Disambiguating Grammars with Tree Automata[1]

Michael D. Adams Matthew Might

University of Utah

Parsing@SLE
November 1, 2015

[1]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.

Disambiguation Problems

- Precedence and Associativity
- Dangling `else`
- ML `if`
- JavaScript constructors

Disambiguation Problems

- Precedence and Associativity
  - Domain specific annotations
  - Doesn't cover C:
    - Invalid: `++ (int) x`
    - Valid: `* (int*) x`
    - Valid: `++ * (int*) x`
- Dangling `else`
- ML `if`
- JavaScript constructors

Disambiguation Problems

- Precedence and Associativity
  - Domain specific annotations
  - Doesn't cover C:
    - Invalid: `++ (int) x`
    - Valid: `* (int*) x`
    - Valid: `++ * (int*) x`
- Dangling `else`
  - Shift/reduce bias
- ML `if`
- JavaScript constructors

Disambiguation Problems

- Precedence and Associativity
  - Domain specific annotations
  - Doesn't cover C:
    - Invalid: `++ (int) x`
    - Valid: `* (int*) x`
    - Valid: `++ * (int*) x`
- Dangling `else`
  - Shift/reduce bias
- ML `if`
  - SDF annotations
- JavaScript constructors

Disambiguation Problems

- Precedence and Associativity
  - Domain specific annotations
  - Doesn't cover C:
    - Invalid: `++ (int) x`
    - Valid: `* (int*) x`
    - Valid: `++ * (int*) x`
- Dangling `else`
  - Shift/reduce bias
- ML `if`
  - SDF annotations
- JavaScript constructors
  - Complicated grammar

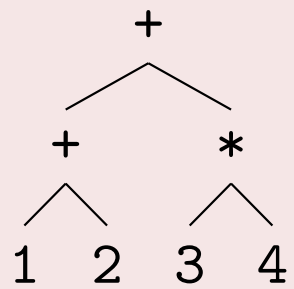
Disambiguation Problems

- Precedence and Associativity
  - ~~Domain specific annotations~~ Tree automata
  - ~~Doesn't cover C:~~ Tree automata cover C:
    - Invalid: `++ (int) x`
    - Valid: `* (int*) x`
    - Valid: `++ * (int*) x`
- Dangling `else`
  - ~~Shift/reduce bias~~ Tree automata
- ML `if`
  - ~~SDF annotations~~ Tree automata
- JavaScript constructors
  - ~~Complicated grammar~~ Tree automata

Tree automata

Basic Idea

- String automata: 1234
- Tree automata:



Properties

- Closed under union, intersection and negation
- Never adds ambiguities
- LR(k) closed
- Conjecture: Produces optimal grammars

Tree Automata

```
1+2+3*4
```

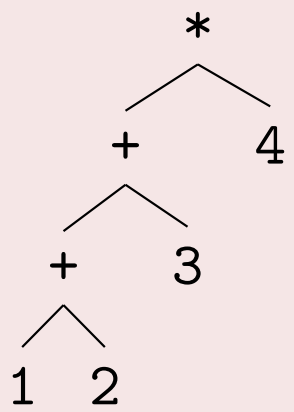
Grammar

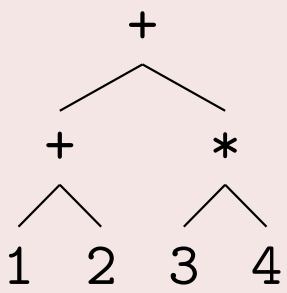
```
Exp → nat
Exp → Exp "+" Exp
Exp → Exp "*" Exp
Exp → "(" Exp ")"
```

