Restricting Grammars with Tree Automata

MICHAEL D. ADAMS and MATTHEW MIGHT, University of Utah, USA

Precedence and associativity declarations in systems like `yacc` resolve ambiguities in context-free grammars (CFGs) by specifying restrictions on allowed parses. However, they are special purpose and do not handle the grammatical restrictions that language designers need in order to resolve ambiguities like dangling `else`, the interactions between binary operators and functional `if` expressions in ML, and the interactions between object allocation and function calls in JavaScript. Often, language designers resort to restructuring their grammars in order to encode these restrictions, but this obfuscates the designer's intent and can make grammars more difficult to read, write, and maintain.

In this paper, we show how tree automata can modularly and concisely encode such restrictions. We do this by reinterpreting CFGs as tree automata and then intersecting them with tree automata encoding the desired restrictions. The results are then reinterpreted back into CFGs that encode the specified restrictions. This process can be used as a preprocessing step before other CFG manipulations and is well behaved. It performs well in practice and never introduces ambiguities or LR(k) conflicts.

CCS Concepts: • **Theory of computation → Grammars and context-free languages**; **Tree languages**; • **Software and its engineering → Syntax**; **Translator writing systems and compiler generators**; **Parsers**;

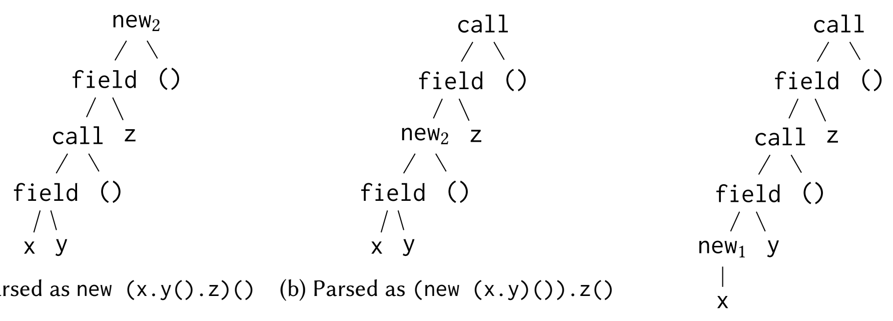
Additional Key Words and Phrases: Tree automata; Ambiguous grammars

ACM Reference Format:
Michael D. Adams and Matthew Might. 2017. Restricting Grammars with Tree Automata. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 82 (October 2017), 25 pages.
https://doi.org/10.1145/3133906

1 INTRODUCTION

Designing context-free grammars (CFGs) that have no ambiguities or unintended parses can be challenging. Aside from the special cases of precedence and associativity, many tools provide little or no assistance in this, and designers often have no choice but to refactor their grammar to eliminate unintended parses. These refactorings obscure the designer's intent and complicate maintenance.

In this paper, we show how tree automata can encode such restrictions without needing to restructure the grammar. The key is that, while intersection and difference are undecidable on CFGs, they are computable when we reinterpret grammars as tree automata. This is because tree automata are essentially CFGs viewed in terms of the set of parse *trees* they produce instead of the set of *strings* they parse. Thus the intersection of CFGs is defined in terms of sets of parsable strings while the intersection of tree automata is defined in terms of sets of parse trees. For example, the intersection of `S → S a | ` ϵ and `S → a S | ` ϵ in terms of strings is any CFG that accepts strings of the form $\mathtt{a}^n$ for all $n \in \mathbb{N}$. However, the parse trees of these two grammars differ. One grows down to the left. The other grows down to the right. The only tree shared is ϵ. Thus, their intersection as tree automata is any tree automaton that accepts only the tree ϵ.



(a) Parsed as `new (x.y().z)()` (b) Parsed as `(new (x.y)()).z()` (c) Parsed as `(new x).y().z()`

Fig. 1. Possible parse trees for `new x.y().z()`

Because this tree perspective distinguishes between multiple parse trees for the same string, it is a natural fit for resolving ambiguities, which produce multiple parse trees for the same string. This is in contrast to context-free languages, which are sets of strings and thus cannot distinguish parse trees.

As a practical example, Figure 1 shows possible parses of the string `new x.y().z()` with a naive JavaScript grammar. Figure 1a parses the string as if it were `new (x.y().z)()`, where the prototype of `new` is `x.y().z` and the final `()` is the arguments to the constructor. Figure 1b parses the string as if it were `(new (x.y)()).z()`, where the prototype of `new` is `x.y`, the first `()` is the arguments to the constructor, and `.z()` is a method call on the resulting object. Finally, Figure 1c parses the string as if it were `(new x).y().z()`, where the prototype of `new` is `x`, `new` is in its no-argument form (hence, in the parse tree it is new$_1$ instead of new$_2$), and `.y().z()` are method calls on the resulting object.

The JavaScript specification resolves this in favor of Figure 1b by using a somewhat unintuitive refactoring of the grammar. With tree automata, however, the grammar can be left alone. Instead, we restrict the grammar with the following two tree automata.

$$\neg\,(\texttt{any}^* \;.\; \texttt{new}_2(\square,\ \_) \;.\; (\texttt{field}(\square,\ \_) \mid \texttt{array}(\square,\ \_))^* \;.\; \texttt{call}(\_,\ \_))$$
$$\neg\,(\texttt{any}^* \;.\; \texttt{call}(\square,\ \_) \;.\; (\texttt{field}(\square,\ \_) \mid \texttt{array}(\square,\ \_))^* \;.\; \texttt{new}_1(\_))$$

The first automaton disallows `call` in the prototype of new$_2$ and eliminates Figure 1a. The second automaton disallows new$_1$ in the function position of a `call` and eliminates Figure 1c. The `field` and `array` in these automata are because JavaScript allows an arbitrary number of field accesses or array indexings between `call` and new$_1$ or new$_2$.

These expressions can be read like regular expressions over paths from the root of the parse tree. The sequencing operator (`.`) concatenates paths that end at a □ in its left argument with the start of paths in its right argument. The Kleene star (∗) iterates this concatenation zero or more times. The wild card (_) allows any sub-tree, and the `any` operator traverses to any child. Finally, negation (¬) rejects any tree matching its argument. This example is explained in more detail in Section 4.2.3.

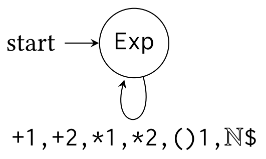
The advantage of this technique is that the original grammar can be written in a simple way that reflects the designer's intent. There is no need to complicate it with a structure dictated by the encoding of grammatical restrictions. Instead, each restriction can be encoded in its own separate tree automaton. This makes grammatical specifications modular and composable. Once these tree automata are intersected with the CFG, we get a new CFG encoding the constraints from the tree

```
Exp ⟶ nat
Exp ⟶ Exp '+' Exp
Exp ⟶ Exp '*' Exp
Exp ⟶ '(' Exp ')'
```

(a) Context-free grammar

$$\begin{aligned}
\texttt{Exp} &\longrightarrow_{\mathbb{N}} \epsilon\\
\texttt{Exp} &\longrightarrow_{+} \texttt{Exp Exp}\\
\texttt{Exp} &\longrightarrow_{\star} \texttt{Exp Exp}\\
\texttt{Exp} &\longrightarrow_{()} \texttt{Exp}
\end{aligned}$$

(b) Tree automaton as tree rewrites (Section 2.2)



(c) Tree automaton as a string automaton over paths (Section 2.3)

$$\texttt{(+(}\square\texttt{, }\square\texttt{) | }\star\texttt{(}\square\texttt{, }\square\texttt{) | ()(}\square\texttt{))}^{*}\texttt{ . }\mathbb{N}\texttt{()}$$

(d) Tree automaton as a regular expression over paths (Section 2.4)

Fig. 2. A simple expression language

automaton. This new CFG can be passed to any CFG parsing tool, so this process can be used as a preprocessor in front of other parsing tools.

The contributions of this paper are:

+ a review of the basics of tree automata and different ways of representing them (Section 2);
+ demonstrations of how tree automata can resolve the classic ambiguity problems of precedence, associativity, dangling `else`, and ML's functional `if` (Section 3);
+ case studies of our technique on two real-world languages: C and JavaScript (Section 4);
+ benchmarks of the resulting grammars and the process of creating them (Section 5); and
+ proofs of several properties of tree automata that ensure they are well behaved when interacting with either unambiguous or LR(k) parts of a grammar (Section 6).

We review related work in Section 7 and conclude in Section 8.

2 BASICS OF TREE AUTOMATA

Since tree automata are not as well known as string automata, this section reviews the basics of tree automata. A more rigorous and in-depth discussion can be found in Comon et al. [2007].

In this paper, our notations are basic with minimal syntactic sugar. Practical tools may want to incorporate a more sophisticated language designs or higher-level notations for common patterns such as precedence and associativity.

2.1 Informal Definition

Intuitively speaking, tree automata are like CFGs with two key differences. First, productions are labeled. This label does not have to be unique and is used as a constructor in the tree to be matched. Second, unlike CFGs, which take a *string* and either accept or reject it, tree automata take a *tree* and either accept or reject it.

As an example of this, consider the CFG in Figure 2a for a simple expression language. We map this grammar to a tree automaton by assigning the labels $\mathbb{N}$, +, $\star$, and `()` to the productions. We thus get the tree automaton in Figure 2b. For the sake of presentation, we elide terminals from these productions. We thus use ϵ on the right-hand side of the $\mathbb{N}$ production to indicate that it has no children. Terminals can be added back when converting the tree automaton back to a CFG.

A tree automaton accepts a tree if it can be produced by starting from the initial non-terminal and successively using productions as tree rewrites. For example, the tree automaton in Figure 2b starts with the `Exp` non-terminal as shown in Figure 3a. We then rewrite `Exp` using `Exp` $\longrightarrow_{+}$ `Exp Exp` to get the tree in Figure 3b. Further rewrites of the two `Exp` in this tree produce Figure 3c

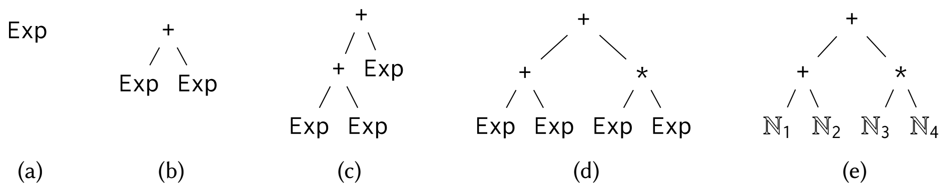


Fig. 3. Example expansion sequence for the tree automaton in Figure 2b

and then Figure 3d by using `Exp` $\to_+$ `Exp Exp` and `Exp` $\to_*$ `Exp Exp`, respectively. Finally, `Exp` $\to_{\mathbb{N}}$ ϵ rewrites all remaining `Exp` to produce Figure 3e. For the sake of clarity, we annotate terminals (e.g., $\mathbb{N}$) in this and later trees with the actual token (e.g., 1, 2, etc.) in a subscript. Note that the tree in Figure 3e is one possible parse tree that could be produced by the CFG in Figure 2a for the string `1 + 2 + 3 * 4`.

