Recursion and the Problem of Operator Associativity

Recap: Eliminating Left Recursion

- We have seen that LL(k) parsers cannot handle grammars with left recursion
- Solution seen in this course: Eliminate left recursion
- Example

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid a \mid b
\end{aligned}
$$

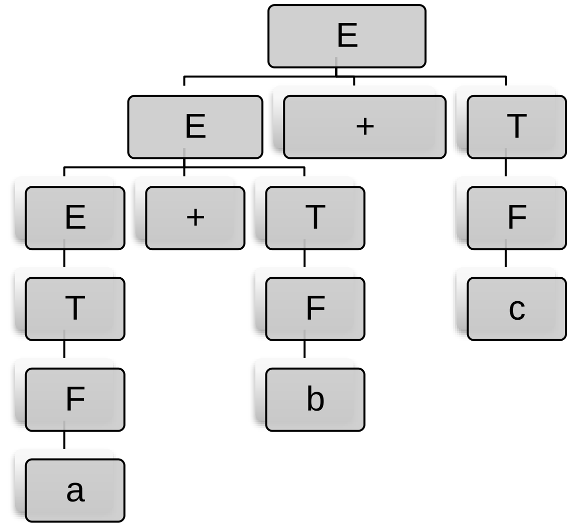
can be transformed to

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \varepsilon \\
T &\rightarrow FT' \\
T' &\rightarrow * FT' \mid \varepsilon \\
F &\rightarrow (E) \mid a \mid b
\end{aligned}
$$

Syntax Tree for Transformed Grammar

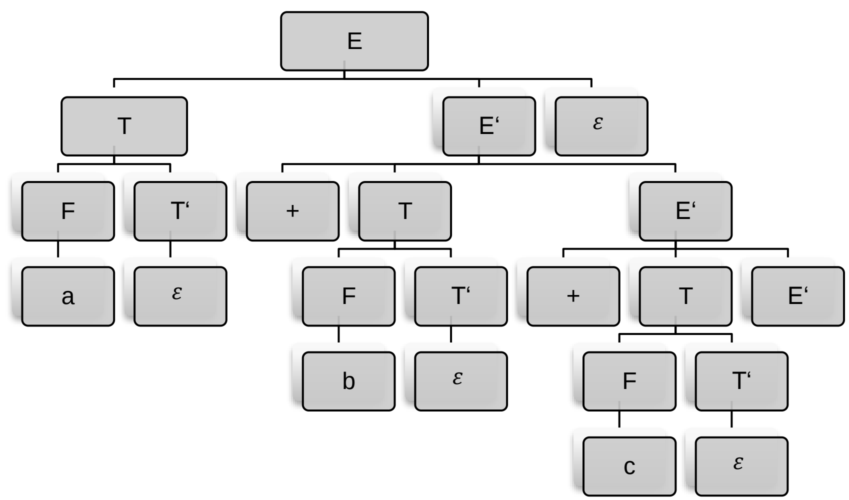
- As mentioned, the transformation of the grammar will preserve the generated language but *not the syntax tree*
- Example: $a + b + c$
- Syntax tree with left-recursive grammar

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid a \mid b \mid c$$



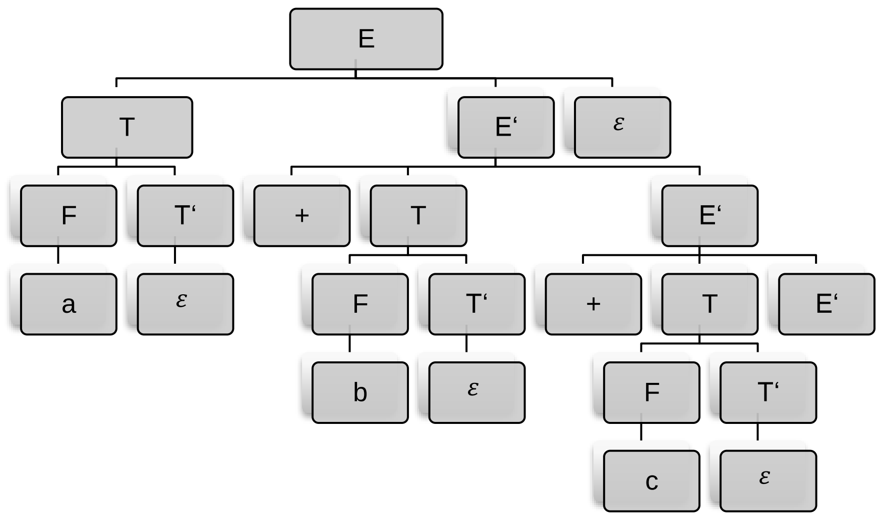
- Syntax tree with transformed grammar

$$E \rightarrow TE'$$
$$E' \rightarrow \ +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid a \mid b$$



Problem of Operator Associativity

- Not so nice: The shape of the syntax tree of the transformed grammar seems to suggest that the + operator is right-associative, i.e. $a + b + c$ is interpreted as $a + (b + c)$
  - That’s not what we want…



- This is a consequence of the elimination of the left recursion

Possible Solutions

- Different solutions possible to represent left associativity in the syntax tree:
  1. Use a parser that doesn't have problems with recursive grammars, for example LR-parser
     - LR-parsers are annoying to implement by hand. Requires parser generator tool
  2. Use a top-down parser with right-to-left parsing instead of LL(k) left-to-right parsing
     - Works for this case, but then we will have the same problem for operators for which we *want* right associativity
  3. After finishing parsing, transform the syntax tree from right to left associativity
  4. Fix the syntax tree during parsing
     - See the next slide

Fix the syntax tree during parsing

- Let's look at these three rules:

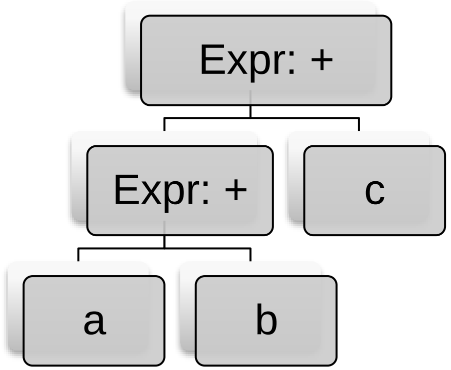
$$E \rightarrow TE'$$
$$E' \rightarrow \ +TE' \mid \varepsilon$$

- If we use the Kleene-star notation, we can see better what is happening in the grammar:

$$E \rightarrow T\ (+T)^*$$

- We can implement the Kleene-star as a loop and build a nice AST:

```
t = parseTerm()
while(lookahead==Plus) {
    t = new Expr(Plus, t, parseTerm())
}
return t
```



- Do the same in parseTerm() for the multiplication operator

Conclusion

- In a recursive-descent parser, you can parse left-associative operators with a loop.
  - Right-associative operators can be still parsed with right recursion.
- This shows the advantage and disadvantage of a hand-written recursive descent parser:
  - Good: You have full control over the parsing process and can generate the AST in the way you want it
  - Bad: You have to implement a loop for each precedence level. For languages with many precedence levels like C or C++, this becomes ugly and slow! To parse a simple expression like “a”, the code is checking the while-loops of all precedence levels
    - Speed can be improved with an algorithm called “precedence climbing” https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm
    - Not a problem for the language in this course ☺