Little Languages

Lecture 13:

Leftmost Grammar Derivation and Expr Trees

Leftmost Derivation Strategy for LL(1)

1. Begin with your start symbol

2. Extend it with its production rule
   - Terminal nodes can consume terminal tokens directly
   - Nonterminal nodes you'll come back to
   - When there is alternation, 0 or more (*), 0 or 1 (?), 1 or more (+)…
     - You'll need to *peek ahead* one token and follow any non-terminals to find the *first terminal* to match the peeked token. Add node for step to *follow*.

3. Once you've completed a production rule, start on step 2 with the leftmost non-terminal node in your diagram. Stop when all non-terminals replaced.

Parse tree practice: ((1))

```
Expr -> Atom
Atom -> '(' Expr ')' | number
```

1 * 2

```
Expr          -> MaybeMulDiv
Atom          -> '(' Expr ')' | number
MaybeMulDiv   -> Atom MulDivOp?
MulDivOp      -> ('*' | '/') Atom
```

```
1 * 2 * 3
```

```
Expr          -> MaybeMulDiv
Atom          -> '(' Expr ')' | number
MaybeMulDiv   -> Atom MulDivOp?
MulDivOp      -> ('*' | '/') Atom
```

1 / 2 / 3

```
Expr          -> MaybeMulDiv
Atom          -> '(' Expr ')' | number
MaybeMulDiv   -> Atom MulDivOp?
MulDivOp      -> ('*' | '/') Atom MulDivOp?
```

PollEv Speed Round

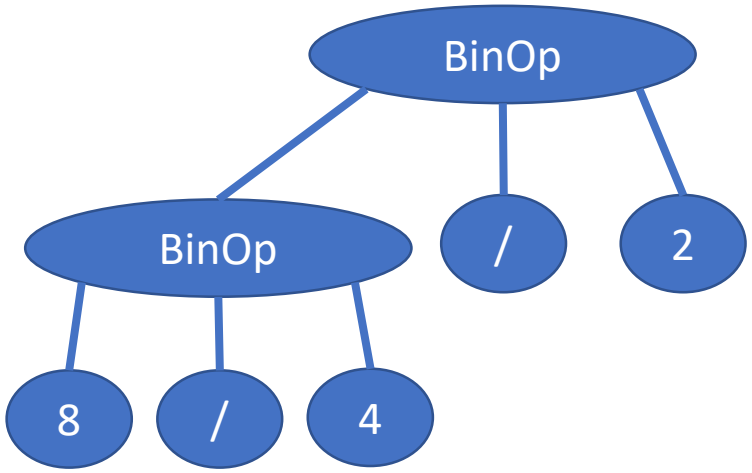
Comparing the expressions A and B

A. (8 / 4) / 2
B. 8 / (4 / 2)

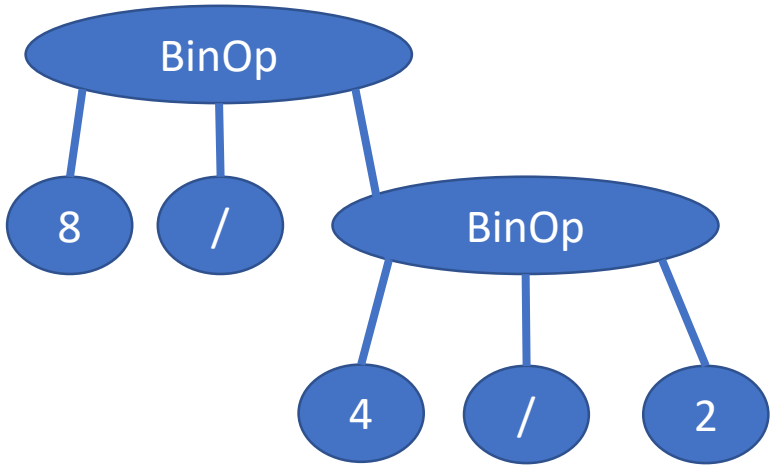
Choose one:
A is **equal to** B
A is **greater than** B
A is **less than** B

