CSCI 742 - Compiler Construction

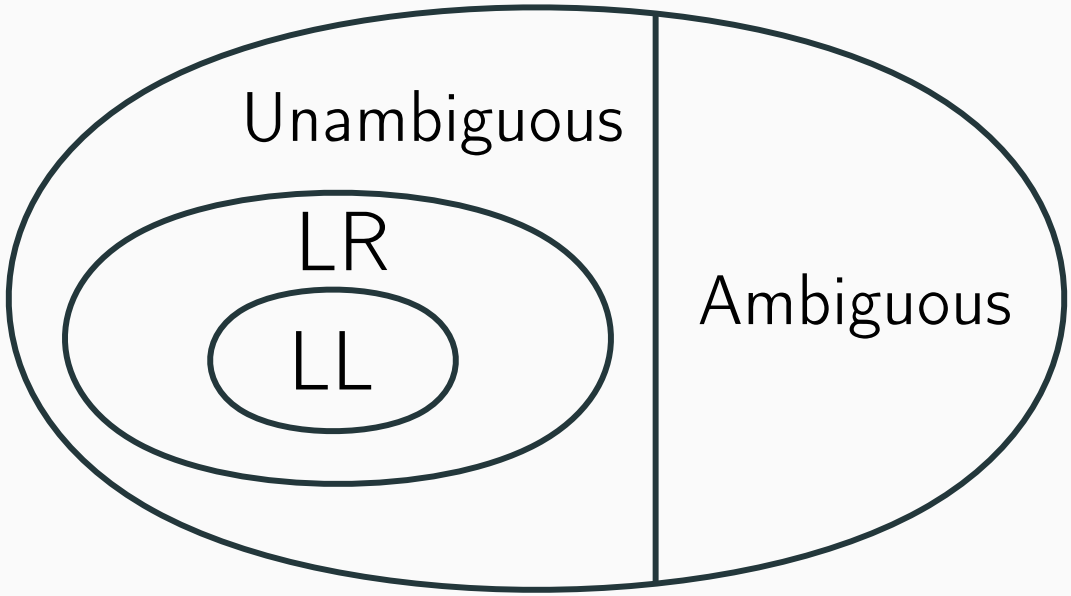
Lecture 11

Grammar Transformations

Instructor: Hossein Hojjat

February 9, 2017

Space of Context-free Grammars

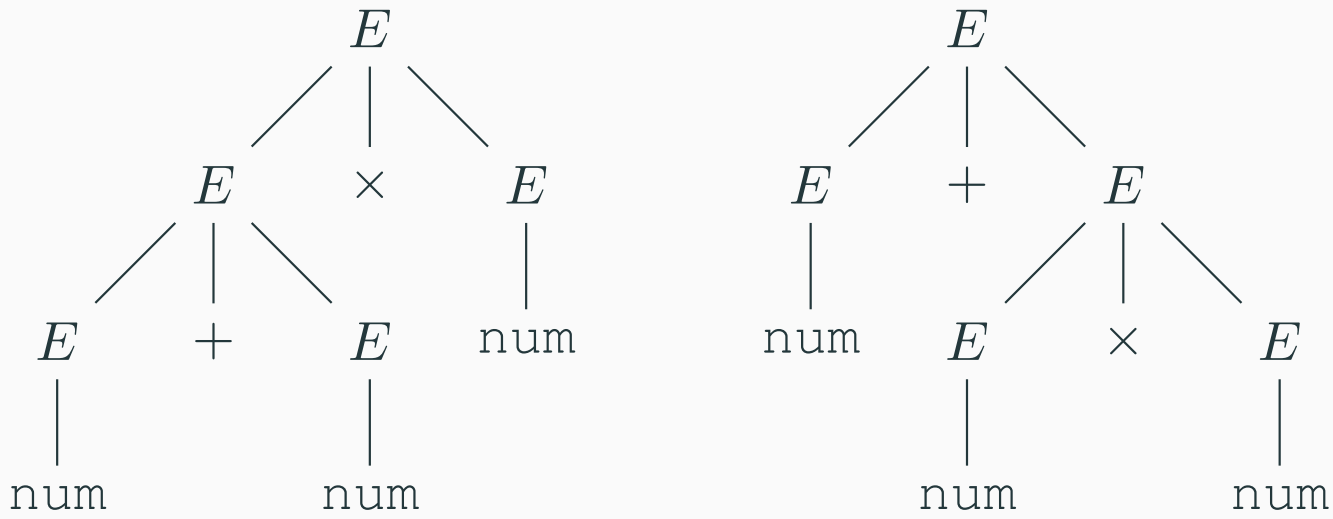


Context-Free Grammars (CFG)

Recap: Ambiguity

- Ambiguous grammar: a word has more than one parse tree
- There is no algorithm to decide if a grammar is ambiguous

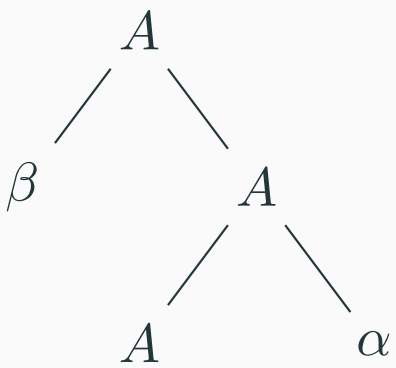
$$E \to E + E$$
$$E \to E \times E$$
$$E \to \texttt{num}$$

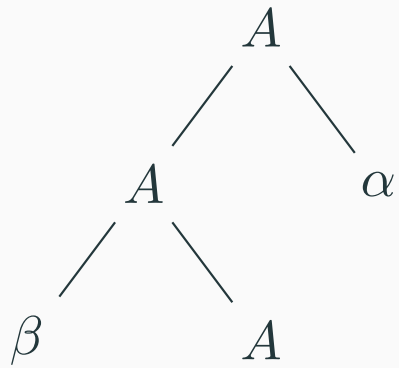


Two parse trees for $\texttt{num} + \texttt{num} \times \texttt{num}$

Ambiguity Patterns

- There are common ambiguity patterns in context-free grammars
- Example:
- If a non-terminal $A \in N$ is both left-recursive and right-recursive then G is ambiguous
  - Left-recursive: it has a derivation $A \Rightarrow^+ A\alpha \quad (\alpha \in (N \cup T)^+)$
  - Right-recursive: it has a derivation $A \Rightarrow^+ \beta A \quad (\beta \in (N \cup T)^+)$