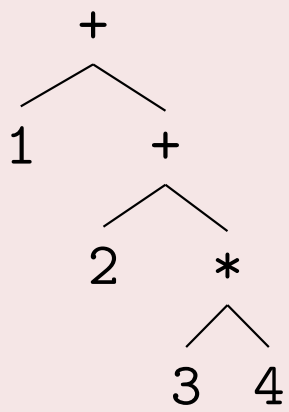
Tree Automata

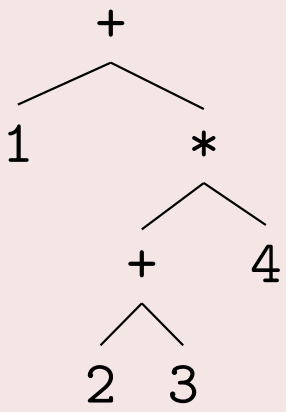
1+2+3*4

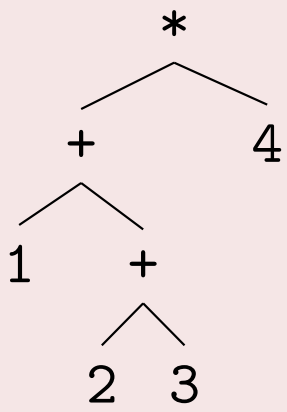
Trees











Tree Automata

```
1+2+3*4
```

Grammar

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

Tree Automata

```
1+2+3*4
```

Grammar

$\texttt{Exp} \rightarrow_{\mathbb{N}}$ `nat`

$\texttt{Exp} \rightarrow_{+}$ `Exp "+" Exp`

$\texttt{Exp} \rightarrow_{*}$ `Exp "*" Exp`

$\texttt{Exp} \rightarrow_{()}$ `"(" Exp ")"`

Tree Automata

1+2+3*4

Grammar

$$
\begin{array}{lll}
\texttt{Exp} & \rightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp} & \rightarrow_{\texttt{+}} & \texttt{Exp} \quad \texttt{Exp} \\
\texttt{Exp} & \rightarrow_{\texttt{*}} & \texttt{Exp} \quad \texttt{Exp} \\
\texttt{Exp} & \rightarrow_{\texttt{()}} & \texttt{Exp}
\end{array}
$$

Tree Automata

1+2+3*4

Grammar

$$
\begin{array}{lll}
\texttt{Exp} & \rightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp} & \rightarrow_{\texttt{+}} & \texttt{Exp} \quad \texttt{Exp} \\
\texttt{Exp} & \rightarrow_{\texttt{*}} & \texttt{Exp} \quad \texttt{Exp} \\
\texttt{Exp} & \rightarrow_{\texttt{()}} & \texttt{Exp}
\end{array}
$$

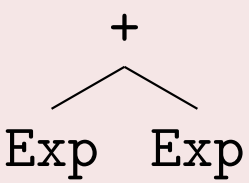
$$\texttt{Exp}$$

Tree Automata

1+2+3*4

Grammar

$$
\begin{array}{lll}
\texttt{Exp} & \rightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp} & \rightarrow_{\texttt{+}} & \texttt{Exp} \quad \texttt{Exp} \\
\texttt{Exp} & \rightarrow_{\texttt{*}} & \texttt{Exp} \quad \texttt{Exp} \\
\texttt{Exp} & \rightarrow_{\texttt{()}} & \texttt{Exp}
\end{array}
$$

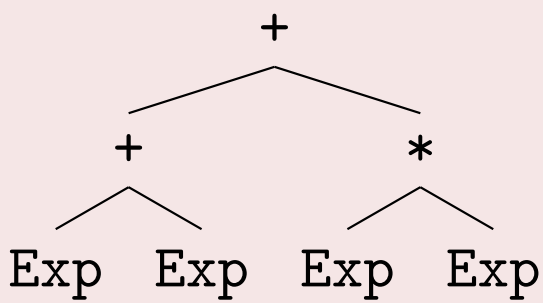


Tree Automata

1+2+3*4

Grammar

$$
\begin{array}{lll}
\texttt{Exp} & \rightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp} & \rightarrow_{\texttt{+}} & \texttt{Exp} \quad \texttt{Exp} \\
\texttt{Exp} & \rightarrow_{\texttt{*}} & \texttt{Exp} \quad \texttt{Exp} \\
\texttt{Exp} & \rightarrow_{\texttt{()}} & \texttt{Exp}
\end{array}
$$

