CSCI 742 - Compiler Construction

Lecture 13

Grammar Transformations

Instructor: Hossein Hojjat

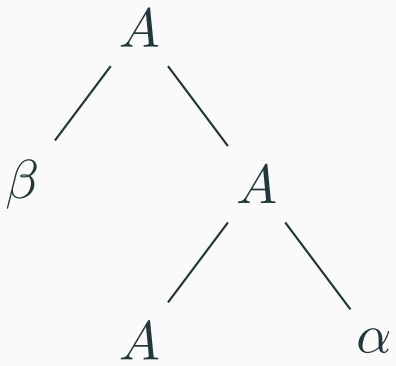
February 22, 2017

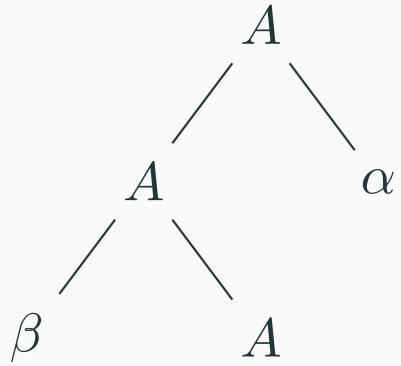
Recap: Chomsky Normal Form (CNF)

- **Chomsky Normal Form (CNF)**: a special format for context-free grammars
- Any context-free language can be expressed by a CNF grammar
- Chomsky normal forms has both theoretical and practical significance
  - CYK is a polynomial parsing algorithm what works best for CNF
  - Parse tree is always a finite binary tree with maximum depth $|w|$ for a word w

Deciding Ambiguity

- Ambiguous grammar: a word has more than one parse tree
- There is no algorithm to decide if a grammar is ambiguous
- 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 CNF conversion
- Occasionally more natural grammar is the ambiguous one
- Parser generators allow disambiguating declarations for ambiguous grammars

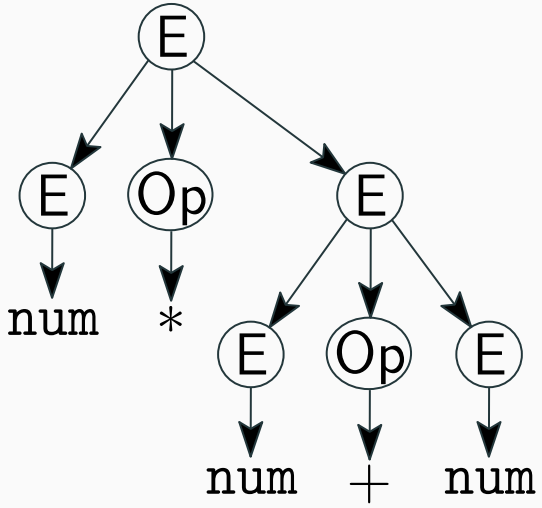
Resolving Ambiguity

Example

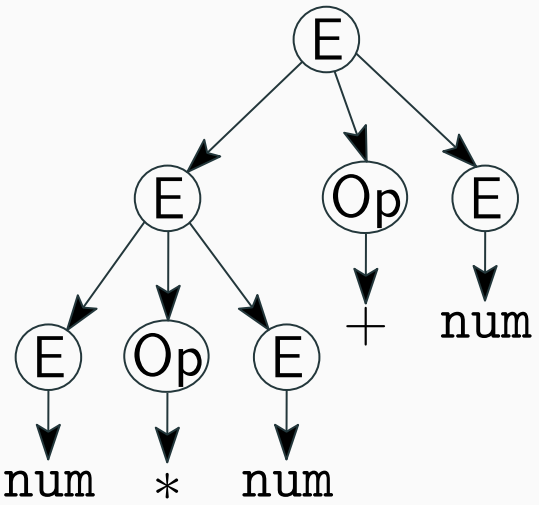
- This grammar is ambiguous

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

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



`num * (num + num)`



`(num * num) + num`

Resolving Ambiguity

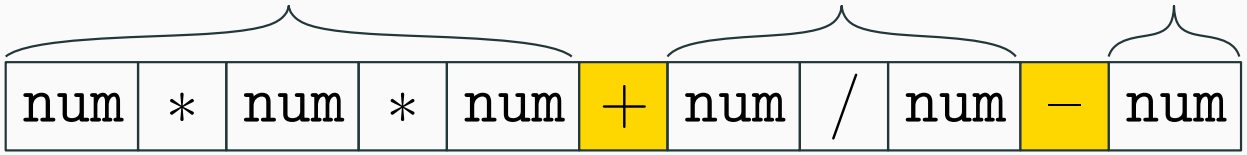
Example

- This grammar is ambiguous

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

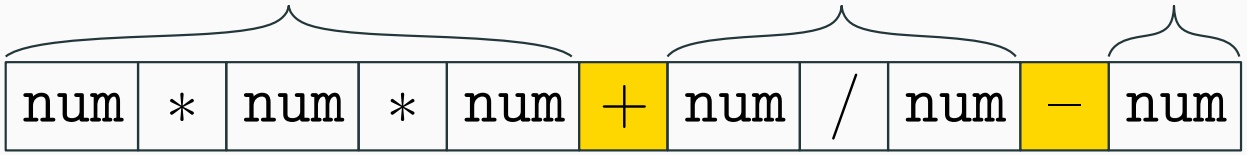
- Grammar does not consider operator precedence
- We can eliminate ambiguity by rewriting it to a new grammar