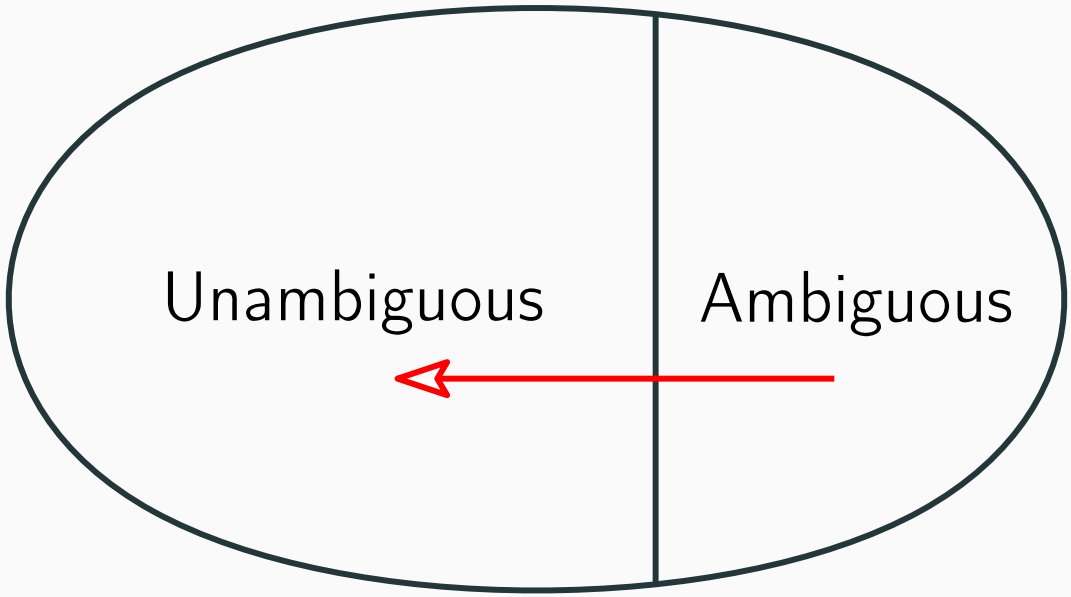




Resolving Ambiguity

- Designing unambiguous grammars is usually tricky
- Sometimes it is possible to eliminate ambiguity by rewriting grammar
  - similar to Chomsky normal form conversion
- Occasionally more natural grammar is the ambiguous one
- Parser generators allow disambiguating declarations for ambiguous grammars

Eliminating Ambiguity



Goal: transform an ambiguous grammar to an equivalent unambiguous grammar

Equivalent Grammars

- Grammars G_1 and G_2 are equivalent if they generate the same language

$$L(G_1) = L(G_2)$$

- In other words if a sentence can be derived from one of the grammars it can be derived also from other grammar

Common Examples of Ambiguity

- Operators with different priorities

$$x + y \times z = t$$

- Associativity of operators of the same priority

$$x + y - z + t$$

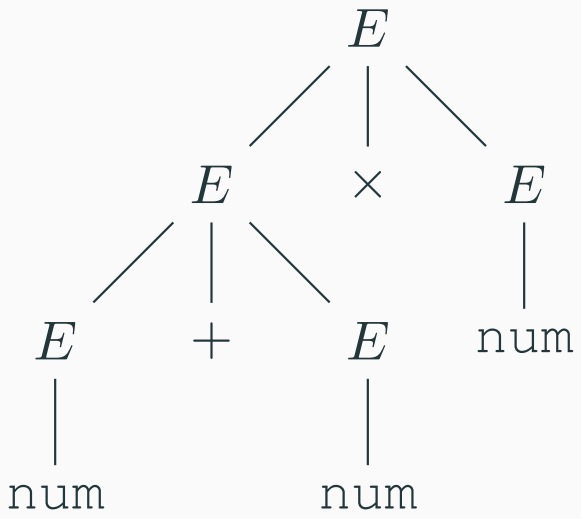
- Dangling `else`

```
if a then if b then s1 else s2
```

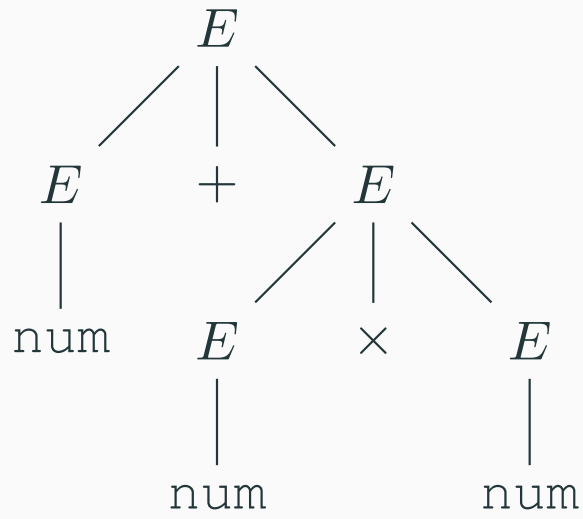
Example Ambiguity (Priority)

Two parse trees for $\texttt{num} + \texttt{num} \times \texttt{num}$

$$E \to E + E$$
$$E \to E \times E$$
$$E \to \texttt{num}$$



priority($+$) > priority($\times$)