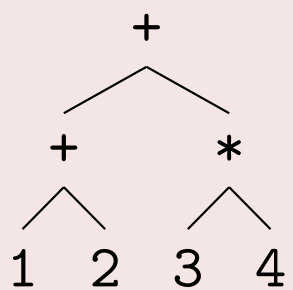


Tree Automata

1+2+3*4

Grammar

$$
\begin{array}{lll}
\texttt{Exp} & \rightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp} & \rightarrow_{\texttt{+}} & \texttt{Exp} \quad \texttt{Exp} \\
\texttt{Exp} & \rightarrow_{\texttt{*}} & \texttt{Exp} \quad \texttt{Exp} \\
\texttt{Exp} & \rightarrow_{\texttt{()}} & \texttt{Exp}
\end{array}
$$



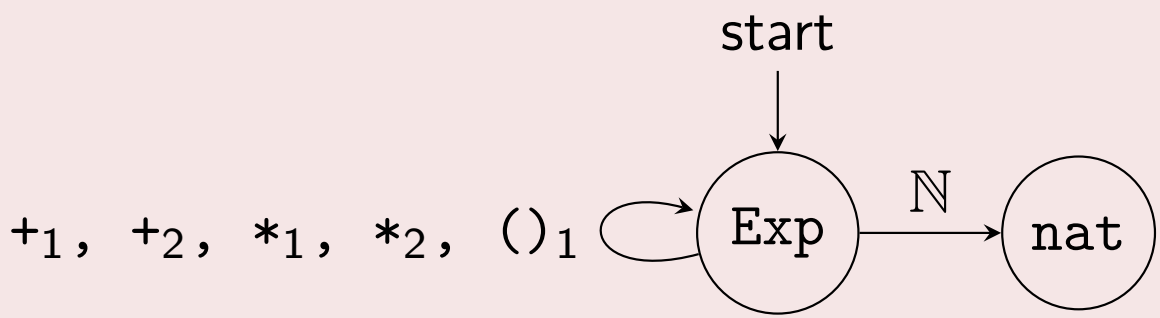
Tree Automata

1+2+3*4

Grammar

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

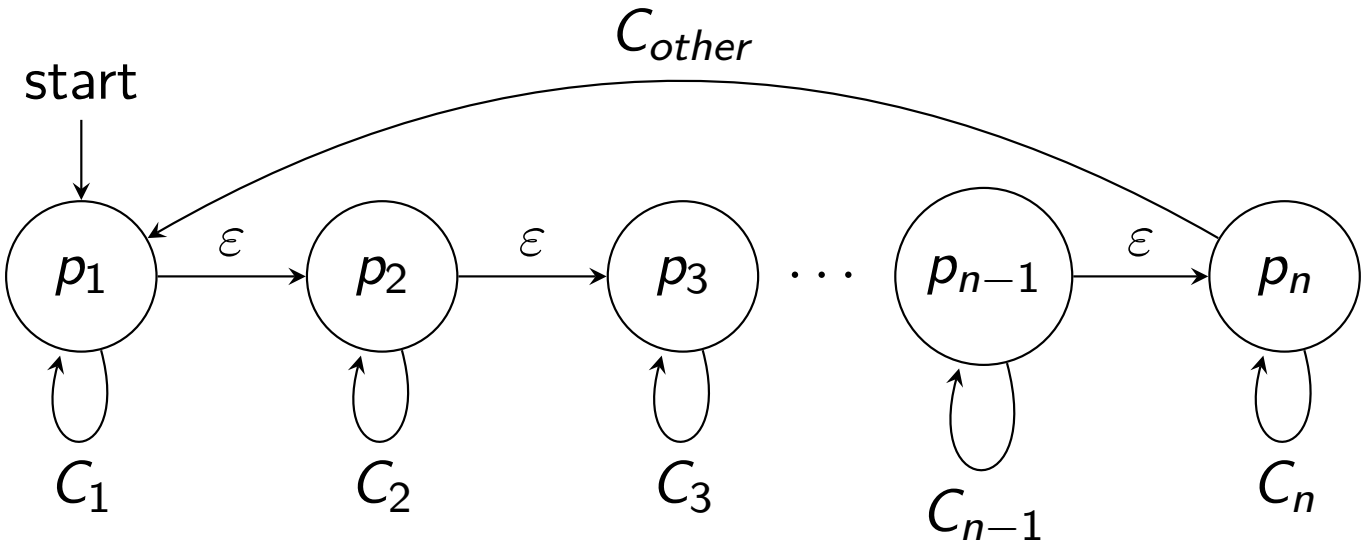
Automaton



Regular Expression

