# Disambiguating Grammars with Tree Automata

Michael D. Adams
University of Utah

Matthew Might
University of Utah

## Abstract

Dealing with precedence and associativity rules in context free grammars (CFGs) has long posed a challenge. While they can be encoded in the structure of the CFG, the result can be difficult to work with and often obfuscates the language designer's intent. Many parsing systems offer alternatives in the form of specialized precedence and associativity declarations, but these are limited and do not handle many similar situations such as the special rules surrounding a dangling `else`, the precedence of `if` in ML, or the interactions between `new` and function arguments in JavaScript.

In this presentation, we show that tree automata can specify all of these while still allowing the CFG to be written in a natural manner. This unified theory subsumes and generalizes these other techniques and provides an elegant means of specifying grammatical restrictions.

When applied to an existing CFG, this technique generates a new CFG that encodes the constraints from a given tree automata. This process is closed for LR($k$) and LL($k$) grammars and thus can be used as a preprocessing step that is compatible with existing parsing frameworks.

***Categories and Subject Descriptors*** D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.3.4 [*Programming Languages*]: Processors—Parsing

***General Terms*** Algorithms; Languages

***Keywords*** Parsing; Tree Automata

## 1. Applying Tree Automata to Grammars

Tree Automata (TA) are an elegant method of specifying constraints on context free grammars (CFG) such as those that determine associativity and precedence and those that resolve issues like the dangling `else` problem. In this presentation, we give several examples of this and show how this unified approach resolves several classic problems in this area. We then present two case studies on the use of TA to specify exotic language constraints found common languages such as C and JavaScript.

In each of these cases, this is accomplished by intersecting the CFG with a TA. The result of this intersection is another CFG that encodes the original CFG restricted to parse trees accepted by the TA. Thus this approach can act as a front-end on any other parsing system that accepts CFGs. Notably, unlike with intersecting CFGs with other CFGs, intersecting a CFG with a TA is decidable and can be efficiently computed.

## 2. A Simple Example: Precedence and Associativity

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

```
Exp → nat
    | Exp "+" Exp
    | Exp "*" Exp
    | "(" Exp ")"
```

**Figure 1.** Example Grammar

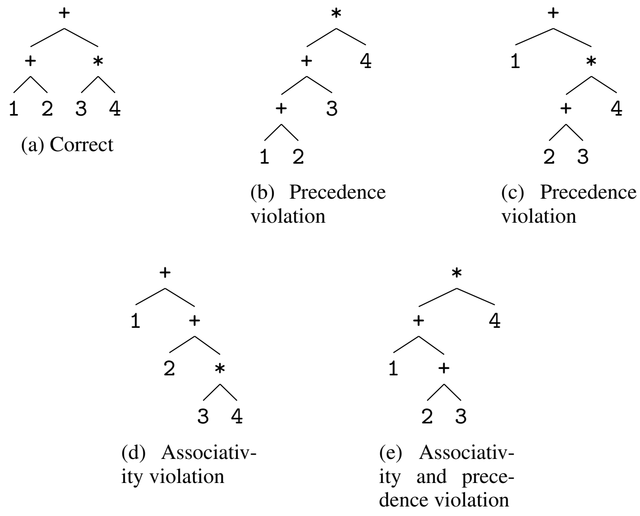
(a) Correct
(b) Precedence violation
(c) Precedence violation
(d) Associativity violation
(e) Associativity and precedence violation

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

```
Exp →Nat   ϵ
Exp →Plus  Exp Exp
Exp →Times Exp Exp
Exp →Paren Exp
```

**Figure 3.** Tree automata for the CFG in Figure 1

In order to use TA to express these restrictions, we first map the grammar to a TA and assign a label to each production. For the sake of concision, we will omit terminals from these TA and thus we have the resulting tree automata represented by the grammar in Figure 3. We will add these terminals back when we convert it back into a CFG.