Allowing production labels to be non-unique allows different productions in a tree automaton to construct the same type of tree node. In this paper, this happens when two different parts of a tree automaton need to discuss the same CFG production, and an example of this is found in Section 3.1.3.

2.2 Tree Automata as Productions

Formally, a tree automaton is a tuple $A = (Q, F, S, \delta)$ where Q is a set of automata states, F is a set of tree-constructor labels, $S \subseteq Q$ is a set of root states, and δ is a set of productions. Each label $f \in F$ has a fixed arity $|f| \in \mathbb{N}$. Productions are of the form $q \to_f q_1 q_2 \cdots q_{|f|}$ where $q, q_1, q_2, \cdots, q_n \in Q$ and $f \in F$. This indicates that a leaf in state q can be rewritten to a tree node labeled with f that has $q_1, q_2, \cdots, q_{|f|}$ as children (i.e., $f(q_1, q_2, \cdots, q_{|f|})$). A tree automaton accepts a tree if and only if there are tree rewrites from an element of S to that tree.

We may also allow epsilon productions of the form $q \to_\epsilon p$. These indicate that a leaf in state q can be rewritten to state p without creating any child nodes. Similar to finite-state *string* automata, *tree* automata both with and without epsilon edges are equivalent in power and can be mutually derived from each other [Comon et al. 2007].

The definition we have just given is for top-down tree automata, but there is another category of tree automata, bottom-up tree automata. Instead of reading a production $q \to_f q_1 q_2 \cdots q_{|f|}$ as an expansion rule from q to $f(q_1, q_2, \cdots, q_{|f|})$, for bottom-up tree automata we read it as a collapsing rule from $f(q_1, q_2, \cdots, q_{|f|})$ to q. Where top-down automata start with a root state in S and accept only if expanding downward can construct the tree to be matched, bottom-up automata start at the leaves of a tree to be matched and accept only if collapsing upward can result in a single root node that is in S.

Tree automata are also classified based on whether they are deterministic or non-deterministic. An automaton is deterministic if during rewriting there is at most one rewrite that can be done at a particular point in the tree. Otherwise, it is non-deterministic. For deterministic top-down automata, there is thus at most one production for a given q and f. For deterministic bottom-up automata, there is thus at most one production for a given f and $q_1, q_2, \cdots, q_{|f|}$. Epsilon productions are not allowed in either.

Non-deterministic top-down, non-deterministic bottom-up, and deterministic bottom-up automata are equivalent in expressibility and can be converted into each other [Comon et al. 2007].

$e \in Exp := \emptyset \mid e \vert e \mid f(e, e, \cdots, e)$	Empty, union, and constructors
$\mid e \,._i e \mid \Box_i \mid e^{*i}$	Sequencing, sequencing hole, and Kleene star
$\mid \neg\, e \mid e \& e$	Negation and intersection
$\mid$ _ $\mid$ `left`$_i$ $\mid$ `right`$_i$ $\mid$ `any`$_i$	Wild card and child positions

Fig. 4. Regular expressions for tree automata. (Let $i \in \mathbb{N}$ or be omitted, and $f \in F$ where F is the type of labels.)

Deterministic top-down automata are strictly weaker than the others in terms of expressibility [Comon et al. 2007]. For simplicity of presentation, this paper glosses over these differences except where they are relevant.

Using algorithms similar to those for string automata, tree automata can be unioned, intersected, and complemented [Comon et al. 2007]. Likewise, there are algorithms for determinization and minimization but only for bottom-up automata [Comon et al. 2007]. Because these algorithms are standard, we do not delve into algorithmic details except for the performance considerations discussed in Section 5.

A CFG can be converted to a tree automaton by labeling its productions, and the reverse can be done by erasing the labels. The remaining differences are in the interpretation of productions. For example, CFGs match strings, while tree automata match trees. In this paper, we move freely between CFGs and tree automata.

2.3 Tree Automata as String Automata over Paths

The above definition of tree automata treats them as tree-rewrite systems. However, sometimes it is convenient to consider them in terms of string automata over paths. These automata are not for the input tokens being parsed but rather for paths between the root and leaves of the tree. For example, Figure 3e has four paths. Each starts from the + at the root, goes down through subtrees labeled with + and ⋆, and ends at one of the $\mathbb{N}$ leaves. We encode these paths as strings of constructor labels interleaved with the position of the child to follow. Leaves, which have no children, use the distinguished symbol \$ to end the path.

In this paper, we use automata for paths where all states are accepting and an automaton fails only if there is no transition matching the next step of the path. Thus, it does not matter to which state the final \$ goes.

For example, the path to $\mathbb{N}_3$ in Figure 3e is encoded as $+2{\star}1\mathbb{N}_3\$$. The + is for the + at the root of the tree. The 2 indicates that the path goes down the second child of the root. The ⋆ is for the ⋆ that labels the second child of the root, and the 1 indicates the path goes down the first child of that ⋆ sub-tree. Finally, $\mathbb{N}_3$ and \$ are for the $\mathbb{N}_3$ leaf, which has no children. The other paths that go to $\mathbb{N}_1$, $\mathbb{N}_2$, and $\mathbb{N}_4$ are encoded as $+1{+}1\mathbb{N}_1\$$, $+1{+}2\mathbb{N}_2\$$, and $+2{\star}2\mathbb{N}_4\$$, respectively.

An automaton accepts a tree if it accepts the strings for all of the paths in that tree using the same automaton states for any shared prefixes. For example, with Figure 3e, it must accept $+1{+}1\mathbb{N}_1\$$ and $+1{+}2\mathbb{N}_2\$$ using the same states for +1+ and all prefixes of +1+. Formally, an automaton accepts a tree t if $accept\,(q_0, t)$ holds, where q_0 is the start state and $accept$ is defined as follows.

$$accept\left(q, f\left(t_1, t_2, \cdots, t_{|f|}\right)\right) = \exists q'.q \xrightarrow{f} q' \wedge \begin{cases} \exists q''.q' \xrightarrow{\$} q'' & \text{if } |f| = 0 \\ \bigwedge\limits_{1 \le i \le |f|} \left(\exists q''.q' \xrightarrow{i} q'' \wedge accept\,(q'', t_i)\right) & \text{if } |f| > 0 \end{cases}$$

$$\begin{aligned}
\_ \quad &= (C_1(\square, \cdots, \square) \mid C_2(\square, \cdots, \square) \mid \cdots \mid C_n(\square, \cdots, \square))^{*}. \\
&\quad (D_1() \mid D_2() \mid \cdots \mid D_m()) \\
\\
\texttt{left}_i &= C_1(\square_i, \_, \cdots, \_) \mid C_2(\square_i, \_, \cdots, \_) \mid \cdots \mid C_n(\square_i, \_, \cdots, \_) \\
\texttt{right}_i &= C_1(\_, \cdots, \_, \square_i) \mid C_2(\_, \cdots, \_, \square_i) \mid \cdots \mid C_n(\_, \cdots, \_, \square_i) \\
\texttt{any}_i &= C_1(\square_i, \_, \cdots, \_) \mid C_1(\_, \square_i, \_, \cdots, \_) \mid \cdots \mid C_1(\_, \cdots, \_, \square_i) \mid \\
&\quad C_2(\square_i, \_, \cdots, \_) \mid C_2(\_, \square_i, \_, \cdots, \_) \mid \cdots \mid C_2(\_, \cdots, \_, \square_i) \mid \\
&\quad \vdots \\
&\quad C_n(\square_i, \_, \cdots, \_) \mid C_n(\_, \square_i, \_, \cdots, \_) \mid \cdots \mid C_n(\_, \cdots, \_, \square_i)
\end{aligned}$$

Fig. 5. Desugarings of wild card (_), left$_i$, right$_i$, and any$_i$

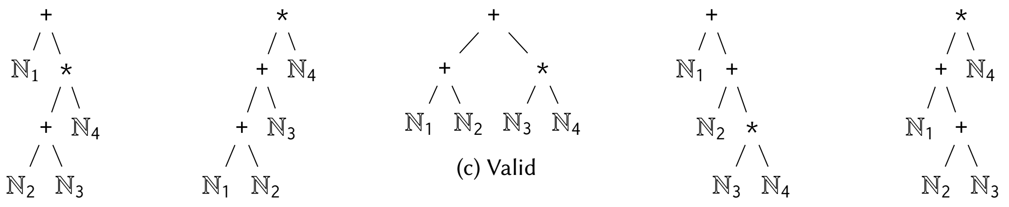
To convert a tree automaton into a string automaton for paths, we start with an automaton that has a start state s and an epsilon edge $s \xrightarrow{\epsilon} s'$ for every root state $s' \in S$ in the tree automaton. We also add a state, p, to be used for leaf nodes and a state q for every state in the Q of the tree automaton. Then, for every production of the form $q \rightarrow_f q_1 q_2 \cdots q_{|f|}$, if $|f| = 0$, then we add an edge $q \xrightarrow{f} q'$ and $q' \xrightarrow{\$} p$ where q' is fresh. If $|f| \neq 0$, then we add the edges $q \xrightarrow{f} q', q' \xrightarrow{1} q_1, q' \xrightarrow{2} q_2, \cdots, q' \xrightarrow{|f|} q_{|f|}$ where q' is fresh. Finally, we add an epsilon edge, $q \xrightarrow{\epsilon} q'$, for each epsilon production, $q \rightarrow_\epsilon q'$, in the tree automaton.

Applying this process to the tree automaton in Figure 2b and simplifying the result produces the string automaton in Figure 2c. For the sake of brevity, we let $q_1 \xrightarrow{fn} q_2$ stand for edges $q_1 \xrightarrow{f} q'$ and $q' \xrightarrow{n} q_2$ with a fresh q' and say the automaton processes fn when it processes f followed by n. Because this grammar has only one non-terminal, this automaton is almost trivial, but later automata in this paper have more interesting structures.

2.4 Tree Automata as Regular Expressions over Paths

Tree automata can also be expressed in the style of regular expressions over paths using the forms in Figure 4. The minimal complete set of operators for tree regular expressions is empty ($\emptyset$), union ($e \mid e$), constructors ($f(e, e, \cdots, e)$), sequencing ($e \,._i\, e$), sequencing hole ($\square_i$), and Kleene star (e^{*_i}). Empty ($\emptyset$) rejects all trees, and union ($e_1 \mid e_2$) unions the sets of tree accepted by e_1 and e_2. Constructors ($f(e_1, e_2, \cdots, e_n)$) match a tree if and only if the root has the constructor label f and has children that match the component sub-expressions $e_1, e_2, \cdots, e_n$.