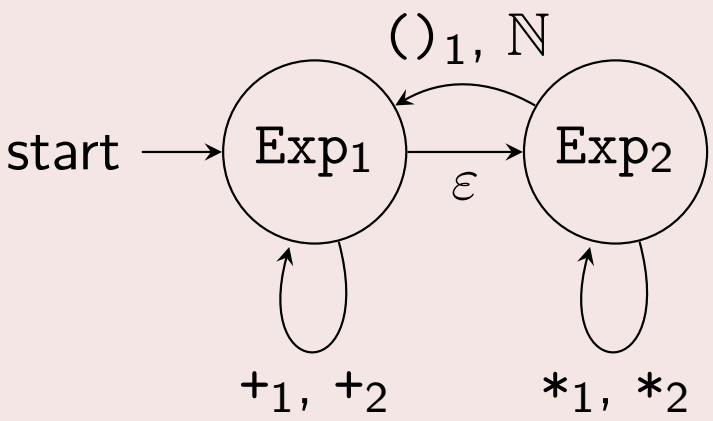
$$(+(\square,\ \square)\ |\ *(\square,\ \square)\ |\ ()(\square))^{*}\ .\ \mathbb{N}$$

Example: Precedence



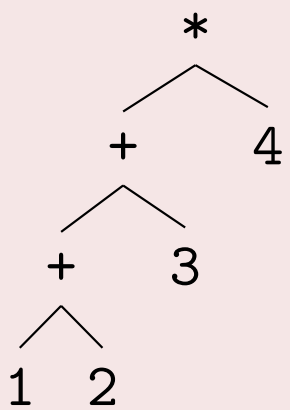
Example: Precedence

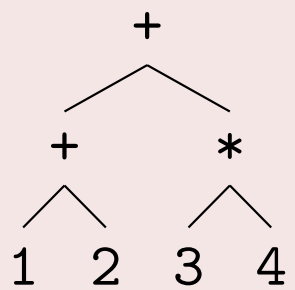
Automaton



Grammar

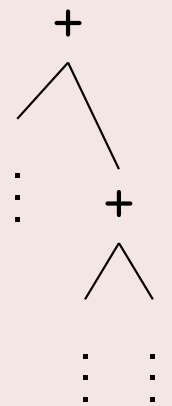
$$\begin{aligned}
\mathtt{Exp}_1 &\to_{+} \mathtt{Exp}_1\ \mathtt{Exp}_1 \\
\mathtt{Exp}_1 &\to_{\epsilon} \mathtt{Exp}_2 \\
\\
\mathtt{Exp}_2 &\to_{\mathbb{N}} \epsilon \\
\mathtt{Exp}_2 &\to_{*} \mathtt{Exp}_2\ \mathtt{Exp}_2 \\
\mathtt{Exp}_2 &\to_{()} \mathtt{Exp}_1
\end{aligned}$$

