CS 160 Compilers

Lecture 9: More about Parsing

Yu Feng

Fall 2021

CFGs in detail

- A CFG consists of:
  - A set of terminals T
  - A set of non-terminals N
  - A start symbol S (non-terminal)
  - A set of productions: $X \rightarrow Y_1 Y_2 \ldots Y_n$

where $X \in N$ and $Y_i \in (T \cup N \cup \{\varepsilon\})$

CFGs example

- Recall the earlier fragment of Patina:

$$
\begin{aligned}
\mathit{EXPR} \rightarrow\ & \textbf{if}\ \mathit{EXPR}\ \textbf{then}\ \mathit{EXPR}\ \textbf{else}\ \mathit{EXPR} \\
\mid\ & \mathit{EXPR} + \mathit{EXPR} \\
\mid\ & \mathit{ID}
\end{aligned}
$$

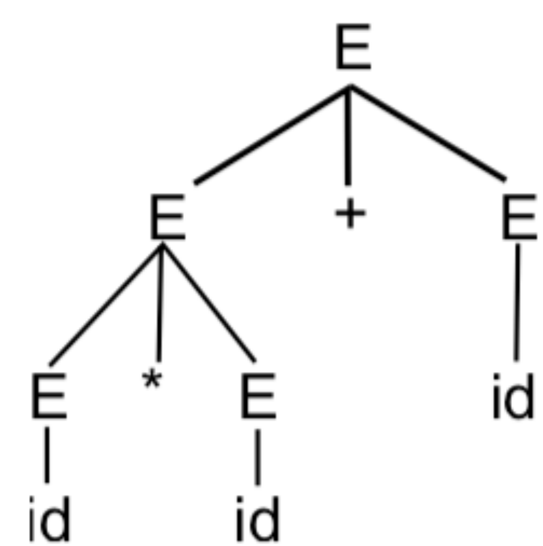
- Some strings in this language:

ID
IF ID THEN ID ELSE ID
ID + ID
IF ID THEN ID+ID ELSE ID

From derivations to parse trees

- A derivation is a sequence of productions: $\boldsymbol{S} \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots$
- A derivation can be drawn as a tree
  - Start symbol is the tree's root
- For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X

```
   E
→ E+E
→ E*E+E
→ id*E+E
→ id*id + E
→ id*id + id
```



Left-most and right-most derivations

- The example we looked at is a left-most derivation
- This means: At each step, we replace the left-most non-terminal
- There is also a similar notion of right-most derivation

Derivations and parse trees

- Observe that left-most and right-most derivations have the same parse tree
- The only difference is the order in which branches are added
- But when parsing tokens, we only care about the final parse tree, which may have many different derivations
- Left-most and right-most derivations are important in parser implementations

Ambiguity

- Consider this grammar:

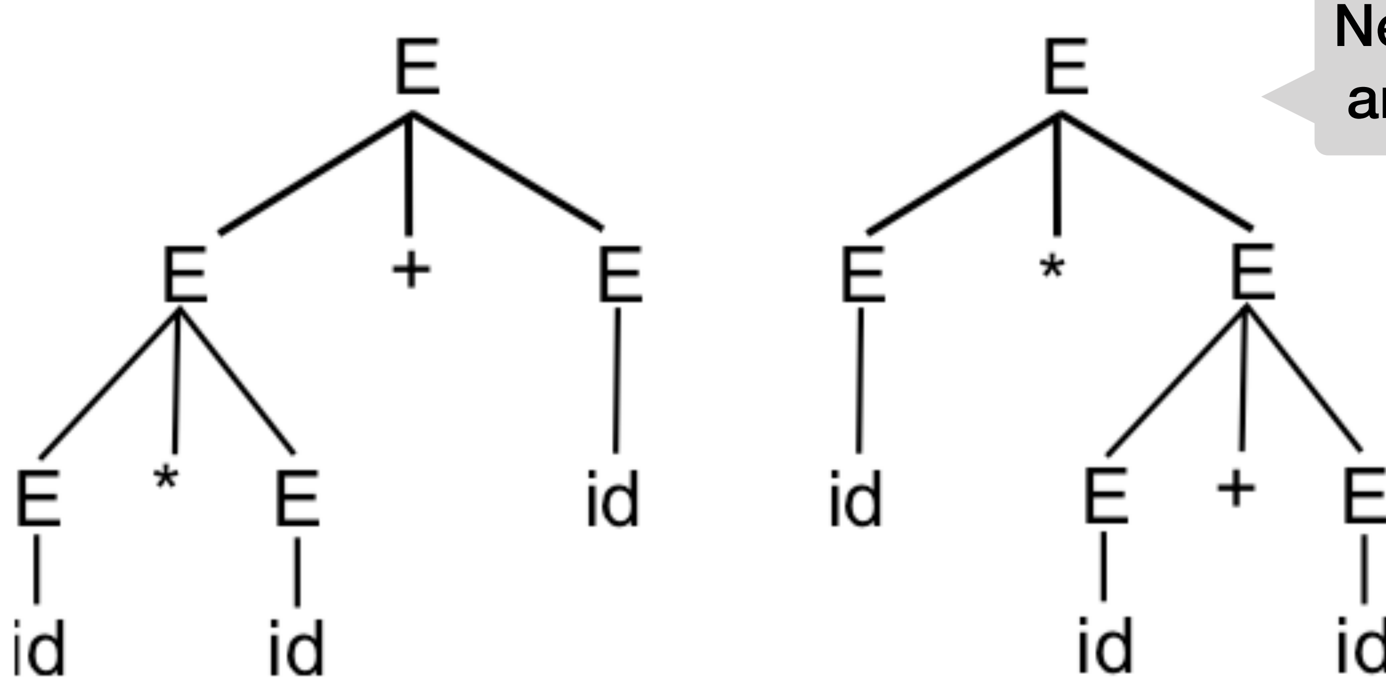
$EXPR \rightarrow E * E$

$| \ E+E \ | \ (E)$

$| \ id$

- Now, this string *id*id+id* has two parse trees!

Need Precedence and Associativity



Ambiguity

- A grammar is ambiguous if it has more than one parse tree for some string
- Equivalently: There is more than one left-most or right-most derivation for some string
- Ambiguity is bad!
- Leaves meaning of programs ill-defined

Dealing with ambiguity

- First method: Rewrite grammar unambiguously
- Question: How can we write simple arithmetic expressions unambiguously?
- Solution: Enforce precedence of times over plus by generating all pluses first

$$S \longrightarrow E + E \mid E$$
$$E \longrightarrow id * E \mid id \mid E * id \mid (E)$$

Dealing with ambiguity

- However, converting grammars to unambiguous form can be very difficult
- It also often results in horrible, unintuitive grammars with many non-terminals
- It is also fundamentally impossible to transform an ambiguous grammar into a unambiguous grammar
- For this reason, tools such as bison include disambiguation mechanisms

https://www.gnu.org/software/bison/

Precedence and Associativity

- Instead of rewriting the grammar:
  - Use the more natural ambiguous grammar
  - Along with disambiguating declarations
- The parser tool bison allows you to declare precedence and associativity for this

Associativity Declarations

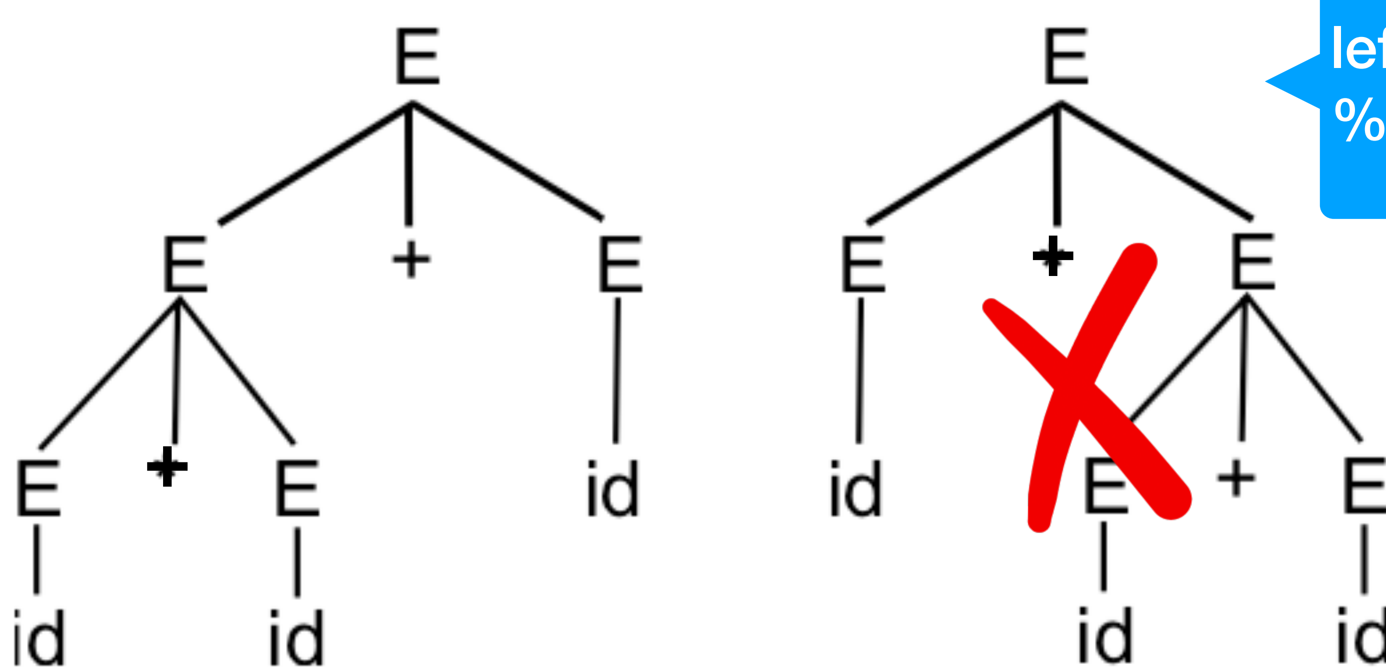
- Consider this grammar:

$EXPR \rightarrow E * E$

$|\ E{+}E\ |\ (E)$

$|\ id$

- Now, this string *id+id+id* has two parse trees!



left associativity of plus:
%left +

Precedence Declarations

- Consider this grammar:

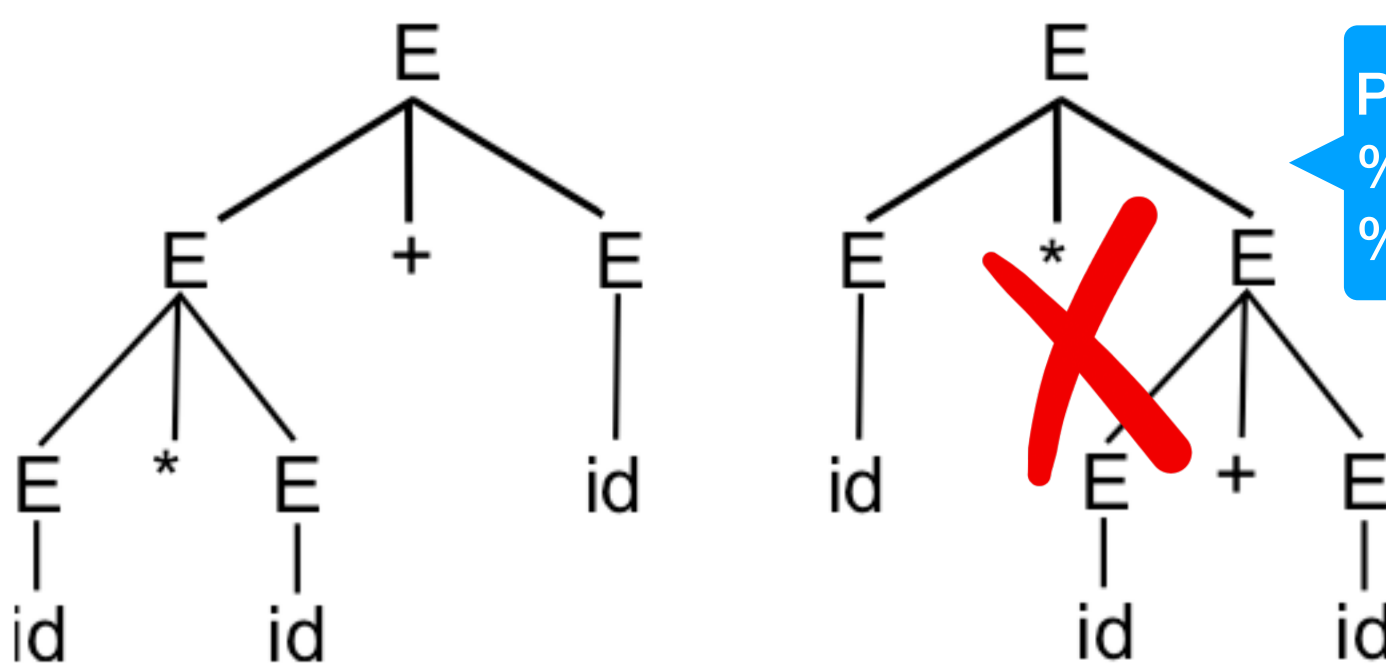
$EXPR \rightarrow E * E$

$| \ E+E \ | \ (E)$

$| \ id$

All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associativity. When two tokens declared in different precedence declarations associate, the one declared ***later*** has the ***higher*** precedence and is grouped first.

- Now, this string $id*id+id$ has two parse trees!



Precedence Declaration:
%left +
%left *

TODOs by next lecture

- Hw2 will be due soon. Please start ASAP!