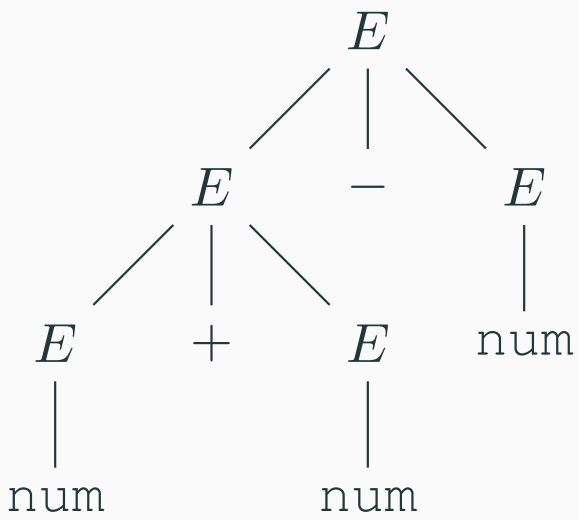
priority($\times$) > priority($+$)

Multiplication should take precedence over addition

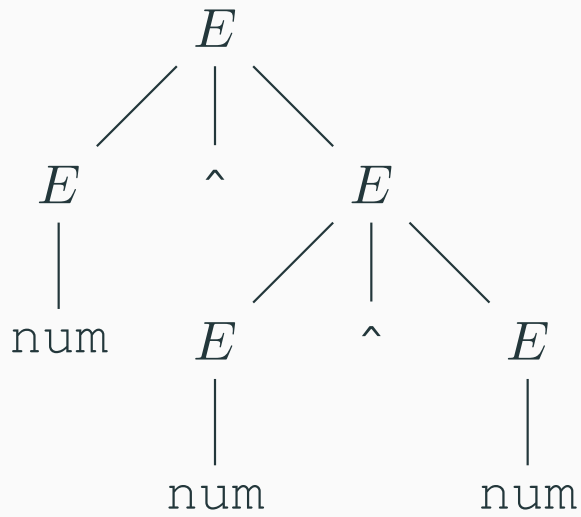
Example Ambiguity (Associativity)

How do operators with same priority associate in a sequence?

$$E \to E + E$$
$$E \to E - E$$
$$E \to E \ \hat{} \ E$$
$$E \to \texttt{num}$$



Left-associative



Right-associative

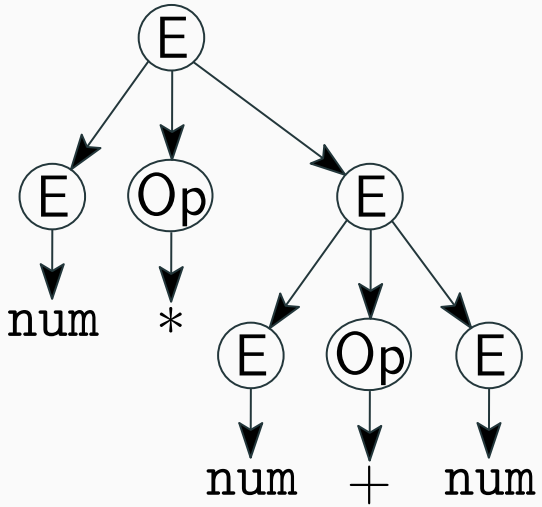
Resolving Ambiguity

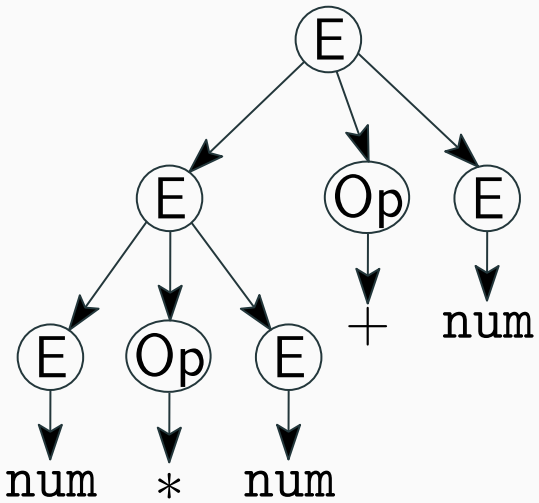
Example

- This grammar is ambiguous

$$\text{E} \rightarrow \text{E Op E} \mid \texttt{num}$$
$$\text{Op} \rightarrow + \mid - \mid * \mid /$$

