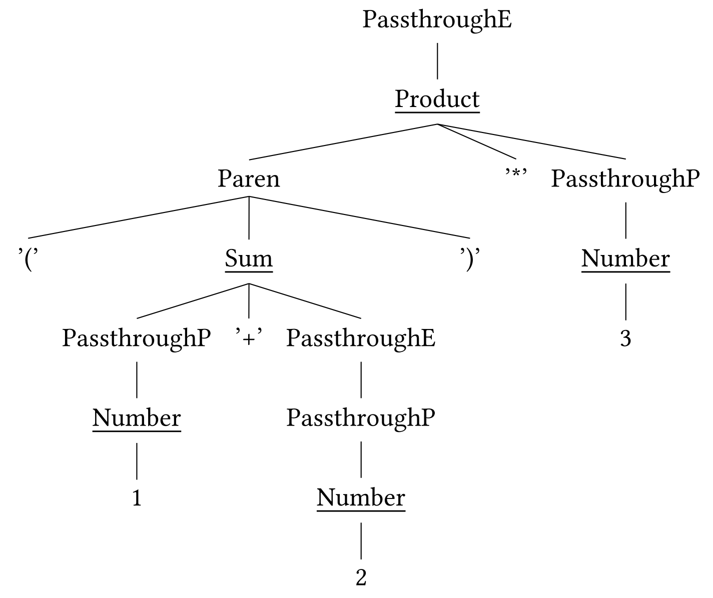
(a) The (ambiguous) intuitive grammar without parentheses.

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

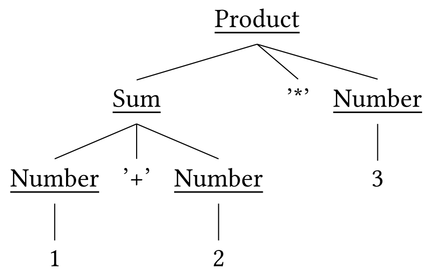
⟨*product*⟩ ::= Product: ⟨*atom*⟩ '*' ⟨*product*⟩
| PassthroughP: ⟨*atom*⟩

⟨*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.

$$\begin{tikzcd}
L(G) & L(T) \arrow[l, "yield"'] \\
L(G') & L(T') \arrow[l, "yield"'] \arrow[u, "semantic"']
\end{tikzcd}$$

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.

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

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

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.

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

3.1 Modifications in G'

For this paper we consider two possible modifications:

- Adding parentheses for explicit grouping.
- Forbidding non-terminals in certain positions from expanding using certain productions.

The latter requires more explanation. As an example, in the following grammar we are forbidding the second ⟨<*expr*>⟩ in the first production from expanding using the "Sum" production:

⟨*expr*⟩ ::= Sum: ⟨*expr*⟩ '+' $\langle expr\rangle_{Sum}$
| Number: ⟨*number*⟩

Transforming this into a normal grammar, and adding parentheses, we get the following:

⟨*expr*⟩ ::= Sum: ⟨*expr*⟩ '+' $\langle expr\rangle_{Sum}$
| Number: ⟨*number*⟩
| Paren: '(' ⟨*expr*⟩ ')'

$\langle expr_{Sum}\rangle$ ::= Number: ⟨*number*⟩
| Paren: '(' ⟨*expr*⟩ ')'

In other words, we produce a grammar where addition is left-associative[4], but we can use explicit grouping via parentheses to override that.

3.2 An Alternative Word View

This section introduces an alternative (isomorphic) definition of a word that will be used to motivate the correctness of later algorithms. The modifications permitted by the previous section significantly limit how G' can differ from G. 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[5] 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 ("$1+2*3$", {1–3}), while "$((1+2))*3$" is equivalent to ("$1+2*3$", {1–3, 1–3}). The first component will be referred to as a *basic word*, the second as a *range bag*

We will now note a few things about the words $w' \in$ *words*(t) for some given $t \in L(T)$:

- All w' have the same basic word. Limiting the modifications to those in the previous section means that they can only differ in parentheses. This also implies that two trees that share a word (i.e., that can be ambiguous) must also share a basic word.
- Some parentheses are required, i.e., some ranges must be present in all w'. For example, removing the parentheses in "$(1 + 2) * 3$" changes the tree produced, since multiplication has higher precedence than addition.
- There is a finite set of possible ranges in the range bags of w'. Grouping parentheses can only be added if they exactly cover a node in the parse tree, and other parentheses can only be added where G allows them.
- Duplicated grouping parentheses do not matter. For example, $((1 + 2)) * 3$ permits the same syntax trees as $(1 + 2) * 3$. If there are no parentheses present in G then all parentheses are grouping parentheses, thus we can consider the range bag as a *set* instead.