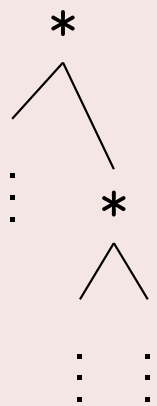


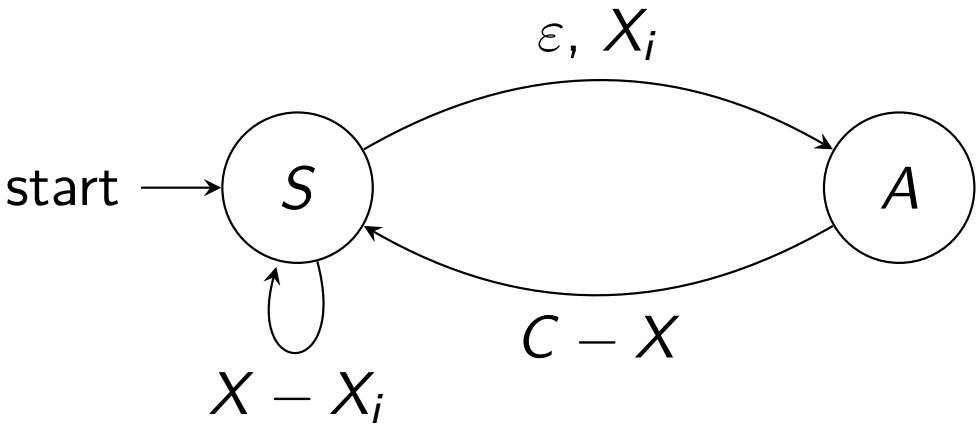


Example: Associativity

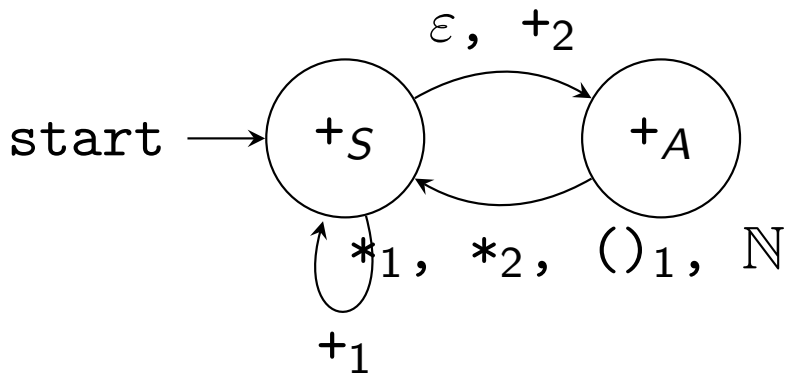
Forbidden

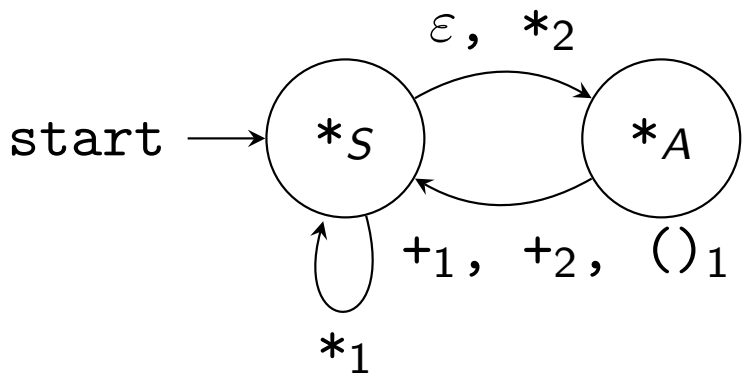






Example: Associativity





$$\begin{aligned}
\mathtt{Exp} &\to_{+} \mathtt{Exp}\ \mathtt{Exp}+ \\
\mathtt{Exp} &\to_{\epsilon} \mathtt{Exp}+
\end{aligned}$$

$$\begin{aligned}
\mathtt{Exp}+ &\to_{\mathbb{N}} \epsilon \\
\mathtt{Exp}+ &\to_{*} \mathtt{Exp}\ \mathtt{Exp} \\
\mathtt{Exp}+ &\to_{()} \mathtt{Exp}
\end{aligned}$$

$$\begin{aligned}
\mathtt{Exp} &\to_{*} \mathtt{Exp}\ \mathtt{Exp}* \\
\mathtt{Exp} &\to_{\epsilon} \mathtt{Exp}*
\end{aligned}$$

$$\begin{aligned}
\mathtt{Exp}* &\to_{\mathbb{N}} \\
\mathtt{Exp}* &\to_{+} \mathtt{Exp}\ \mathtt{Exp} \\
\mathtt{Exp}* &\to_{()} \mathtt{Exp}
\end{aligned}$$