- Two parse trees for `num * num + num`





`num * (num + num)` `(num * num) + num`

Resolving Ambiguity

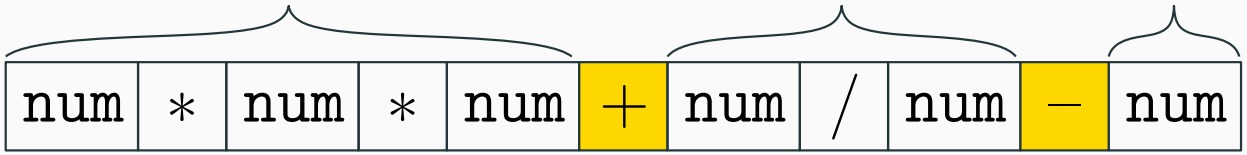
Example

- This grammar is ambiguous

$$\mathsf{E} \rightarrow \mathsf{E}\ \mathsf{Op}\ \mathsf{E} \mid \texttt{num}$$
$$\mathsf{Op} \rightarrow + \mid - \mid * \mid /$$

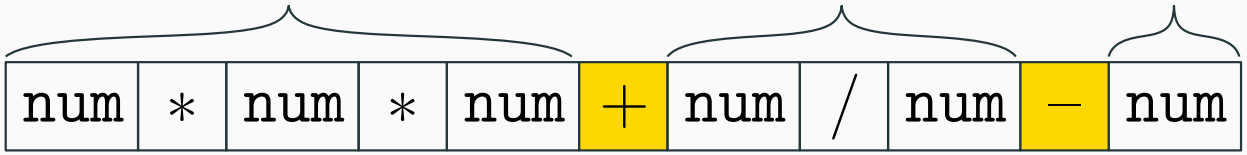
- Grammar does not consider operator precedence
- We can eliminate ambiguity by rewriting it to a new grammar

Resolving Ambiguity: Rewriting



- **Intuition:** since * and / bind more tightly than + and -, think of an expression as a series of "blocks" of terms multiplied and divided together joined by +s and -s

Resolving Ambiguity: Rewriting



Force a construction order where

- First decide how many "blocks" will be of terms joined by + and -
- Then expand those blocks by filling in the integers multiplied and divided together
- A possible grammar:

$$S \to T \mid S + T \mid S - T$$
$$T \to \texttt{num} \mid T * \texttt{num} \mid T/\texttt{num}$$

- Grammar is left recursive: makes operators left associative

Left Factoring

Question:

- Is the following grammar ambiguous? If yes how can we fix it?

$$S \to \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{if}\ E\ \texttt{then}\ S \mid \cdots$$
$$E \to \cdots$$

Left Factoring

Question:

- Is the following grammar ambiguous? If yes how can we fix it?

$$S \rightarrow \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{if}\ E\ \texttt{then}\ S \mid \cdots$$
$$E \rightarrow \cdots$$

Possible Solution:

- On expanding S we cannot choose between productions when the next token is `if`
- We can solve this problem by factoring out the common parts
- This is called left-factoring

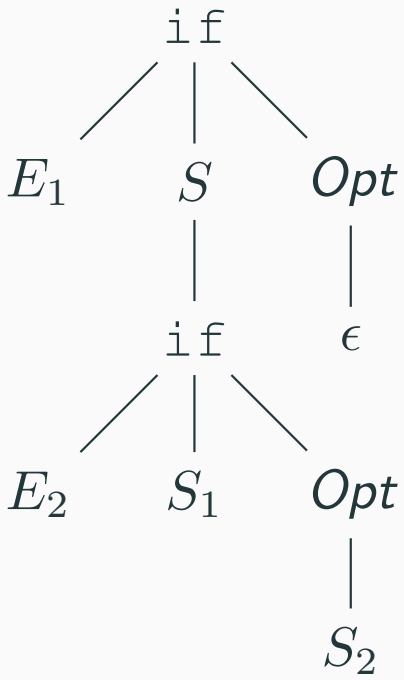
$$S \rightarrow \texttt{if}\ E\ \texttt{then}\ S\ \mathit{Opt}$$
$$\mathit{Opt} \rightarrow \texttt{else}\ S \mid \epsilon$$