### 2.1 Precedence

First, consider precedence, which involves enforcing restrictions between the `Plus` and `Times` constructors. Specifically, once inside a `Times`, we must not use a `Plus` until we have used some other

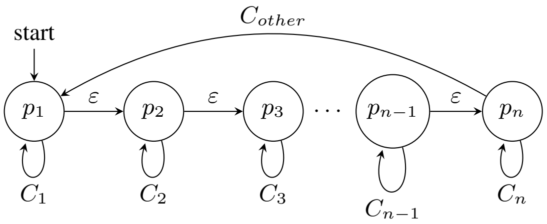

**Figure 4.** General TA for precedence

```
Exp  →Nat   ϵ
Exp  →Plus  Exp Exp
Exp  →Times Exp1 Exp1
Exp  →Paren Exp

Exp1 →Nat   ϵ
Exp1 →Times Exp1 Exp1
Exp1 →Paren Exp
```

**Figure 5.** TA for precedence

constructor such as `Paren`. For example, this rejects the parses in Figure 2b and Figure 2c.

For the general case, a TA encoding precedence looks like in Figure 4 where $p_1, p_2, \cdots, p_n$ are states for each precedence level, each $C_i$ is the set of constructors (i.e., production labels) allowed at that precedence level and $C_{other}$ is all other constructors which allow us to go back to the start of the precedence hierarchy. This meshes nicely with our intuitive notion of ascending precedence.

For the relatively simple case of `Plus` and `Times` we then get the TA represented by as a grammar in Figure 5.

### 2.2 Associativity

Now consider how to impose the left associativity rules for `Plus` and `Times`. To do this we want to exclude the cases where `Plus` occurs as the right child of another `Plus` and likewise for `Times`. These can be expressed by the rule that no part of the parse tree may match either of the following:

```
Plus(_, Plus(_, _))
Times(_, Times(_, _))
```

These follow similar patterns and can both be encoded as the automaton in Figure 6 where $C_i$ is the constructor paired with the child number in which another occurrence of that same constructor is not allowed (in this case `Plus` and `Times` respectively), $C_j$ is the set of those same constructors paired with the child numbers that do *not* restrict their children, and $C_{other}$ is the set of all other constructors. The $S$ state represents when the associativity rule is not restricting the grammar, and the $A$ state represents when associativity needs to prohibit certain constructors.

The grammar representation for these is then as in Figure 7 and Figure 8.

### 2.3 Merging with the Grammar

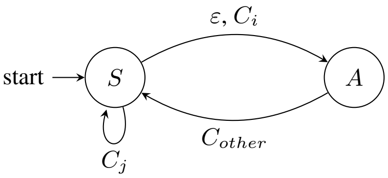

**Figure 6.** TA for associativity

```
Exp  →Nat   ϵ
Exp  →Plus  Exp Exp+
Exp  →Times Exp Exp
Exp  →Paren Exp

Exp+ →Nat
Exp+ →Times Exp Exp
Exp+ →Paren Exp
```

**Figure 7.** TA for associativity of `Plus`

```
Exp  →Nat   ϵ
Exp  →Plus  Exp Exp
Exp  →Times Exp Exp*
Exp  →Paren Exp

Exp* →Nat
Exp* →Plus  Exp Exp
Exp* →Paren Exp
```

**Figure 8.** TA for associativity of `Times`

```
Exp   →Nat   ϵ
Exp   →Plus  Exp Exp+
Exp   →Times Exp1 Exp*1
Exp   →Paren Exp

Exp+  →Nat   ϵ
Exp+  →Times Exp Exp*1
Exp+  →Paren Exp

Exp1  →Nat   ϵ
Exp1  →Times Exp1 Exp*1
Exp1  →Paren Exp

Exp*1 →Nat   ϵ
Exp*1 →Paren Exp
```

**Figure 9.** Intersected TA

Finally, the TA for precedence and associativity can be combined with the TA in Figure 3 by using known algorithms for intersecting TA. The result is the TA represented by the grammar in Figure 9. This process allows us to modularly specify grammar constraints and automatically combine them into a single CFG that encodes all of them. Thought the examples of precedence and associativity are fairly simple, this same technique can be used to specify more complicated constraints such as those for ML `if`, the dangling `else` problem, and even more exotic constraints found in common languages such as C and JavaScript.

## Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant Number 1248464 and the Defense Advanced Research Projects Agency under agreement no. AFRL FA8750-15-2-0092. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.