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

Remove Left Recursion

- Left recursion often poses problems for parsers
- 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

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

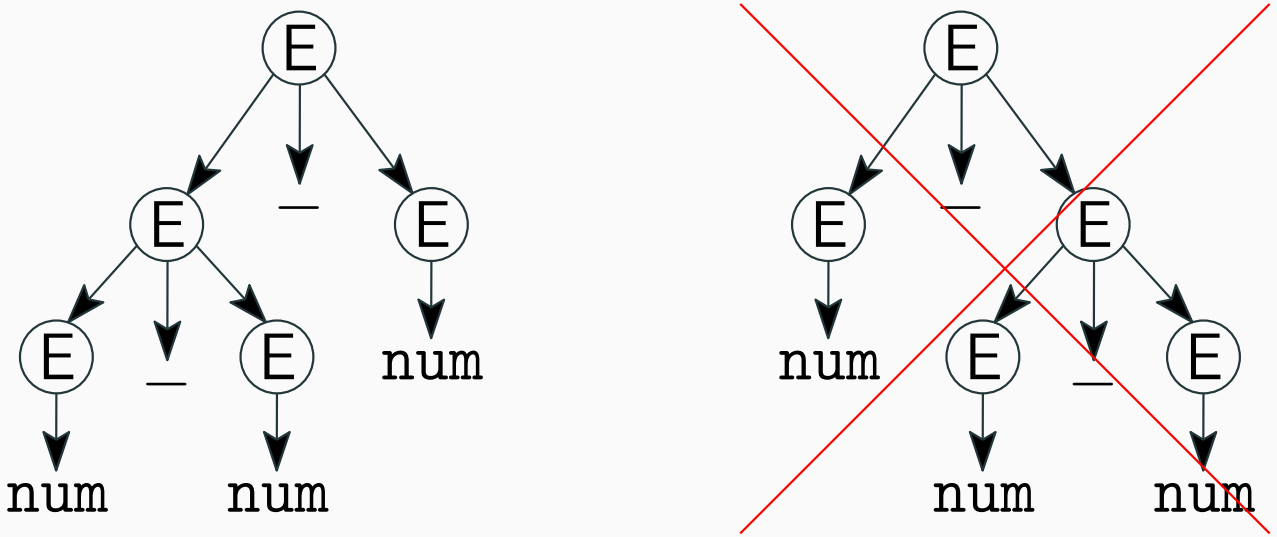
Resolving Ambiguity: Precedence & Associativity

- 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:

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

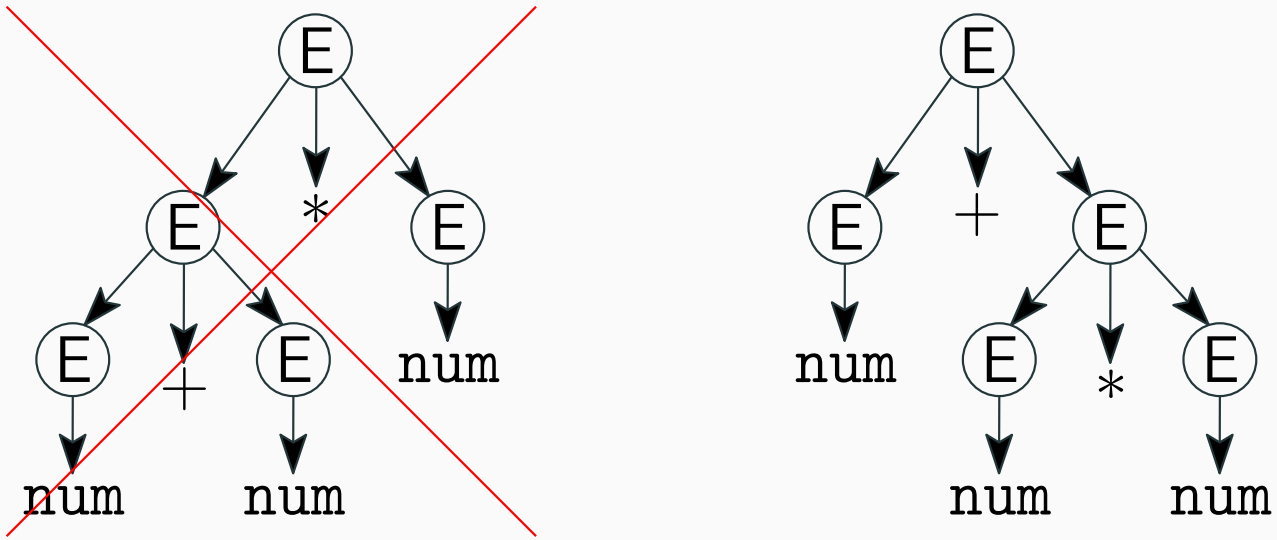


- Left associativity declaration: %left –

Precedence Declarations

- Consider ambiguous grammar:

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



- Precedence declarations:

  %left $+$

  %left $*$

Implementing Declarations in CYK

Precedence Declarations

- when multiple productions compete for being a child in the parse tree, select the one with **least** precedence

Left Associativity

- when multiple productions compete for being a child in the parse tree, select the one with **largest** left subtree

Left Factoring

Question:

- Is the following grammar ambiguous?
  $S \rightarrow$ if E then S else $S \mid$ if E then S

Answer:

- 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

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

Exercise

Question:

Left factor the following grammar:

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