The sequencing ($._i$) and sequencing hole ($\square_i$) forms are more subtle. As might be expected, $e_1 \,._i\, e_2$ requires sequentially matching the expressions e_1 and e_2 and corresponds to sequencing the string automata in Section 2.3. However, because parse trees can have multiple branches, sequencing may need to differentiate between branches. For example, the root of the tree in Figure 3e has two children that a sequencing operator could proceed with. Thus, we use a hole ($\square_i$) to explicitly label where an expression finishes matching and annotate both the hole ($\square_i$) and the sequencing operator ($._i$) with the index i to specify which $\square_i$ and $._i$ correspond to each other. For example, we may have `+(`$\square_1$`,` $\square_2$`)` $._1$ `+(_, _)` $._2$ `*(_, _)`. Due to the annotated subscripts, this matches the left child of a `+` node with `+(_, _)` and the right child with `*(_, _)` and is equivalent to `+(+(_, _), *(_, _))`. From this sequencing operator we can then define the Kleene star, e^{*_i}, as zero or more e sequenced with $._i$. We omit the subscripts when these forms do not need to be differentiated. For example,



(a) Precedence violation (b) Precedence violation (c) Valid (d) Associativity violation (e) Associativity and precedence violation

Fig. 6. Possible parse trees for `1 + 2 + 3 * 4`

the expressions `(+(`$\square_1$`, `$\square_1$`)` $._1$ `*(_, _))` and `(+(`$\square$`, `$\square$`) . *(_, _))` are equivalent to each other and match `*(_, _)` against both the left and right child of a + node.

With the exception of sequencing hole ($\square_i$) these forms mirror the minimal complete set of operators for string regular expressions. Also, as with string regular expressions, Kleene star (e^{*i}) is what makes it possible to express non-trivial automata that contain more than a fixed, finite set of accepted values.

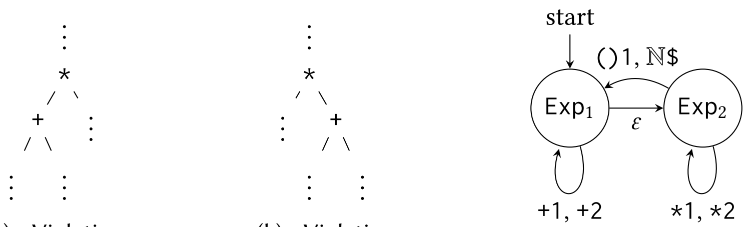
Writing tree automata purely in terms of these minimal forms can be impractical in some cases, similar to how it can be impractical to write some string regular expressions without a wild card. So, in this paper we also include negation ($\neg e$), intersection ($e \& e$), wild card (_), and child positions ($\texttt{left}_i$, $\texttt{right}_i$, and $\texttt{any}_i$).

Negation ($\neg e_1$) and intersection ($e_1 \& e_2$) respectively complement and intersect the sets of trees accepted by e_1 and e_2. They are not in the minimal complete set of operators, but we treat them as primitive as encoding them in the minimal forms can be complicated [Comon et al. 2007].

Wild card (_) accepts all trees, while $\texttt{left}_i$, $\texttt{right}_i$, and $\texttt{any}_i$ traverse to the leftmost child, rightmost child, or any child, respectively. Note that wild card (_) differs from $\texttt{left}_i$, $\texttt{right}_i$, and $\texttt{any}_i$ in that it consumes an entire subtree, while $\texttt{left}_i$, $\texttt{right}_i$, and $\texttt{any}_i$ descend only a single step. Thus, wild card cannot be sequenced with further regular expressions, but $\texttt{left}_i$, $\texttt{right}_i$, and $\texttt{any}_i$ can. As with sequencing hole ($\square_i$), $\texttt{left}_i$, $\texttt{right}_i$, and $\texttt{any}_i$ use the index i to specify the sequencing ($._i$) with which to proceed after traversing to the appropriate child. For example, we could check if the right child of the current tree node is a + with $\texttt{right}_1$ $._1$ `+(_,_)`. Applying Kleene star to $\texttt{left}_i$, $\texttt{right}_i$, or $\texttt{any}_i$ allows us to traverse arbitrarily far into the tree and is a common pattern in this paper. So, we could check if a tree contains a + node anywhere or on the rightmost descending path with the expressions ($\texttt{any}^*$ `. +(_,_))` or ($\texttt{right}^*$ `. +(_,_))`, respectively.

Note that if you encode strings as a tree where each node has only a single child and letters in the string correspond to tree-constructor labels, then $\texttt{left}_i$, $\texttt{right}_i$, and $\texttt{any}_i$ are all equivalent to each other and correspond to the wild card in string regular expressions.

The desugarings for these forms are in Figure 5, where $C_1, C_2, \cdots, C_n$ and $D_1, D_2, \cdots, D_m$ are the labels in F for which $\forall i.\ |C_i| > 0$ and $\forall i.\ |D_i| = 0$. These desugarings depend on F, the tree-constructor labels over which a tree automaton operates. Thus, their desugaring is different for different base grammars. This is similar to the alphabet (i.e., Σ in most presentations) in string regular expressions and how a wild card for string regular expressions is encoded relative to that alphabet. In this paper, F is specified by production labels in the CFG (though particular automata



(a) Violating pattern (b) Violating pattern (c) Automaton

$$\begin{array}{l} ((+(\square,\ \square))^* \ . \\ \ (*(\square,\ \square))^* \ . \\ \ (\mathbb{N}() \mid ()(\square)))^* \end{array}$$

(d) Regular expression

$$\begin{array}{lll} \mathtt{Exp}_1 & \rightarrow_{+} & \mathtt{Exp}_1\ \mathtt{Exp}_1 \\ \mathtt{Exp}_1 & \rightarrow_{\epsilon} & \mathtt{Exp}_2 \\ \\ \mathtt{Exp}_2 & \rightarrow_{\mathbb{N}} & \epsilon \\ \mathtt{Exp}_2 & \rightarrow_{*} & \mathtt{Exp}_2\ \mathtt{Exp}_2 \\ \mathtt{Exp}_2 & \rightarrow_{()} & \mathtt{Exp}_1 \end{array}$$

(e) Tree rewrites

Fig. 7. Precedence for the CFG in Figure 2a

might use only a subset). Thus, $\mathtt{left}_i$, $\mathtt{right}_i$, and any_i compose when they are for the same grammar but do not compose when they are for different grammars (e.g., C versus JavaScript).

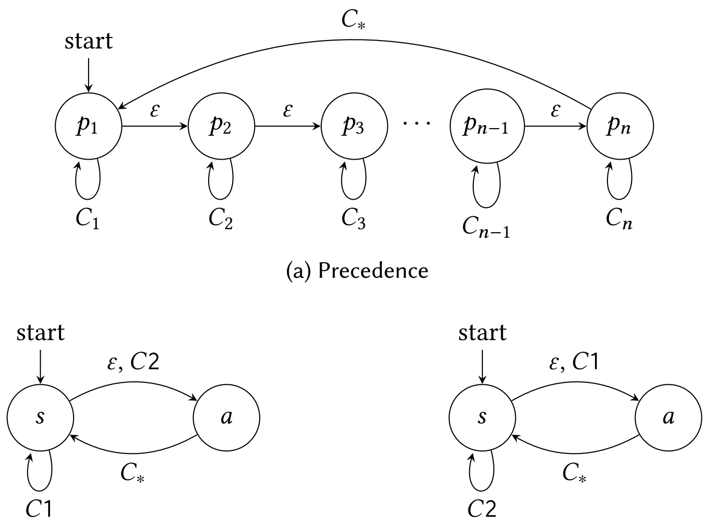
3 APPLYING TREE AUTOMATA TO CONTEXT-FREE GRAMMARS

We argue that tree automata are an elegant method of specifying constraints on CFGs. They can specify associativity and precedence rules and can resolve ambiguities like the dangling `else` problem. In this section, we give examples of this and show how this unified approach resolves several classic ambiguity problems. To do this, we first label each production in the grammar with a constructor name and map the CFG to a tree automaton. Then, we intersect the resulting tree automaton with a tree automaton specifying the desired restrictions. Finally, we map the tree automaton resulting from that intersection back to a CFG. Any semantic actions associated with the original productions can then be assigned to the productions in the resulting CFG based on the constructors labeling the productions of the tree automata.

3.1 Precedence and Associativity

As an example of precedence and associativity, consider the grammar in Figure 2a when parsing the string `1 + 2 + 3 * 4`. This grammar is ambiguous, and thus all five parse trees in Figure 6 are possible. However, making multiplication have tighter precedence than addition and requiring that both operators be left associative excludes all parse trees except Figure 6c.

3.1.1 Precedence. Enforcing precedence for the grammar in Figure 2a involves enforcing restrictions between the + and * constructors. When inside *, we must not use + until we use some other constructor such as `()`. Any tree with a structure like in Figure 7a or Figure 7b should be rejected. For example, Figure 6a, Figure 6b, and Figure 6e all match Figure 7a. This leaves only Figure 6c and Figure 6d, which we resolve with associativity in Section 3.1.2.



(a) Precedence

(b) Left associativity

(c) Right associativity

Fig. 8. General tree automata for precedence and associativity

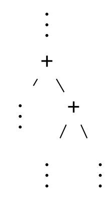
We can enforce this with the tree automaton in Figure 7c. Note that the ℕ$ transition terminates a path, so it does not matter to which state it goes.

In the general case for precedence, once we use a constructor from the precedence hierarchy, we may use only those constructors further along the hierarchy until we use some constructor not in the hierarchy. For example, once using `*`, we may not use `+` until we use `()`. Let C_i be the constructors at level i of the hierarchy and C_* be the constructors not in the hierarchy. As we go down a parse tree, we forbid constructors in C_i as children of constructors in C_j where $i < j$.

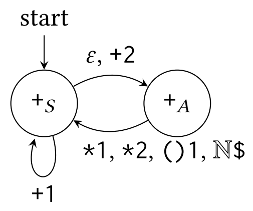
Figure 8a shows the general structure of a tree automaton enforcing this. Each state p_i is occupied when only operators of precedence i or higher are allowed, and C_i contains fn for each $f \in C_i$ and $1 \le n \le |f|$. This graph meshes nicely with our intuitive notion of ascending precedence. It starts in p_1, but once an element of, say, C_3 is used, it follows the epsilon transitions to p_3. Once there, the constructors in C_1 and C_2 cannot be used unless a constructor from C_* is used first. Using this pattern for `+` and `*` results in the automaton in Figure 7c.

We can also express this tree automaton as the regular expression in Figure 7d, which is equivalent to the automaton in Figure 7c. The intuition here is that, when descending a tree, we are allowed to traverse zero or more `+` constructors followed by zero or more `*` constructors, which may then be followed by a single `()` or ℕ constructor. ℕ has no children, but `()` does, so once a `()` is traversed, we may repeat this entire pattern.

Finally, expressing this tree automaton in terms of productions results in Figure 7e. This closely parallels how one would encode precedence directly into a CFG. An important difference, though, is that we can specify a tree automaton for precedence separately from the tree automata for associativity and combine them later. (In Section 3.1.3, we show how to do this combination.) On the other hand, encoding these rules into a CFG requires that they be encoded simultaneously.