Example: Combined grammar

Combined

$$
\begin{array}{lll}
\texttt{Exp}_1 & \rightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp}_1 & \rightarrow_{+} & \texttt{Exp}_1\ \texttt{Exp}_{1+} \\
\texttt{Exp}_1 & \rightarrow_{*} & \texttt{Exp}_2\ \texttt{Exp}_{2*} \\
\texttt{Exp}_1 & \rightarrow_{()} & \texttt{Exp}_1 \\
 & & \\
\texttt{Exp}_{1+} & \rightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp}_{1+} & \rightarrow_{*} & \texttt{Exp}_2\ \texttt{Exp}_{2+} \\
\texttt{Exp}_{1+} & \rightarrow_{()} & \texttt{Exp}_1 \\
 & & \\
\texttt{Exp}_2 & \rightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp}_2 & \rightarrow_{*} & \texttt{Exp}_2\ \texttt{Exp}_{2*} \\
\texttt{Exp}_2 & \rightarrow_{()} & \texttt{Exp}_1 \\
 & & \\
\texttt{Exp}_{2*} & \rightarrow_{\mathbb{N}} & \epsilon \\
\texttt{Exp}_{2*} & \rightarrow_{()} & \texttt{Exp}_1
\end{array}
$$

Simplified

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

Usage

Grammar

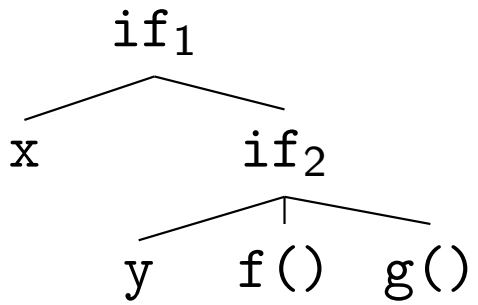
```
Exp →ℕ  nat
Exp →+  Exp "+" Exp
Exp →*  Exp "*" Exp
Exp →()  "(" Exp ")"
```

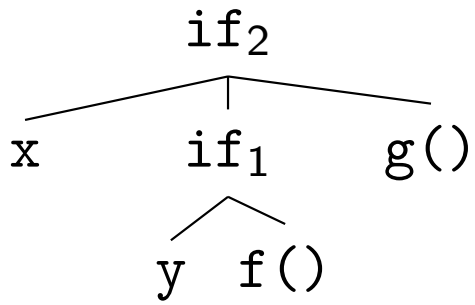
Restrictions

```
Precedence: ((+(□, □))*
              . (*(□, □))*
              . (nat | ()(□))*)*
Assoc. for +: not EF(+(_, +(_, _)))
Assoc. for *: not EF(*(_, *(_, _)))
```

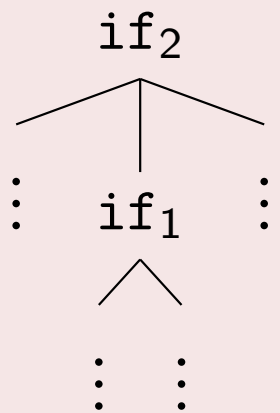
Example: Dangling `else`

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





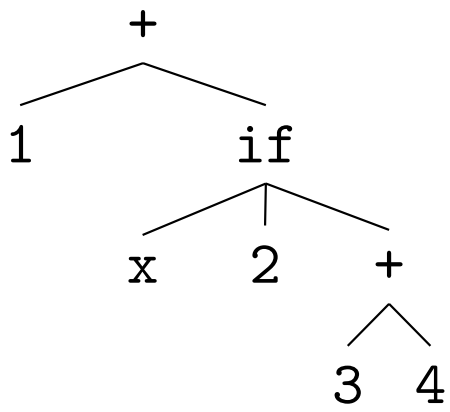
Forbidden

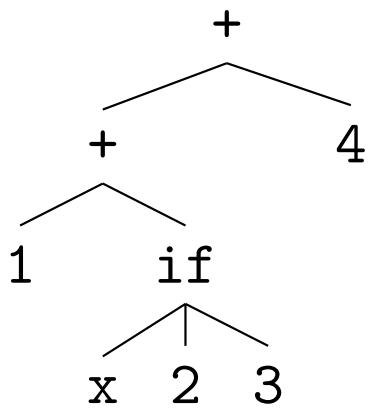



```
not (EF (if2(_, □, _)
         . if1(_, _)))
```