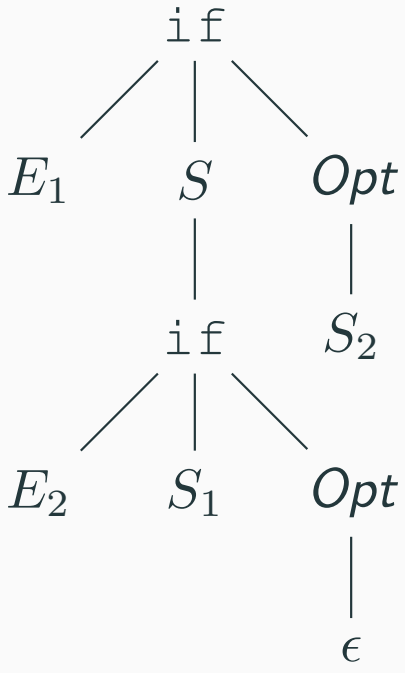
- Is the grammar still ambiguous?

Ambiguous `if-then-else` Grammar

$$S \to \texttt{if}\ E\ \texttt{then}\ S\ \mathit{Opt}$$
$$\mathit{Opt} \to \texttt{else}\ S \mid \epsilon$$

- Which `if` is the `else` attached to?





$$\texttt{if}\ E_1\ \texttt{then}\ \texttt{if}\ E_2\ \texttt{then}\ S_1\ \texttt{else}\ S_2$$

Grammar for Closest-`if` Rule

- Want to rule out `if` E `then` $\boxed{\texttt{if } E \texttt{ then } S}$ `else` S
- Impose that unmatched `if` statements occur only in the `else` clauses

$$
\begin{aligned}
S &\rightarrow \textit{Matched} \mid \textit{Unmatched} \\
\textit{Matched} &\rightarrow \texttt{if } E \texttt{ then } \textit{Matched} \texttt{ else } \textit{Matched} \\
\textit{Unmatched} &\rightarrow \texttt{if } E \texttt{ then } S \\
&\mid \quad \texttt{if } E \texttt{ then } \textit{Matched} \texttt{ else } \textit{Unmatched}
\end{aligned}
$$

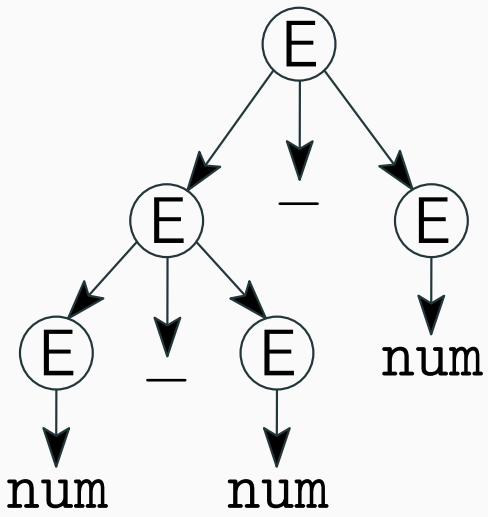
Resolving Ambiguity: Precedence & Associativity