(a) Violating pattern



(b) Automaton

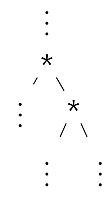
```
¬(any* .
  +(_,+(_,_)))
```

(c) Regular expression

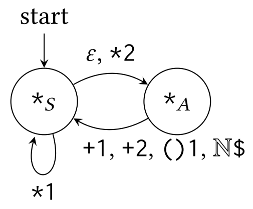
$$\begin{aligned} \texttt{Exp} &\rightarrow_{+} \texttt{Exp}\ \texttt{Exp+} \\ \texttt{Exp} &\rightarrow_{\epsilon} \texttt{Exp+} \\[1ex] \texttt{Exp+} &\rightarrow_{\mathbb{N}} \epsilon \\ \texttt{Exp+} &\rightarrow_{\star} \texttt{Exp}\ \texttt{Exp} \\ \texttt{Exp+} &\rightarrow_{()} \texttt{Exp} \end{aligned}$$

(d) Tree rewrites

Fig. 9. Associativity of + for the CFG in Figure 2a



(a) Violating pattern



(b) Automaton

```
¬(any* .
  ⋆(_,⋆(_,_)))
```

(c) Regular expression

$$\begin{aligned} \texttt{Exp} &\rightarrow_{\star} \texttt{Exp}\ \texttt{Exp}\star \\ \texttt{Exp} &\rightarrow_{\epsilon} \texttt{Exp}\star \\[1ex] \texttt{Exp}\star &\rightarrow_{\mathbb{N}} \epsilon \\ \texttt{Exp}\star &\rightarrow_{+} \texttt{Exp}\ \texttt{Exp} \\ \texttt{Exp}\star &\rightarrow_{()} \texttt{Exp} \end{aligned}$$

(d) Tree rewrites

Fig. 10. Associativity of ⋆ for the CFG in Figure 2a

3.1.2 Associativity. Associativity also imposes restrictions, but now the restriction is on sub-trees having children with the same constructor as themselves. For example, to impose left associativity of + and ⋆, we reject anything matching Figure 9a or Figure 10a. This rejects Figure 6d and Figure 6e.

In the general case, a constructor for a left (right) associative operator may not have a right (left) child with the same constructor. For example, for a left associative +, + may not have + as its right child. Left (right) associativity is thus encoded with a tree automata structured like in Figure 8b (Figure 8c). In these automata, C is the constructor label of the associative operator and C_* contains fn for all other constructors f and $1 \le n \le |f|$. From s, every constructor is allowed either directly or by first following the epsilon transition to a. From a, only constructors other than C are allowed. When this automaton is in state s, descending into the right (left) child of C puts it in state a, at which point we must traverse C_* before using C again. Descending the left (right) child of C or any child of any other constructor puts it in state s. Using this pattern to enforce left associativity of + and ⋆ results in the automata in Figure 9b and Figure 10b, respectively.

We can also express these tree automata as the regular expressions in Figure 9c and Figure 10c. The innermost patterns, `+(_, +(_, _))` and `⋆(_, ⋆(_, _))`, match the cases when associativity is violated. The any* expression looks for such violations anywhere in the tree, and ¬ rejects any tree in which such violations are found.

Finally, expressing these tree automata in terms of productions results in Figure 9d and Figure 10d. As with precedence, these closely parallel how one would encode associativity into a CFG.

3.1.3 Combining Precedence and Associativity. Now that we have a tree automaton for precedence and tree automata for associativity of + and ⋆, we can combine these with the tree automaton for the original grammar by using the tree-automata intersection algorithm [Comon et al. 2007].

$$\begin{array}{lll}
\texttt{Exp}_{1\_\_} & \longrightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp}_{1\_\_} & \longrightarrow_{+} & \texttt{Exp}_{1\_\_}\ \texttt{Exp}_{1+\_} \\
\texttt{Exp}_{1\_\_} & \longrightarrow_{\star} & \texttt{Exp}_{2\_\_}\ \texttt{Exp}_{2\_*} \\
\texttt{Exp}_{1\_\_} & \longrightarrow_{()} & \texttt{Exp}_{1\_\_} \\
\\
\texttt{Exp}_{1+\_} & \longrightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp}_{1+\_} & \longrightarrow_{\star} & \texttt{Exp}_{2\_\_}\ \texttt{Exp}_{2\_*} \\
\texttt{Exp}_{1+\_} & \longrightarrow_{()} & \texttt{Exp}_{1\_\_} \\
\\
\texttt{Exp}_{2\_\_} & \longrightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp}_{2\_\_} & \longrightarrow_{\star} & \texttt{Exp}_{2\_\_}\ \texttt{Exp}_{2\_*} \\
\texttt{Exp}_{2\_\_} & \longrightarrow_{()} & \texttt{Exp}_{1\_\_} \\
\\
\texttt{Exp}_{2\_*} & \longrightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp}_{2\_*} & \longrightarrow_{()} & \texttt{Exp}_{1\_\_}
\end{array}$$

(a) Grammar output from intersection

$$\begin{array}{lll}
\texttt{Exp} & \longrightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp} & \longrightarrow_{+} & \texttt{Exp Term} \\
\texttt{Exp} & \longrightarrow_{\star} & \texttt{Term Factor} \\
\texttt{Exp} & \longrightarrow_{()} & \texttt{Exp} \\
\\
\texttt{Term} & \longrightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Term} & \longrightarrow_{\star} & \texttt{Term Factor} \\
\texttt{Term} & \longrightarrow_{()} & \texttt{Exp} \\
\\
\texttt{Factor} & \longrightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Factor} & \longrightarrow_{()} & \texttt{Exp}
\end{array}$$

(b) Grammar after duplicate removal and renaming

$$\begin{array}{lll}
\texttt{Exp} & \longrightarrow_{+} & \texttt{Exp Term} \\
\texttt{Exp} & \longrightarrow_{\epsilon} & \texttt{Term} \\
\\
\texttt{Term} & \longrightarrow_{\star} & \texttt{Term Factor} \\
\texttt{Term} & \longrightarrow_{\epsilon} & \texttt{Factor} \\
\\
\texttt{Factor} & \longrightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Factor} & \longrightarrow_{()} & \texttt{Exp}
\end{array}$$

(c) Grammar after epsilon introduction

Fig. 11. Intersection of the tree automata for precedence and associativity

This creates a tree automaton that accepts a tree only if all the intersected tree automata accept it. For these tree automata, the intersection algorithm produces the tree automaton in Figure 11a.

The intersection algorithm effectively builds a Cartesian product similar to that for string automata intersection. $\texttt{Exp}_{1\_\_}$ corresponds to the Cartesian product of $\texttt{Exp}_1$, `Exp`, and `Exp` in Figure 7e, Figure 9d, and Figure 10d, respectively. $\texttt{Exp}_{1+\_}$ corresponds to $\texttt{Exp}_1$, $\texttt{Exp}_+$, and `Exp` in Figure 7e, Figure 9d, and Figure 10d, respectively. $\texttt{Exp}_{2\_\_}$ corresponds to $\texttt{Exp}_2$, `Exp`, and `Exp` in Figure 7e, Figure 9d, and Figure 10d, respectively. Finally, $\texttt{Exp}_{2\_*}$ corresponds to $\texttt{Exp}_2$, `Exp`, and $\texttt{Exp}_\star$ in Figure 7e, Figure 9d, and Figure 10d, respectively.

As mentioned in Section 2.1, this is an example of why constructor labels are non-unique as $\texttt{Exp}_{1\_\_}$, $\texttt{Exp}_{1+\_}$, and $\texttt{Exp}_{2\_\_}$ all use the $\star$ constructor.

This grammar can be simplified by first noting that $\texttt{Exp}_{1+\_}$ and $\texttt{Exp}_{2\_\_}$ are duplicates. In the general case, such duplication can be eliminated using the the tree-automata minimization algorithm [Comon et al. 2007]. This results in Figure 11b where we have also renamed the non-terminals to more conventional names. `Exp` corresponds to $\texttt{Exp}_{1\_\_}$. `Term` corresponds to $\texttt{Exp}_{1+\_}$ and $\texttt{Exp}_{2\_\_}$. Finally, `Factor` corresponds to $\texttt{Exp}_{2\_*}$.

We can further simplify this grammar by noting that the productions in `Term` and `Factor` are subsets of those in `Exp` and `Term`, respectively. Thus, we can replace the overlapping productions in `Exp` and `Term` with epsilon productions to `Term` and `Factor`, respectively. This results in Figure 11c. This grammar is exactly how one would normally encode precedence and associativity into a CFG if doing so by hand. The difference is that we created this *compositionally* by specifying the precedence and associativity rules separately and combining them later.

In the general case, we take the productions in each non-terminal, find the non-terminals that disjointly contain the largest subset of that set of productions, and replace those with epsilon productions to those non-terminals. In Section 5.1.1, we discuss the effectiveness of this heuristic.

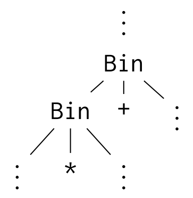
3.1.4 Generic Operators. In addition to the ways of specifying precedence and associativity in Section 3.1.1 and Section 3.1.2, tree automata allow precedence rules to be specified for generic

```
Exp   → Exp BinOp Exp
Exp   → ‘(’ Exp ‘)’
Exp   → nat
BinOp → ‘+’
BinOp → ‘*’
```

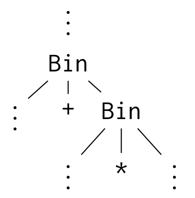
(a) CFG with generic operators

$$\begin{array}{lll} \mathtt{Exp} & \rightarrow_{\mathtt{Bin}} & \mathtt{Exp\ BinOp\ Exp} \\ \mathtt{Exp} & \rightarrow_{\mathtt{()}} & \mathtt{Exp} \\ \mathtt{Exp} & \rightarrow_{\mathbb{N}} & \epsilon \\ \mathtt{BinOp} & \rightarrow_{+} & \epsilon \\ \mathtt{BinOp} & \rightarrow_{\star} & \epsilon \end{array}$$

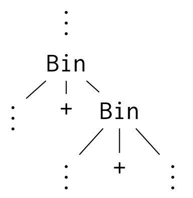
(b) Tree automaton for generic operators



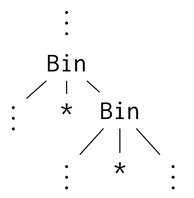
(c) Violating pattern for precedence



(d) Violating pattern for precedence



(e) Violating pattern for associativity of +



(f) Violating pattern for associativity of ⋆

$$\mathtt{((Bin(\square,\ +(),\ \square))^{*}\ .\ (Bin(\square,\ \star(),\ \square))^{*}\ .\ (\mathbb{N}()\ |\ ()(\square)))^{*}}$$

(g) Precedence

$$\neg\mathtt{(any^{*}\ .\ Bin(\_,\ +(),\ Bin(\_,\ +(),\ \_)))}$$

(h) Associativity for +

$$\neg\mathtt{(any^{*}\ .\ Bin(\_,\ \star(),\ Bin(\_,\ \star(),\ \_)))}$$

(i) Associativity for ⋆

Fig. 12. Precedence and associativity for generic operators

operators where one production specifies multiple operators of different precedences and associativities. For example, in Figure 12a the second production of `Exp` specifies both + and ⋆ expressions depending on which `BinOp` is used. Thus, precedence violations look like in Figure 12c and Figure 12d, and associativity violations for + and ⋆ look like in Figure 12e and Figure 12f, respectively. Unlike in Figure 7, Figure 9, and Figure 10, these patterns are not based on whether + or ⋆ is a child of another + or ⋆. In fact, + and ⋆ never have children. Instead, whether one `Bin` is allowed to be the child of another `Bin` depends on whether their `BinOp` children are + or ⋆.