4 Static Resolvability Check

This section describes a decision algorithm for detecting unresolvable ambiguities in a pair of grammars G and G', starting with a version with several limitations, most of which are later lifted. The overarching goal is to find a tree $t \in L(T)$ such that there is no $w' \in L(G')$ for which $parse(w') = \{t\}$. Alternatiely, find a tree that has no unambiguous words.

[4]I'm having some issues typesetting this appropriately, and I'm not all that happy with the explanation.

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

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 $\mathbb{T} \cap \{\texttt{'('}, \texttt{')'}\} = \emptyset$. This implies that all parentheses in G' are grouping parentheses, thus we only need to consider words $w' \in L(G')$ whose range bag is a set.
2. G' is constructed by adding parentheses to all non-terminals of G, with no other modifications, i.e., $G' = (S, P', \mathbb{N}, \mathbb{T} \cup \{\texttt{'('}, \texttt{')'}\}, \mathbb{I} \cup \mathbb{N})$ where $P' = P \cup \{N : N \rightarrow \texttt{'('}N\texttt{')'} \mid N \in \mathbb{N}\}$. This implies that there are no required grouping parentheses.
3. No production has a right-hand side that matches a single non-terminal, i.e., $\forall (i : N \rightarrow r) \in P. \, \mathbb{N} \cap R(r) = \emptyset$[6].

The first two limitations allow us to place all words in *words*(t) for some given t in a lattice, whose structure is formed by the subset ordering of the range sets. For example, the tree in Figure 2d (but where G' has no associativity or precedence) has the lattice of words seen in Figure 4 (omitting the pair of parentheses around the entire word, since they are always permissible).

The bottom word corresponds to the empty range set, while the top word corresponds to the set of possible parentheses ranges. All trees that can be ambiguous must thus share the same bottom word, while the top may differ.

Of particular use is to examine whether the top word (call it w') is ambiguous. There are two cases:

- The top word is unambiguous, then this is not a tree we are looking for.
- It is ambiguous. That means that there is some other lattice for some other tree that also contains the word. Call the top word of this other lattice w'_2. For w' to be in the lattice whose top is w'_2 its rangeset must be a subset of the rangeset of the latter. But this is true also for all the other words in the lattice, thus there are no unambiguous words in the lattice, i.e., this tree is what we are looking for.

The algorithm thus looks for two trees, where the set of possible parentheses for one is a superset of the set of possible parentheses for the other. To do this, we use a linear representation of each lattice: namely the top word. If two trees have the same top word, that implies that their sets of possible parentheses are equal. If we can add parentheses to one word and get the other word, that implies that the rangeset of the former is a subset of the latter. that term later

We will now construct a pushdown automaton that recognizes these top words in such a way that there is a bijection between successful runs and trees in $L(T)$. As a running

[6]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.

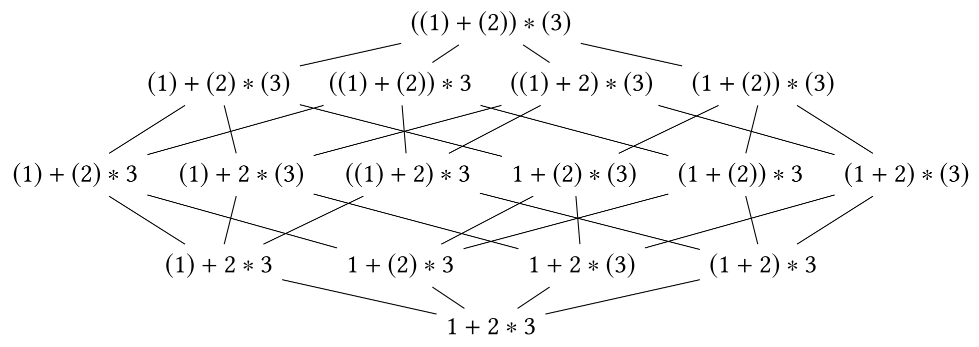
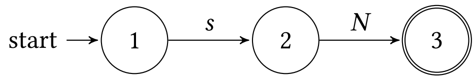


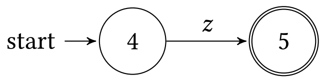
Figure 4. The lattice of words for the tree in Figure 2d, assuming no precedence in G'.

example, we will use the following (very simple) grammar G:

$\langle N \rangle$ ::= succ: 's' $\langle N \rangle$
| 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.