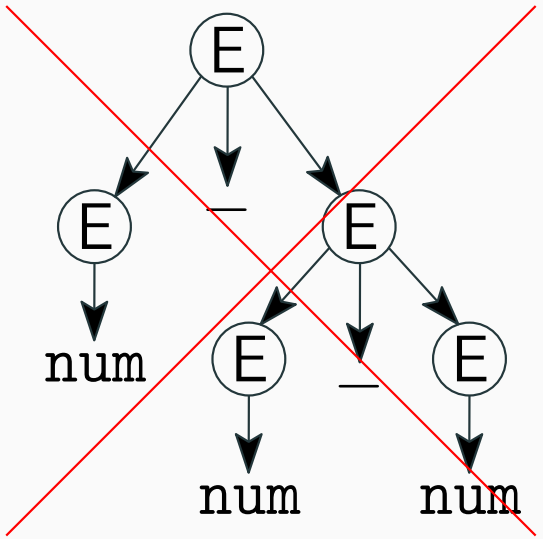
- Instead of rewriting the grammar,
  use the more natural ambiguous grammars
- Use disambiguating declarations to disambiguate grammars
- Most parser generators allow precedence and associativity declarations to disambiguate grammars

Associativity Declarations

- Consider ambiguous grammar:

$$E \to E - E \mid \texttt{num}$$



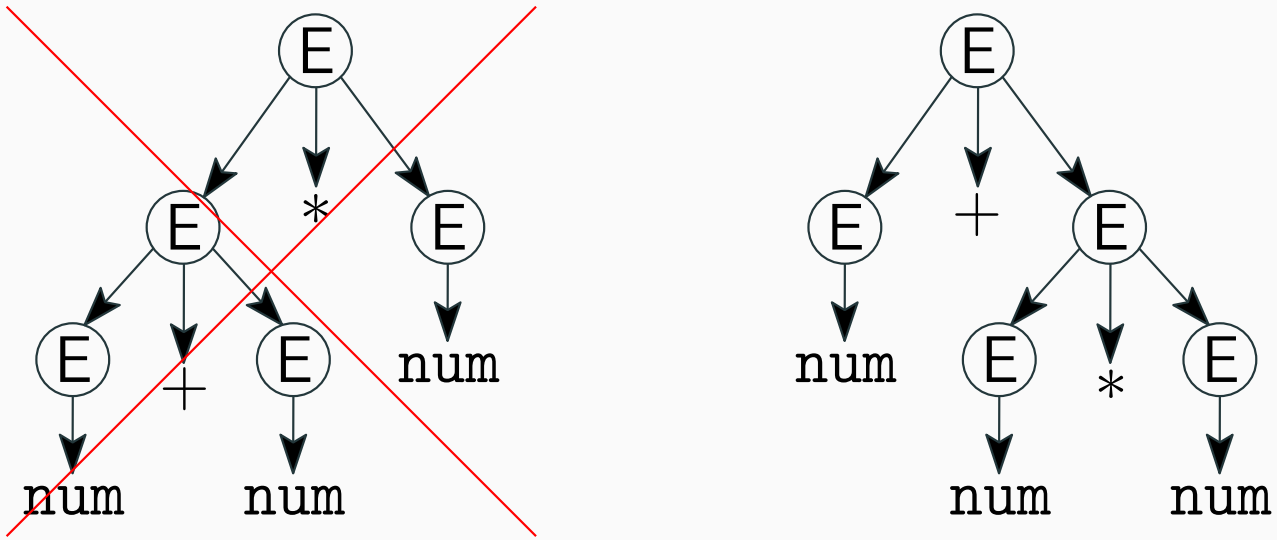


- Left associativity declaration:
  %left –

Precedence Declarations

- Consider ambiguous grammar:

$$E \to E + E \mid E * E \mid \texttt{num}$$

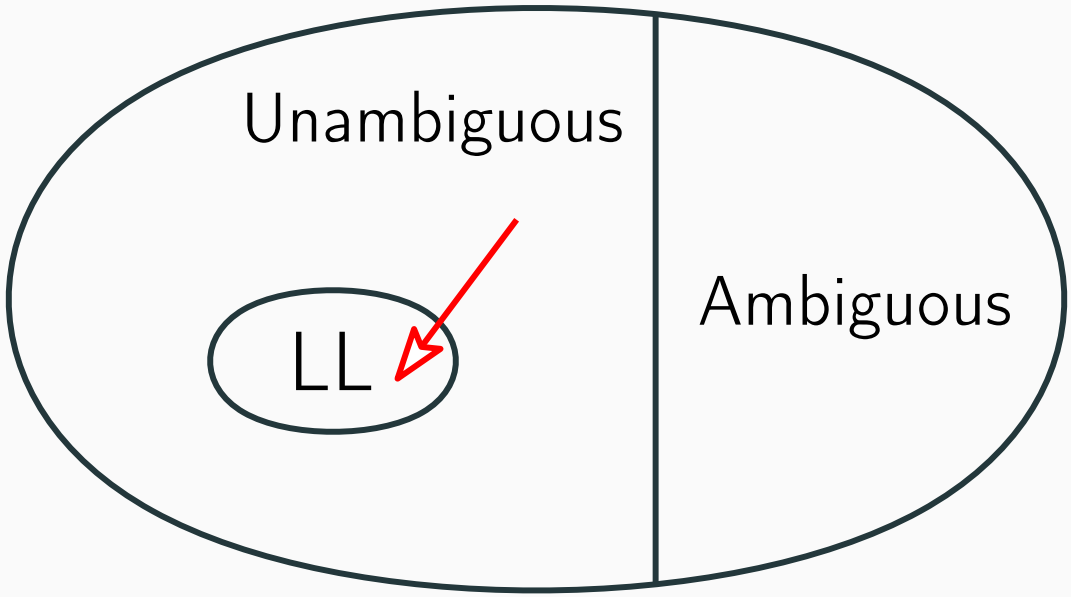


- Precedence declarations (order of precedence is low to high):

  %left $+$

  %left $*$

Create Equivalent LL Grammar



Remove Left Recursion

- Left recursion poses problems for LL parsers (e.g. $S \to Sa$)
- If a non-terminal can expand to a string with itself on the left, parser may expand that non-terminal forever without actually parsing anything
- It is possible to eliminate left recursion by transformation to right recursion
- For a left-recursive pair of rules:

$$A \to A\alpha \mid \beta$$

- Replace with the following rules:

$$A \to \beta A'$$
$$A' \to \alpha A' | \epsilon$$

Exercise

Question

Eliminate left recursion from the following grammar

$$S \to T \mid S + T \mid S - T$$

$$T \to \mathtt{num} \mid T * \mathtt{num} \mid T / \mathtt{num}$$

non-LL Grammar

- Consider the grammar:

$$S \to E + E \mid E$$
$$E \to \texttt{num} \mid (E)$$

- and the two derivations

$$S \Rightarrow E \qquad \Rightarrow (E) \qquad \Rightarrow (\mathsf{num})$$
$$S \Rightarrow E + E \Rightarrow (E) + E \Rightarrow (\mathsf{num}) + E \Rightarrow (\mathsf{num}) + \mathsf{num}$$

- **Question.** Can we decide between

$$S \Rightarrow E$$
$$S \Rightarrow E + E$$

  as the first derivation step based on finite number of lookahead tokens?
- **Answer.** No. Grammar is not LL(k) for any number of k

Making a Grammar LL

- **Problem:** can't decide which S production to apply until we see symbol after first expression
- **Left-factoring:** Factor common prefix E, add new non-terminal E' for what follows that prefix

$$S \to E + E \mid E$$
$$E \to \texttt{num} \mid (E)$$

$$S \to EE'$$
$$E' \to +E \mid \epsilon$$
$$E \to \texttt{num} \mid (E)$$

Making a Grammar LL

- An LL grammar does not have left recursion
- Conversion to LL:

1) First step: remove left recursion from grammar

$$A \rightarrow A\alpha$$
$$\mid \beta$$

2) Second step: left factor the grammar

$$A \rightarrow \alpha\ \beta_1$$
$$\mid \alpha\ \beta_2$$

- This procedure does not necessarily convert any CFG to LL

Exercise

Question:

Left factor the following grammar:

$$
\begin{aligned}
A &\to XA \\
&\mid \quad XB \\
&\mid \quad X \\
&\mid \quad Y \\
&\mid \quad Z
\end{aligned}
$$