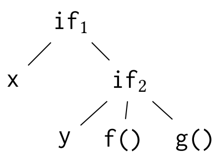
Traditional precedence and associativity cannot examine these `BinOp` when deciding whether to allow one `Bin` inside another, but tree automata can. For example, for the tree automaton in Figure 12b, we can specify precedence using the tree automaton in Figure 12g and left associativity for + and ⋆ using the tree automata in Figure 12h and Figure 12i, respectively.

3.2 Dangling `else`

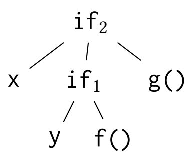
A common and well-known example where CFGs need ambiguity resolution is the dangling `else` problem. For example, in the C programming language, consider the following input.

```
if (x) if (y) f(); else g();
```

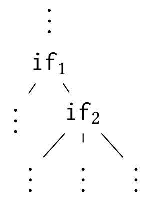
Since C has both two-armed `if` forms that have an `else` clause and one-armed `if` forms that do not, this can be parsed in two different ways. The first is to treat the first `if` as one-armed and the



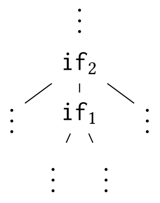
(a) Parse for `if (x) if (y) f(); else g()`



(b) Parse for `if (x) if (y) f(); else g()`



(c) Pattern for Figure 13a



(d) Pattern for Figure 13b

$\neg$(any* . if$_2$(_, if$_1$(_, _), _))

(e) Regular expression excluding Figure 13c

$\neg$(any* . if$_1$(_, if$_2$(_, _, _)))

(f) Regular expression excluding Figure 13d

Fig. 13. Tree automaton for dangling `else`

`else` as belonging with the second `if`. This results in the parse tree in Figure 13a. The second way is to treat the second `if` as one-armed and the `else` as belonging with the first `if`. This results in the parse tree in Figure 13b.

Resolving this ambiguity with a tree automaton is similar to how we handled associativity. In the general case, the ambiguity is between parse trees structured like in Figure 13c and Figure 13d. To resolve this ambiguity, we exclude one of them. Suppose we want to exclude Figure 13d, as is the case in C and many other languages. This is accomplished with the regular expression in Figure 13e.

Focusing on Figure 13e and working from the inside-out, this tree automaton looks for a one-armed `if` with if$_1$(_, _) that is in the `then` branch of a two-armed `if`. The any* expression looks for this pattern anywhere in the parse tree, and the $\neg$ operator rejects the matching trees.

Alternatively, if we want to exclude Figure 13c, we can use the regular expression in Figure 13f. This has the same structure as the previous tree automaton except that it looks for a two-armed `if` in the `then` branch of a one-armed `if`.

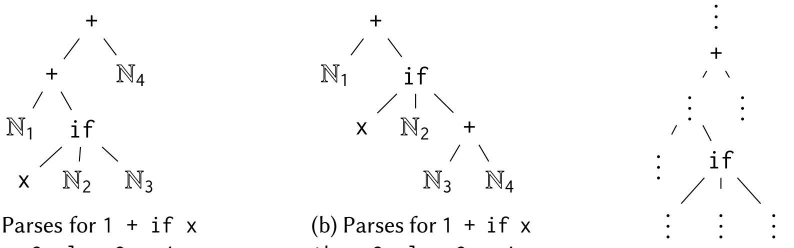
In LR(k)-based parsers, the dangling `else` problem shows up as a shift/reduce conflict and is traditionally resolved by always favoring a shift over a reduce. Effectively this chooses Figure 13a over Figure 13b. However, since there are two parse trees for the same string, the ambiguity is in the CFG itself and is not due to using an LR(k) parser. Thus, favoring shift over reduce uses low-level details of LR(k) parsing to drive language design. With tree automata, the language designer is free to chose either Figure 13a or Figure 13b and need not assume that the parser uses LR(k). (Though it would be interesting future work to explore if tree automata can be used to resolve shift/reduce conflicts that are due to LR(k) and not due to an ambiguity in the CFG itself.)

3.3 Functional `if` Expressions

A variation of dangling `else` is seen in ML and other languages where `if` forms are expressions that can nest in other expressions. For example, we could have the following input where + is a left-associative operator.

```
1 + if x then 2 else 3 + 4
```

This input contains an ambiguity. The `else` branch might contain just the `3` with the `+ 4` being outside the `if`, or it might include all of `3 + 4`. Parse trees for these two cases are in Figure 14a and



(a) Parses for `1 + if x then 2 else 3 + 4`

(b) Parses for `1 + if x then 2 else 3 + 4`

(c) Violating pattern

$$\neg\,(\texttt{any}^* \;.\; \texttt{+}(\square,\ \_) \;.\; \texttt{right}^* \;.\; \texttt{if}(\_,\ \_,\ \_))$$

(d) Regular expression

Fig. 14. Tree automaton for functional `if`

Figure 14b, respectively. The rule in ML is to favor the parse with the largest `else` clause, which chooses Figure 14b as the desired parse.

This problem cannot be solved using traditional precedence rules, because, as seen in Figure 14b, + is allowed both inside and outside an `if`, and traditional precedence rules would necessarily exclude one of these. Using a tree automaton, though, we can resolve this ambiguity. Observe that Figure 14a violates ML's longest-`else` rule due to the + containing the `if` that could instead go under the `if`. In general, this is the case when an `if` is the rightmost descendant of the left child of a +, as is the case in Figure 14a but not Figure 14b. We thus want to forbid the pattern in Figure 14c. This is encoded by the regular expression in Figure 14d. This uses the same structure as for associativity in Section 3.1.2, except that we now have `right`* between + and `if`. This allows an arbitrary length path that traverses into only rightmost children between + and `if`. This example shows how tree automata can express more than traditional precedence or associativity rules by allowing us to restrict descendants and not just immediate children.

4 USE IN REAL-WORLD LANGUAGES

To test using tree automata with real-world languages, we examined the grammars for C 99 [ISO 2007] and JavaScript 5.1 [ECMA 2011] and looked for instances where tree automata could simplify the grammar. These include not only precedence, associativity, and dangling `else` as already described in Section 3 but also language specific simplifications. In Section 4.1 and Section 4.2, we describe these simplifications for C and JavaScript, respectively. Later in Section 5, we report benchmarks for the resulting grammars.

4.1 C

4.1.1 Precedence and Casts. C has a twist on the usual precedence rules. Cast expressions, such as `(int*)x`, are not allowed in some unary expressions such as pre-increment. For example, `++(int*)x` is not allowed. This could normally be expressed by saying these unary expressions have a tighter precedence than cast expressions. However, cast expressions *are* allowed inside certain other unary expressions such as pointer dereferencing. For example, `*(int*)x` is allowed. Furthermore, with an intervening unary operator such as dereferencing, a cast expression is allowed in unary operators where it is not usually allowed. For example, `++*(int*)x` is allowed.

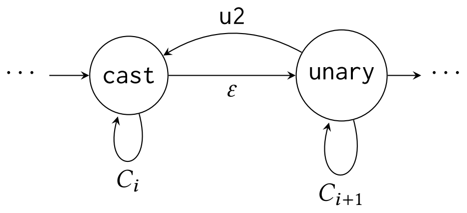


Fig. 15. Precedence of cast and unary operators in C

The rules for these forms cannot be expressed with traditional precedence declarations. They can be expressed, however, using tree automata. In Figure 15, we encode this restriction by first constructing a tree automaton for precedence in the style described in Section 3.1.1. Then, we add an edge for unary each operator, u, that goes backwards in the precedence chain from `unary` expressions to `cast` expressions.

4.1.2 Comma and Assignment Operators. Another place where tree automata could simplify the C grammar is the comma (`,`) operator. This operator evaluates the expressions to its left and right, and returns the result of the right. However, function arguments, array initializers, and `struct` initializers use the comma *token* for other purposes. Thus, the comma *operator* is not allowed in these forms. Instead of the `expression` non-terminal, these forms use the `assignment-expression` non-terminal, which is one step further along the precedence hierarchy than `expression` and does not include the comma operator.

The assignment operator (=) has similar restrictions. Some parts of the grammar permit only constants. This includes bit-width declarations, `enum` declarations, the size part of an array declaration, and the constant in a `case` clause. The C language uses both semantic and syntactic restrictions to ensure this. The semantic restrictions are beyond the scope of this paper, but in order to enforce the syntactic restrictions, the C grammar does something similar to what it does for the comma operator. In places that require a constant expression, instead of the `expression` non-terminal, the `conditional-expression` non-terminal is used. This non-terminal is one further along the precedence hierarchy from the assignment operator and does not contain either the comma or assignment operators. However, while these may be expressed as precedence rules, they have nothing to do with the precedence of the other operators.

Rather than use different non-terminals to enforce these restrictions, we can easily encode both these restrictions as tree automata using a regular expression like the following, where C_2 is a constructor (e.g., comma or assignment) that is not allowed as the ith child of C_i.

$$\neg\,(\mathtt{any}^{*}\ .\ C_1(\__1,\ \cdots,\ \__{i-1},\ \square,\ \__{i+1},\ \cdots,\ \__n)\ .\ C_2(\_,\_))$$

4.1.3 Declarators. In the C grammar, the `abstract-declarator` non-terminal duplicates the `declarator` non-terminal with a few changes. This is due to certain forms not being allowed in certain contexts. This leads to a duplication of seven productions. With tree automata, we could instead use only a single `declarator` form and then specify the restrictions as tree automata. This would also eliminate the ambiguity that the current C grammar has due to `abstract-declarator` and `declarator` both being allowed in `type-name`.