PollEv Speed Round

Which of these two Expr trees best conveys how to compute: 8 / 4 / 2



Left

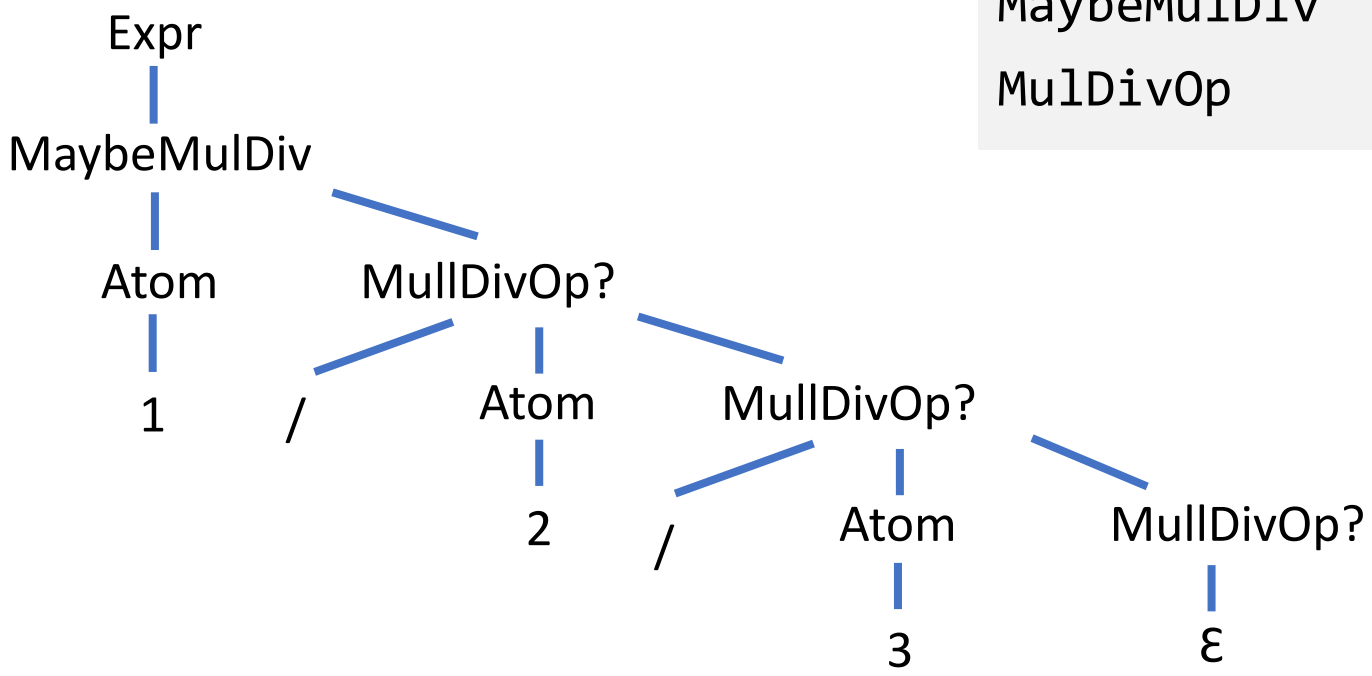


Right

Grammar Derivations vs. Expr Tree Productions

```
Expr          -> MaybeMulDiv
Atom          -> '(' Expr ')' | number
MaybeMulDiv   -> Atom MulDivOp?
MulDivOp      -> ('*' | '/') Atom MulDivOp?
```

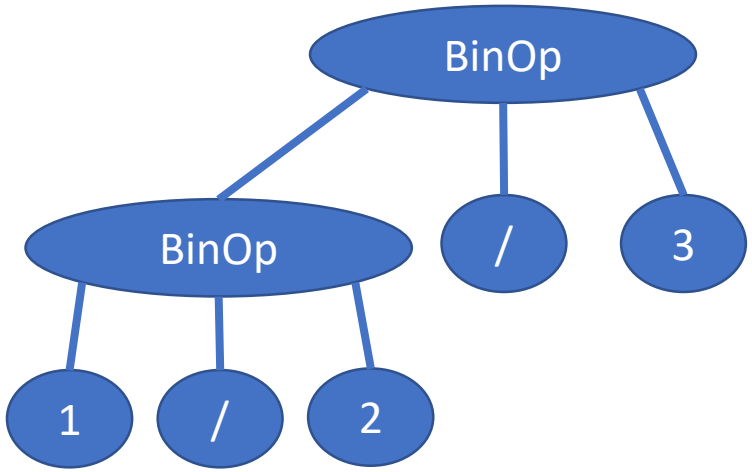
Derivation of: 1 / 2 / 3



Notice the leftmost derivation of this grammar grows deeply on the right-hand side.

However, the desired Expr tree, when operators of the same precedence are encountered one after another, grows deeply on the left-hand side to convey left-to-right evaluation. The key challenge of your Expr parser is producing this Expr tree recursively while your parser processes tokens as shown above.

Desired **Expr** Tree



```
1 + 2 * 3
```

```
Expr         -> MaybeAddSub
Atom         -> '(' Expr ')' | number
MaybeMulDiv -> Atom MulDivOp?
MulDivOp    -> ('*' | '/') Atom MulDivOp?
MaybeAddSub -> MaybeMulDiv AddSubOp?
AddSubOp    -> ('+' | '-') MaybeMulDiv AddSubOp?
```

(1 + 2) * 3

```
Expr          -> MaybeAddSub
Atom          -> '(' Expr ')' | number
MaybeMulDiv -> Atom MulDivOp?
MulDivOp    -> ('*' | '/') Atom MulDivOp?
MaybeAddSub -> MaybeMulDiv AddSubOp?
AddSubOp    -> ('+' | '-') MaybeMulDiv AddSubOp?
```

1 * 2 + 3

```
Expr         -> MaybeAddSub
Atom         -> '(' Expr ')' | number
MaybeMulDiv -> Atom MulDivOp?
MulDivOp    -> ('*' | '/') Atom MulDivOp?
MaybeAddSub -> MaybeMulDiv AddSubOp?
AddSubOp    -> ('+' | '-') MaybeMulDiv AddSubOp?
```