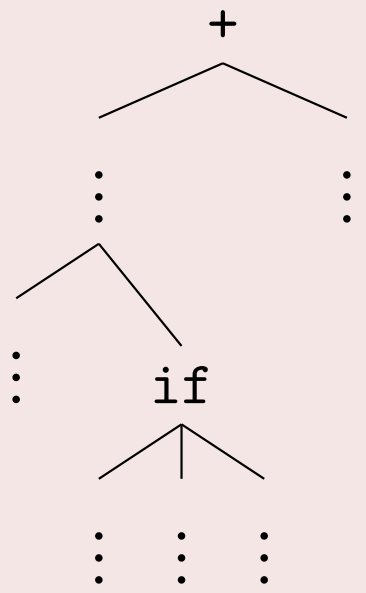
Example: ML's `if`

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





Forbiden

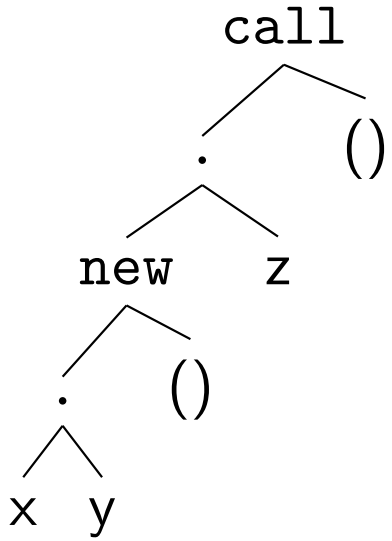


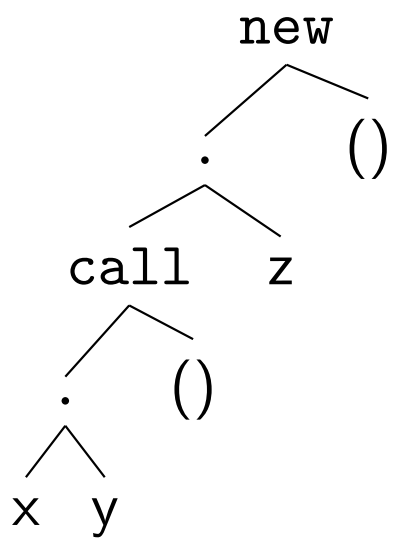
$$\texttt{not (EF (+(}\square\texttt{, \_)} \cdot \texttt{R}^* \cdot \texttt{if(\_, \_, \_))))}$$

$$\text{where } \texttt{R} = \texttt{+(\_, }\square\texttt{)} \mid \texttt{-(\_, }\square\texttt{)} \mid \cdots$$