4.2 JavaScript

4.2.1 The `in` Keyword. JavaScript uses the `in` keyword for two different purposes. The first is in the e `in` e form, which tests whether a given object has a given property. The second is in

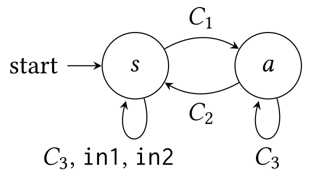


Fig. 16. Tree automaton disambiguating JavaScript's `in`

$$\neg\,(\mathtt{any}^{*}\ .\ (\mathtt{++}_{post}(\square)\ \mid\ \mathtt{--}_{post}(\square))\ .\ (\mathtt{++}_{post}(\square)\ \mid\ \mathtt{--}_{post}(\square)))$$

Fig. 17. Tree automaton restricting JavaScript's postfix operators

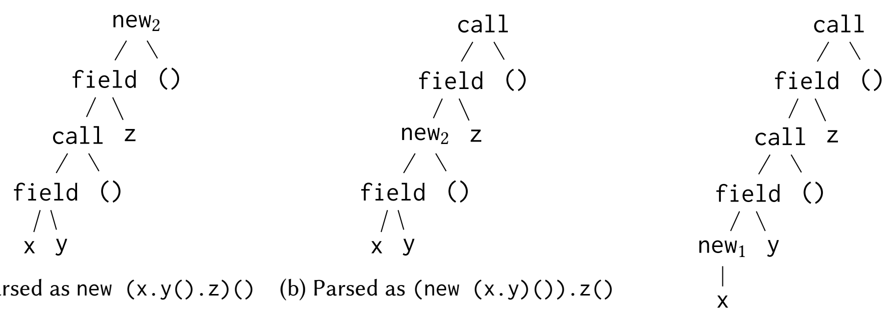
several variations of the `for` statement. For example, `for (`x `in` e`)` s, which iteratively binds x to elements of e and executes s. These two uses could lead to ambiguities, so the JavaScript grammar forbids the e `in` e expression as a descendant of the x or e in the `for` form. However, once inside forms like parentheses, that restriction no longer applies.

This is similar to the comma and assignment operators described in Section 4.1.2. However, the non-terminal defining the e `in` e expression is farther along the precedence hierarchy than in Section 4.1.2. Furthermore, the `for` forms do not participate in the precedence hierarchy. The JavaScript specification handles this by duplicating all of the non-terminals in the precedence hierarchy before e `in` e. One set of these allows e `in` e and is used in places other than these `for` forms. The other set does not allow e `in` e and is used in these `for` forms. This leads to a duplication of thirteen (!) non-terminals.

This restriction can easily be encoded as a tree automaton without needing this duplication. Like with associativity in Section 3.1.2, we simply construct the two-state automaton in Figure 16. The s state is for when the e `in` e expression is allowed, and the a state is for when it is not. C_1 contains those constructors and positions that do not allow the e `in` e expression (e.g., in a `for` construct). C_2 contains those constructors and positions such as parentheses that lift the ban on the `in` expression. C_3 contains all other constructors and child positions. Finally, `in1` and `in2` are the constructors and positions for the e `in` e form.

4.2.2 Precedence. In the JavaScript grammar, most levels of the precedence hierarchy allow child trees to also be at the same precedence as the tree itself. However, the post-increment (suffix `++`) and post-decrement (suffix `--`) operators in JavaScript do not allow this. Their children must be at a strictly tighter precedence. We can encode this as a tree automaton by simply using the normal tree automaton for non-strict precedence that is discussed in Section 3.1.1. We then intersect it with a tree automaton such as in Figure 17 that disallows post-increment and post-decrement from being children of themselves or each other. (This is also an alternate way to encode the restriction in Section 4.1.1.)

4.2.3 The `new` *Keyword.* As mentioned in the example in Section 1, JavaScript allows the first child of the `new` operator to be an arbitrary expression. For example, `new (x.y().z)()` is valid where `x.y().z` is the prototype of the class to be constructed. However, suppose we omit the parentheses around the prototype and have `new x.y().z()`. We could parse this several different ways. We could parse this as before with `x.y().z` as the prototype. Alternatively we could parse this with `x.y` as the prototype and `.z()` being a method call on the constructed object. That would parse this input as if it were `(new (x.y())).z()`. Finally, we could parse this with `x` as the



(a) Parsed as `new (x.y().z)()` (b) Parsed as `(new (x.y)()).z()` (c) Parsed as `(new x).y().z()`

Fig. 18. Possible parse trees for `new x.y().z()`

prototype and `.y().z()` as method calls on the constructed object. That would parse this input as if it were `(new x).y().z()`. These three possible parse trees are shown in Figure 18.

To resolve this ambiguity, JavaScript excludes the use of function calls in the prototype of the `new` operator unless it is enclosed by something like parentheses. This excludes the parse in Figure 18a. JavaScript also excludes the use of the no-argument form of `new` in the function position of a function call. This excludes the parse in Figure 18c. This leaves only Figure 18b as the correct parse. This cannot be specified with traditional precedence declarations as it forbids function application and no-argument `new` from not only being immediate children of the `new` operator and function calls, respectively, but also from being descendants. In the JavaScript grammar, these rules are implemented using the productions in Figure 19. Instead of the usual chain-like structure that we see in a precedence hierarchy, these form the diamond structure in Figure 20.

This restriction is much more easily and intuitively expressed with the tree automata in Figure 21. They are similar to the tree automaton that resolves the functional `if` in Section 3.3 or that prevents comma and assignment operators in some positions in Section 4.1.2. We simply require that a `call` not occur under a new$_2$ and a new$_1$ not occur under a `call`. The `field` and `array` in these automata are because JavaScript allows an arbitrary number of field accesses or array indexings between `call` and new$_2$.

5 BENCHMARKS

To test the practicality of our technique, we collected benchmarks for the grammars in Section 4. In Section 5.1 we discuss basic statistics for the grammars. In Section 5.2 we discuss how quickly these grammars parse input, and in Section 5.3 we discuss how long it takes to generate these grammars from their constituent automata.

5.1 Basic Statistics

Figure 22 shows basic statistics for the C and JavaScript grammars discussed in Section 4. We measured the grammars given in the specifications (original) and our simplified grammars both before (pre-intersection) and after (post-intersection) intersecting with the tree automata.

The pre-intersection grammars reduced the number of non-terminals and productions, but significantly increased the number of LR conflicts. This increase is to be expected given that the tree automata that resolve ambiguities have not been intersected with them yet. The post-intersection grammars had the same number of conflicts as the original grammar and the same or slightly fewer

```
LeftHandSideExpression =
     NewExpression
   | CallExpression
NewExpression =
     MemberExpression
   | ‘new’ NewExpression
CallExpression =
     MemberExpression Arguments
   | CallExpression Arguments
   | CallExpression ‘[’ Expression ‘]’
   | CallExpression ‘.’ IdentifierName
MemberExpression =
     ‘new’ MemberExpression Arguments
   | ...
```

Fig. 19. How JavaScript encodes the restrictions on new

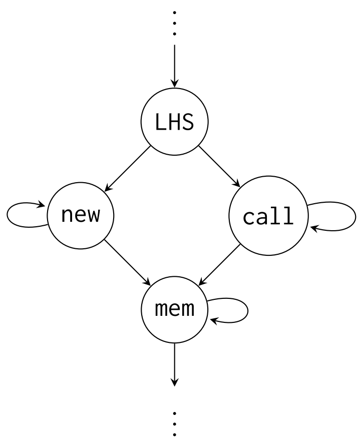


Fig. 20. Graph structure of how JavaScript encodes the restrictions on new. (Edge labels omitted for clarity.)

$$\neg(\texttt{any}^* \;.\; \texttt{new}_2(\square,\ \_) \;.\; (\texttt{field}(\square,\ \_) \mid \texttt{array}(\square,\ \_))^* \;.\; \texttt{call}(\_,\ \_))$$
$$\neg(\texttt{any}^* \;.\; \texttt{call}(\square,\ \_) \;.\; (\texttt{field}(\square,\ \_) \mid \texttt{array}(\square,\ \_))^* \;.\; \texttt{new}_1(\_))$$

Fig. 21. Tree automata resolving ambiguities between JavaScript’s new and call

parse states than the original grammar. The post-intersection grammars differed from the original grammars only due to differences in where to introduce epsilon productions.

5.1.1 Epsilon Production Introduction. The heuristic that we used for introducing epsilon productions is the one described in Section 3.1.3. It iteratively replaces sets of productions with an epsilon production to the non-terminal containing the largest matching subset of productions. In the case of a tie, we chose the set with the fewest productions of the form $N \to_f \epsilon$. This was added due to productions like the following.

$$A \to_f X \quad A \to_g Y \quad A \to_h \epsilon \qquad B \to_f X \quad B \to_g Y \qquad C \to_g Y \quad C \to_h \epsilon$$

Here A and C are likely optional forms of B and Y, respectively, so we want to replace $A \to_f X$ and $A \to_g Y$ with $A \to_\epsilon B$ instead of replacing $A \to_g Y$ and $A \to_h \epsilon$ with $A \to_\epsilon C$. This tie breaker was used four times in the C grammar and one time in the JavaScript grammar.

Minor differences remained between the original and post-intersection grammars even after applying this heuristic. There was one case in the JavaScript grammar of our heuristic doing a factoring that was not in the original grammar. The remaining differences fell into three categories based on the transformation that, if applied to the original grammar, would eliminate the difference. The first two categories involve inlining non-terminals that have only a single production. In the first category, if the production is an epsilon transition, it can be inlined everywhere. In the second category, if the production is a constructor transition, it can be inlined only when the non-terminal is the right-hand side of an epsilon transition. The final category is inlining non-terminals that are used only once if that one use is the right-hand side of an epsilon transition.

To eliminate these differences, we would need to apply the inverse of these operations to the post-intersection grammar. However, there is no clear way to automatically infer where these

	Non-terminals	Productions	GLR States	LR Conflicts
C				
Original	91	240	396	20
Pre-intersection	70	213	354	1003
Post-intersection	76	225	396	20
JavaScript				
Original	93	234	425	4
Pre-intersection	59	176	341	1654
Post-intersection	62	201	399	4

Fig. 22. Basic statistics for the parsers tested in Section 4

should be applied. Fortunately, these differences do not affect the possible parses of the grammar, and as shown in Section 5.2, they do not adversely affect the parsing performance.

5.2 Parsing Performance

In order to check whether our changes affected parser performance, we collected C and JavaScript source code from several open-source code bases that we used as inputs to be parsed. For C these were `less-481`, `sed-4.2.2`, `gzip-1.6`, `shadow-4.2`, `cpio-2.11dfsg`, `bash-4.3`, `grep-2.25`, `tar-1.29b`, `systemd-231`, `busybox-1.22.0`, and `coreutils-8.25`. For JavaScript these were `jquery-3.2.1`, `react-native-0.43.0`, `react-15.4.2`, and `angular.js-1.6.4`. About half the files could not be parsed by the grammar from the standards due to things like language extensions, differing language versions, or preprocessor macros. These were excluded from benchmarking.

On our test platform, several standard headers (e.g., `stdio.h` and `stdlib.h`) used language extensions that are not in the C grammar. Thus any source files that included these headers did not parse. We tried several approaches to resolving this. The one that was most successful was to strip all preprocessor declarations from all source files. This meant that preprocessor macro uses might be parsed as variables or functions calls (in the best case) or cause parse errors (in the worst case).