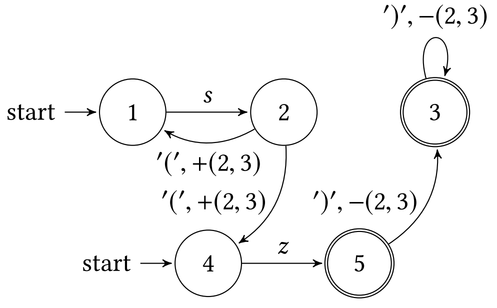




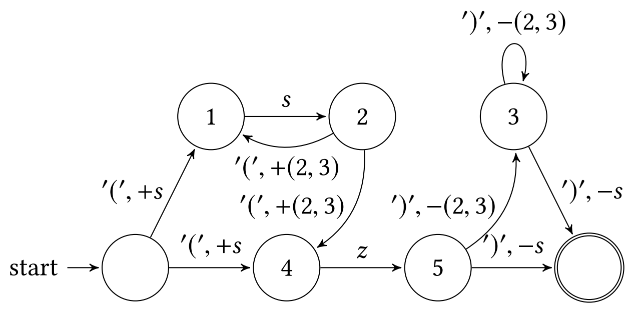
We then combine them into a single pushdown automata by replacing each edge with a non-terminal label $p \xrightarrow{N} q$ with:

- 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,

Intuitively, we parse a child node, but put parentheses around it. This is where we use assumption 3, without it we might introduce double parentheses here.

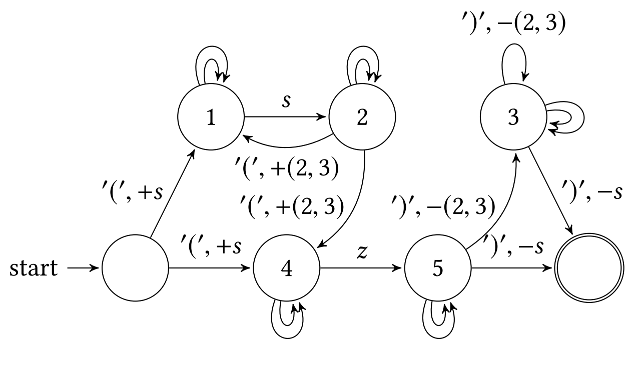


Finally, we add a new initial state and a new final state, and connect them with the initial and final states belonging to the starting non-terminal:



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 (call it $A_{()}$) that recognizes the top word for each tree in $L(T)$. Next we need to be able to add arbitrary parentheses, to produce a word with a rangeset superset. 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 $s \xrightarrow{')',-p} s$. To avoid cluttering the graph too much, these transitions are shown unlabeled below:



We will call this automaton $A'_{()}$. 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.

Two distinct successful runs that accept the same word thus represent two trees where one permits a superset range-set of the other. To find such runs we construct a product automaton and trim it. We can construct a product automaton since both $A_{()}$ and $A'_{()}$ are visibly pushdown, with the same partition of the input alphabet (push on open parenthesis, pop on close parenthesis, do nothing otherwise). We can trim the product since it retains the same partitioning and thus is also visibly pushdown.

In this product automaton, if any transition pushes a stack symbol (a, b) where $a \neq b$, or transitions to a state (p, q) where $p \neq q$, then there is a successful run that corresponds to two distinct runs through $A_{()}$ and $A'_{()}$ (since the automaton is trim).

4.2 Forbidden Children

NOTE: I had some text here about version 2 of the problem, but I haven't had time to rewrite it to align properly with the lattice formulation, so I've removed it. I'll talk about it during the meeting. Also, I probably need a better title here, this is probably a smidge too dramatic.

5 Parsetime Ambiguity Reporting

References

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

[2] 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

[3] Mathieu Caralp, Pierre-Alain Reynier, and Jean-Marc Talbot. [n. d.]. Trimming Visibly Pushdown Automata. In *Implementation and Application of Automata* (2013) *(Lecture Notes in Computer Science)*, Stavros Konstantinidis (Ed.). Springer Berlin Heidelberg, 84–96.

[4] 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.