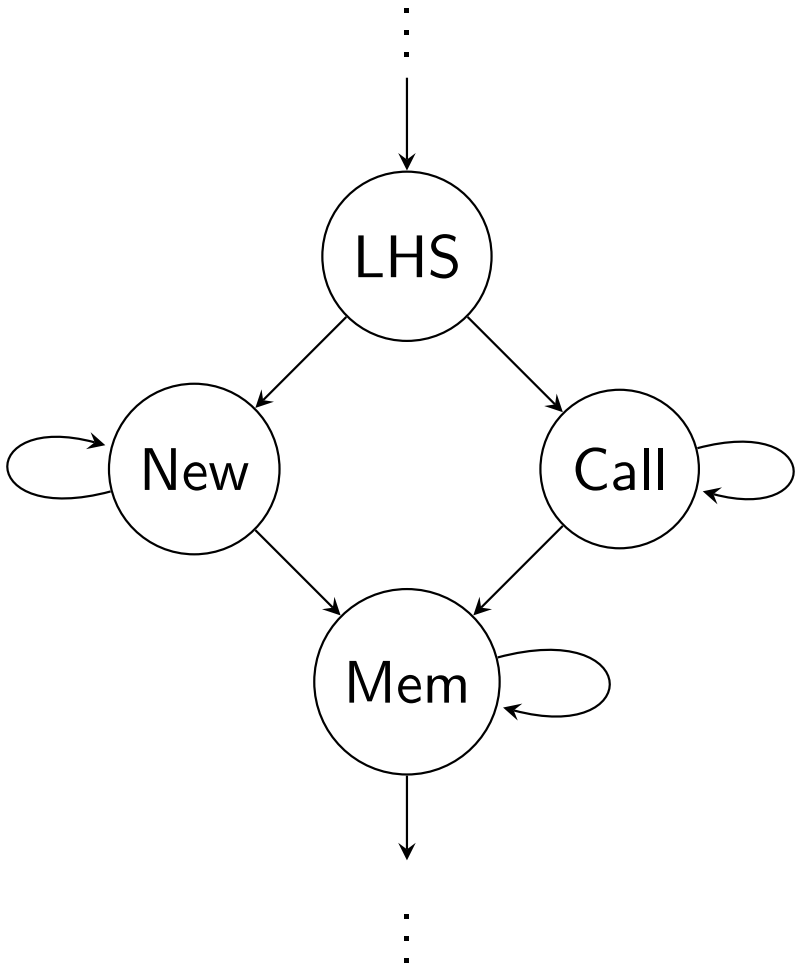
Example: JavaScript

```
new x.y().z()
```





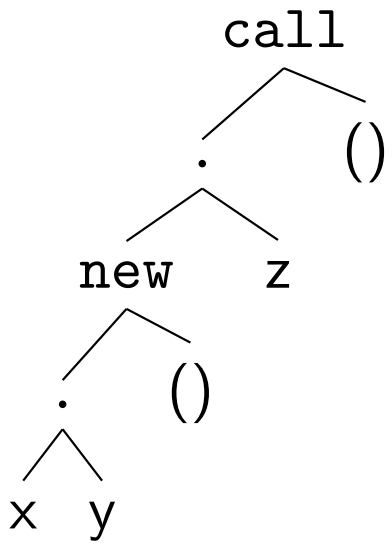
Example: JavaScript

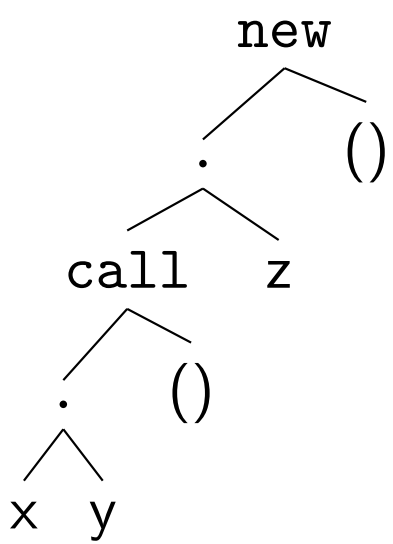


Example: JavaScript

Tree Automata

```
not (EF (New(□, _)
         . (Field(□, _) | Array(□, _))*
         . Call(_, _)))
```





Using Tree Automata

Theorem

Intersecting a TA with a CFG does not introduce ambiguities.

Theorem

LR(k) is closed under intersection with a TA.

Conjecture

The minimization and bottom-up determinization of a CFG results in the fewest LR(k) states of any CFG producing the same parse trees.

Summary

Using TA to restrict grammars

- Write your grammar the "natural" way
- Encode grammatical restrictions as TA
- Intersect the grammar with the TA
- Feed the result to any parsing system you want

Open Question

Does minimization and determinization produce an optimal LR parser?