For benchmarking we used the happy parser generator to generate GLR parsers for both the original and post-intersection grammars and tested their performance with the `criterion` benchmarking library. Inputs were tokenized in advance, so this measurement does not include lexing time. The benchmarks were run on a 2.60GHz Intel Core i7-6700HQ CPU with 32 GB of RAM running Ubuntu 16.10. The tools, compiler, and libraries were taken from the `lts-8.0` snapshot of `stack`. The happy parser generator was run with the `--info` and `--glr` flags in order to produce a GLR parser with a report summarizing the generated parser. All code was compiled with the `-O2` optimizations flag passed to GHC.

The parsing performance of the intersected grammars relative to the original grammars are shown in Figure 23 for C and Figure 24 for JavaScript. There is large amount of scatter in this plot, but this scatter exists in the underlying measurements of the two parsers and does not seem to be due to variable performance between the two implementations. The vertical bands in Figure 24 near 400 and 500 input tokens are due to the locale files for different languages and countries in `angular.js` all having similar structures.

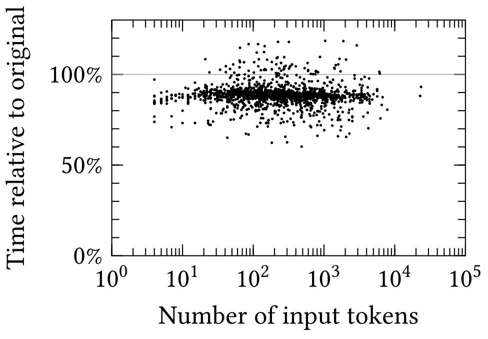


Fig. 23. C parsing times

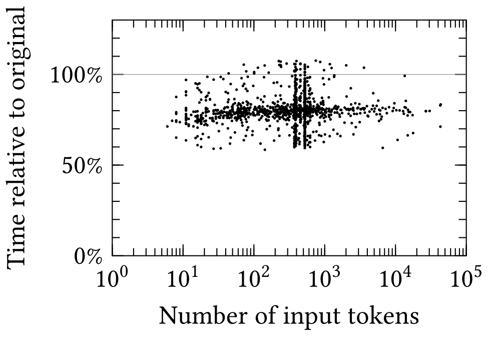


Fig. 24. JavaScript parsing times

Overall the post-intersection grammars for C and JavaScript respectively took 88% and 80% (geometric mean) of the time the originals took. We believe these speedups are due to the elimination of several epsilon transitions as discussed in Section 5.1.1, but did not investigate this more deeply as our goal with this measurement was merely to check if our technique incurs any overhead.

5.3 Grammar Generation Performance

In addition to the parse times in Section 5.2, we examined the time to generate CFGs. We found that the negation operator ($\neg$) caused excessively long processing times when converting from regular expressions to productions. Similar to string regular expressions, converting negation is exponential in the worst case.

While the worst-case is exponential, we optimized our algorithm for the common patterns seen in Section 3 and Section 4. The construction for negation in Comon et al. [2007, Theorem 1.3.1] uses determinization and completion, which are both exponential. Fortunately, while the intermediate automata in this transformation are exponential, the resulting tree automata in Section 3 and Section 4 are not. Given that the final CFGs being generated are not meant to be exponential, it makes intuitive sense that this would tend to be the case. Thus we developed an algorithm that skips the intermediate exponentials and does the transformation in one step.

Given a tree automaton (Q, F, S, δ), our algorithm generalizes the problem to that of finding $\neg \bigcup R$, the complement of the union of a set of states $R \subseteq Q$. For any constructor label $f \in F$, a tree $t = f\left(t_1, t_2, \cdots, t_{|f|}\right)$ is in $\neg \bigcup R$ if and only if for all r in R and $r \rightarrow f\left(q_1, q_2, \cdots, q_{|f|}\right)$ in δ, there is an i such that t_i is not in q_i. Thus, given i_j corresponding to the jth production for f in R, t is in $\neg \bigcup R$ if and only if t is in $\bigcap_j f\left(\_, \cdots, \_, \neg q_{i_j}, \_, \cdots, \_\right)$. Each element of that conjunction rules out a different production in R from being used to match t. If we push that conjunction inside f (by distributivity) and the negation (by De Morgan's law), q_{i_j} from different productions but the same value for i_j combine to form new $\neg \bigcup R'$ that children of f must match. We start this process with $R = S$ and iterate until all needed $\neg \bigcup R$ have been computed.

When run on the tree automata in Section 3 and Section 4, this algorithm still performed poorly, so we improved it by taking advantage of the fact that these automata usually relate to linear paths down the parse tree. Thus, they often focus on one child and use a wild card (_) to ignore its siblings. Since $\neg q_{i_j}$ rejects all trees if q_{i_j} is a wild card, we skip those values for i_j. This greatly reduces the number of values for i_j to be considered.

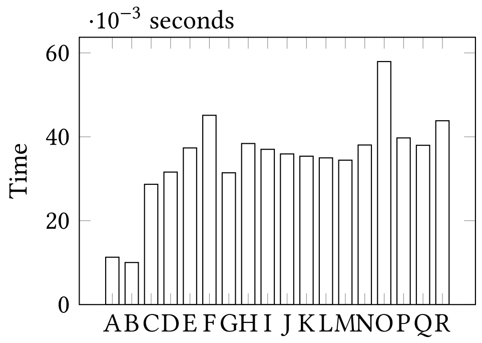


Fig. 25. C grammar generation time. A-B: Declarators (Section 4.1.3). C: Precedence (Section 3.1.1). D: Casts (Section 4.1.1). E: Assignments (Section 4.1.2). F: Commas (Section 4.1.2). G-R: Associativity (Section 3.1.2)

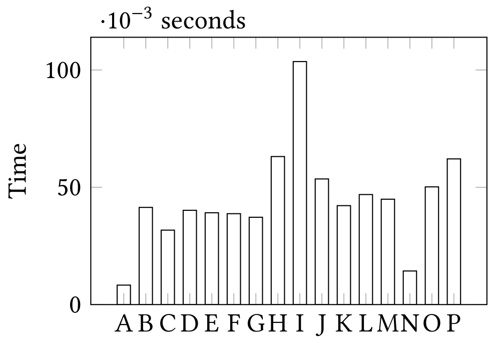


Fig. 26. JavaScript grammar generation time. A: Prefix operators (Section 4.2.2). B: Precedence (Section 3.1.1). C-M: Associativity (Section 3.1.2). N: The `new` keyword (Section 4.2.3). O: Comma (Section 4.1.2). P: The `in` keyword (Section 4.2.1).

This results in the grammar generation times in Figure 25 and Figure 26. Each bar is the time taken as each successive tree automaton further restricts the original grammar. In total this process took 630 milliseconds for C and 720 milliseconds for JavaScript, with an arithmetic mean of 44 and 34 milliseconds per automaton, respectively. Our implementation is relatively naive, so a more sophisticated implementation might perform better. In any case, this is a cost paid when generating the grammar and not when running the parser.

5.4 Error Messages

From a practical point of view, an important consideration is how well the parsers resulting from our technique deal with incomplete or incorrect inputs. We have not rigorously explored this but can make a few observations.

First, the actual process of parsing is handled by whatever parser tool the user chooses. Erroneous inputs are thus handled with all the advantages and disadvantages of that particular tool.

Second, terminal and non-terminal names in the transformed grammar can show up in error messages, and thus it is important that those names be intelligible. Terminal names are taken directly from the original grammar, so they pose no difficulty. On the other hand, non-terminals go through significant transformation.

When intersecting, our implementation uses the Cartesian product of the names of the non-terminals being intersected as the name of the resulting non-terminal. Thus, it produces names like in Figure 11a. The intelligibility of the part of the name that came from the original grammar thus depends on whether the original grammar used sensible names. However, the part of the name from the tree automaton that was intersected posses more difficulty. When, as in Figure 11a, a small number of automata are being intersected, it can take a small but non-trivial amount of thought to understand these non-terminal names. However, the C and JavaScript grammars used 18 and 16 tree automata, respectively. The resulting non-terminal names can be a challenge for the grammar writer to understand let alone a user that is unfamiliar with the grammar.

One way to deal with this would be for the parser to report errors based on the productions used instead of the non-terminals used. Note that while non-terminal names are transformed by intersection, constructor labels are not. They come from the original grammar, and thus their intelligibility depends on whether the original grammar used sensible names. This is an area worth exploring in future work.

6 PROPERTIES

Not only can tree automata specify a variety of grammatical constraints, but they also play nicely with existing grammatical classes. For example, unambiguous grammars remain unambiguous after intersection with bottom-up deterministic tree automata. Likewise, LR(k) languages remain LR(k) after intersection with bottom-up deterministic tree automata.

Of course, already unambiguous or LR(k) grammars may not need restricting, but these results generalize to locally unambiguous or locally LR(k) grammars in the same way that GLR parsing is linear in locally LR(k) parts of a grammar [Lang 1974]. This means that intersecting grammars with bottom-up deterministic tree automata may duplicate ambiguities or conflicts present in the initial grammar but does not introduce new ones. All ambiguities and LR(k) conflicts that exist after intersection can be mapped to corresponding ambiguities or conflicts in the initial grammar that cause them.

These properties apply only when the tree automaton is bottom-up deterministic, but since the *language* class (as opposed to *grammatical* class) for bottom-up deterministic tree automata contains all languages expressible by tree automata [Comon et al. 2007, Theorem 1.1.9, Theorem 1.6.1, and Proposition 1.6.2], similar properties hold for the *language* classes without needing the restriction to bottom-up deterministic tree automata.

6.1 Ambiguities

It is straightforward to show by the following lemma that a bottom-up deterministic tree automaton may eliminate ambiguities in a CFG but never introduces them.

Theorem 6.1. *Intersecting a bottom-up deterministic tree automaton with a CFG never introduces ambiguities not already present in the CFG.*

Proof. In unambiguous parts of a parse tree, there exists only one possible non-terminal for each node in that part of the tree. Likewise, in a bottom-up deterministic tree automaton, there exists only one possible state for each node in that part of the tree. Thus, for a particular node in that part of the tree, the intersection algorithm produces only one possible state for the intersection of the tree automaton and CFG. □

6.2 Closure with LR(k)

As we now show, LR(k) is closed under intersection with bottom-up deterministic tree automata, and furthermore intersection never introduces new shift/reduce or reduce/reduce conflicts to a locally LR(k) grammar.

In the standard construction of an LR(k) parser [Knuth 1965], each state of the machine is represented by an item set, $\mathcal{S}$, where items are of the form $\left[A \to X_1 \cdots X_j \bullet X_{j+1} \cdots X_n ; u\right]$. Here, $A \to X_1 \cdots X_j X_{j+1} \cdots X_n$ is a production in the CFG, A is a non-terminal, $X_1 \cdots X_n$ are terminals or non-terminals, and $u \in \Sigma^k$ is the lookahead string where Σ is the set of terminals.

Let $H_k(W)$ be defined as $H_k(W) = \{a_1 a_2 \cdots a_k \mid \exists u \in \Sigma^* . W \Rightarrow^* a_1 a_2 \cdots a_k u\}$, which computes the possible k-length prefixes of W and where $\Rightarrow^*$ is the transitive closure of the rewrite relation defined by the CFG.

We define the *shift lookahead* for $\mathcal{S}$ as

$$\mathcal{K} = \left\{ v \mid \left[A \to X_1 \cdots X_j \bullet X_{j+1} \cdots X_n ; u\right] \in \mathcal{S}, v \in H_k\left(X_{j+1} \cdots X_n u\right) \right\}$$

where at least one $X_{j+1} \cdots X_n$ contains a non-empty string. For each item in $\mathcal{S}$ of the form $[A \to X_1 \cdots X_n \bullet; u]$, we also define its *reduce lookahead* as u.

In the LR(k) parsing algorithm, a reduce/reduce conflict occurs when there exist two items in $\mathcal{S}$ with the same reduce lookaheads, and a shift/reduce conflict occurs when there exists an item in $\mathcal{S}$ with a reduce lookahead in the shift lookahead for $\mathcal{S}$. Thus, in order to show that a CFG has no new conflicts after intersecting with a bottom-up deterministic tree automaton, it suffices to show that the shift and reduce lookaheads do not overlap any more than they did before the intersection. We show this by proving that the shift and reduce lookaheads are subsets of what they were before intersection.

Lemma 6.2. *For a given item set $\mathcal{S}$ and item in $\mathcal{S}$, the item's shift lookahead is a subset of the corresponding shift lookahead from before intersection.*

Proof. The items in $\mathcal{S}$ all have the form

$$\left[(A, Q) \to (X_1, Y_1) \cdots \left(X_j, Y_j\right) \bullet \left(X_{j+1}, Y_{j+1}\right) \cdots (X_n, Y_n)\, ; u\right]$$

when intersected, where $Q \to_f Y_1 \cdots Y_n$ is a rewrite from the tree automaton and there is some item that before intersection had the form $\left[A \to X_1 \cdots X_j \bullet X_{j+1} \cdots X_n; u\right]$. Because each nonterminal after intersection accepts a subset of words accepted before intersection, the set of words that each (X_i, Y_i) could expand to are a subset of the words X_i can expand to. Thus, $H_k\left(\left(X_{j+1}, Y_{j+1}\right) \cdots (X_n, Y_n)\, u\right)$ is a subset of $H_k\left(X_{j+1} \cdots X_n u\right)$ and the shift lookahead, which is the union of these, is a subset of the original shift lookahead. □

Lemma 6.3. *For a given item set $\mathcal{S}$, its reduce lookahead is a subset of the corresponding reduce lookahead from before intersection.*

Proof. It suffices to show that for each item before intersection, there exists at most one item in $\mathcal{S}$ with the same reduce lookahead. Without loss of generality, let the item before intersection be of the form $[A \to X_1 \cdots X_n \bullet; u]$. Any corresponding item after intersection then has the form $[(A, Q) \to (X_1, Y_1) \cdots (X_n, Y_n) \bullet; u]$ for some $Q \to_f Y_1 \cdots Y_n$ in the tree automaton. By the construction of the LR(k) algorithm, the top elements of the parse stack are $(X_1, Y_1) \cdots (X_n, Y_n)$. As the parse stack is constructed, all items after intersection associated with a particular item from before intersection have the same $Y_1 \cdots Y_n$. Because the tree automaton is bottom-up deterministic, f and $Y_1 \cdots Y_n$ uniquely determine Q. Since Q and $Y_1 \cdots Y_n$ are shared among all post-intersection items associated with a particular pre-intersection item, there can be at most one such item. □

Theorem 6.4. *Intersecting a bottom-up deterministic tree automaton with a CFG does not introduce new LR(k) conflicts.*

Proof. If there are no shift/reduce or reduce/reduce conflicts in a given item set before intersection, the shift and reduce lookaheads do not overlap before intersection. By Lemma 6.2, all shift lookaheads after intersection are subsets of the shift lookaheads before intersection. By Lemma 6.3, all reduce lookaheads after intersection are subsets of the reduce lookaheads before intersection. Thus, intersection does not cause any new conflicts. □

6.3 Language Classes

Finally, we generalize the results in Section 6.1 and Section 6.2 from classes of tree-automata *grammars* to classes of tree-automata *languages*.

By the following lemma, any tree automaton can be converted to be bottom-up deterministic.

Lemma 6.5. *For every tree automaton, there exists an equivalent bottom-up deterministic tree automaton.*

Proof. Proved in Comon et al. [2007, Theorem 1.1.9]. □

Since the language class for bottom-up deterministic tree automata contains all languages expressible by tree automata, Theorem 6.1 and Theorem 6.4 generalize from grammars to languages.

Corollary 6.6. *For any tree automaton and CFG, there exists a CFG that is the intersection of them and does not introduce any ambiguities not present in the original CFG.*

Proof. By Lemma 6.5 along with Theorem 6.1. □

Corollary 6.7. *For any tree automaton and CFG, there exists a CFG that is the intersection of them and does not introduce any LR(k) conflicts not present in the original CFG.*

Proof. By Lemma 6.5 along with Theorem 6.4. □

7 RELATED WORK

A closely related work to ours is by Klarlund et al. [1999]. They develop a logic for reasoning about trees and encode this in tree automata that they run in parallel with the parser. Their logic includes higher-level constructs than ours, but they are all translatable to our operators. Running the tree automata in parallel with the parser incurs a significant overhead. They report using one formula taking three times longer than without, using two formula taking five times longer, and using three formula taking seven times longer. In contrast, our technique encodes the tree automaton into the grammar and pays no such overhead.

Visser [1997] and van den Brand et al. [2002] describe the disambiguation system of SDF. Transitive and non-transitive priorities placed between productions can forbid particular productions from occurring in other productions. However, this restricts only immediate parents or children. Tree automata subsume this but also allow arbitrarily deep recursion. With some effort, SDF's follow restrictions and reject forms are also expressible by tree automata. However, the `avoid` and `prefer` rules are not. Those involve choices between multiple parse trees, while tree automata consider each parse tree separately. Thus `avoid` and `prefer` rules are the only parts of SDF's disambiguation system not subsumed by tree automata.

Thorup [1994] uses tree patterns to exclude certain parses and shows how to modify LR(k) and LL(k) parse tables to exclude parses matching those patterns. The tree patterns are essentially tree fragments with holes that match any tree. Tree patterns are easily expressed with tree automata, but due to supporting Kleene star (e^{*i}), tree automata can have richer structures than are possible with tree patterns.

Afroozeh et al. [2013] develop a pattern notation for illegal derivations. They prove that their system is "safe" in that it excludes only parses that are part of an ambiguity. However, they address only "operator-style ambiguities" and focus on precedence and associativity declarations. Several restrictions that we discuss are not expressible in their system. Examples include C's `declarator` in Section 4.1.3 as it does not involve an ambiguity and the generic operators in Section 3.1.4 as the precedence of generic operators depends on the children of a production and not just the production itself.

Wharton [1976] proposes rules that resolve any ambiguities in a CFG. However, those rules are fixed. The user cannot change them to select the desired parse like is possible with tree automata.

Klint and Visser [1994] develop a taxonomy for describing different disambiguation techniques. In their taxonomy, our technique is incremental, commutative, and generated by a unary predicate. They also propose a technique where patterns are used to exclude certain parses. Their tree patterns are similar to those of Thorup [1994], but they go further by introducing higher-order patterns. These allow arbitrarily deep nesting in patterns, but do not support the recursive patterns expressible with tree automata.

Thorup [1996] is also based on excluding parse trees that match patterns but generalizes patterns to support recursion. This technique has many parallels to ours with some key differences. Our technique builds on the rich theory of tree automata. For example, there are algorithms for taking the complement of tree automata, so we can express both required and forbidden patterns. In theory, the same can be done for Thorup [1996], but it is not explored in that paper. We also go beyond that work by proving that tree automata are well behaved with respect to common language categories.

8 CONCLUSION

The rising use of domain-specific languages (often written by non-experts) and language tool benches (e.g., Raskell, Spoofax, and SDF) makes being able to modularly specify grammars a more important issue than ever. In these settings, grammars are software artifacts that need to be maintained, modified, and extended. Thus, these grammars need to be modular for the same reasons other code needs to be modular. We should not ask users to use the traditional approach of restructuring their grammars for what are ultimately technical concerns.

We have shown how tree automata improve this situation by modularly specifying grammatical restrictions. This forms a unified theory that subsumes many other techniques. Simple tree automata that are expressible in a few lines encode many kinds of restrictions including both classic problems and ones found in real-world languages such as C and JavaScript. Finally, we show that tree automata are well behaved with regard to common language categories and perform well in practice.

ACKNOWLEDGMENTS

This article reports on work supported by the Defense Advanced Research Projects Agency under agreement no. AFRL FA8750-15-2-0092. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

REFERENCES

Ali Afroozeh, Mark van den Brand, Adrian Johnstone, Elizabeth Scott, and Jurgen Vinju. *Safe Specification of Operator Precedence Rules*, pages 137–156. Springer, Cham, 2013. ISBN 978-3-319-02654-1. doi: 10.1007/978-3-319-02654-1_8.

Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, October 2007. URL http://www.grappa.univ-lille3.fr/tata.

ECMA 2011. *Standard ECMA-262: ECMAScript Language Specification.* Ecma International, 5.1 edition, June 2011. URL http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

ISO/IEC 9899:TC3: Programming languages – C. ISO/IEC JTC1/SC22/WG14, September 2007. URL http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf.

Nils Klarlund, Niels Damgaard, and Michael I. Schwartzbach. Yakyak: parsing with logical side constraints. In Grzegorz Rozenberg and Wolfgang Thomas, editors, *Developments in Language Theory, Foundations, Applications, and Perspectives, Aachen, Germany, 6-9 July 1999*, pages 286–301. World Scientific, 1999. ISBN 981-02-4380-4.

Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. Technical Report P9426, University of Amsterdam, December 1994.

Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, December 1965. ISSN 0019-9958. doi: 10.1016/S0019-9958(65)90426-2.

Bernard Lang. *Deterministic techniques for efficient non-deterministic parsers*, pages 255–269. Springer, Berlin, Heidelberg, 1974. ISBN 978-3-540-37778-8. doi: 10.1007/3-540-06841-4_65.

Mikkel Thorup. Controlled grammatic ambiguity. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16 (3):1024–1050, May 1994. ISSN 0164-0925. doi: 10.1145/177492.177759.

Mikkel Thorup. Disambiguating grammars by exclusion of sub-parse trees. *Acta Informatica*, 33(5):511–522, August 1996. ISSN 0001-5903 (Print) 1432-0525 (Online). doi: 10.1007/BF03036460.

Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. *Disambiguation Filters for Scannerless Generalized LR Parsers*, pages 143–158. Springer, Berlin, Heidelberg, 2002. ISBN 978-3-540-45937-8. doi: 10.1007/3-540-45937-5_12.

Eelco Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.

R. M. Wharton. Resolution of ambiguity in parsing. *Acta Informatica*, 6(4):387–395, December 1976. ISSN 0001-5903 (Print) 1432-0525 (Online). doi: 10.1007/BF00268139.