CS251: Introduction to Language Processing

Syntax Analysis

Vishwesh Jatala

Department of CSE

Indian Institute of Technology Bhilai

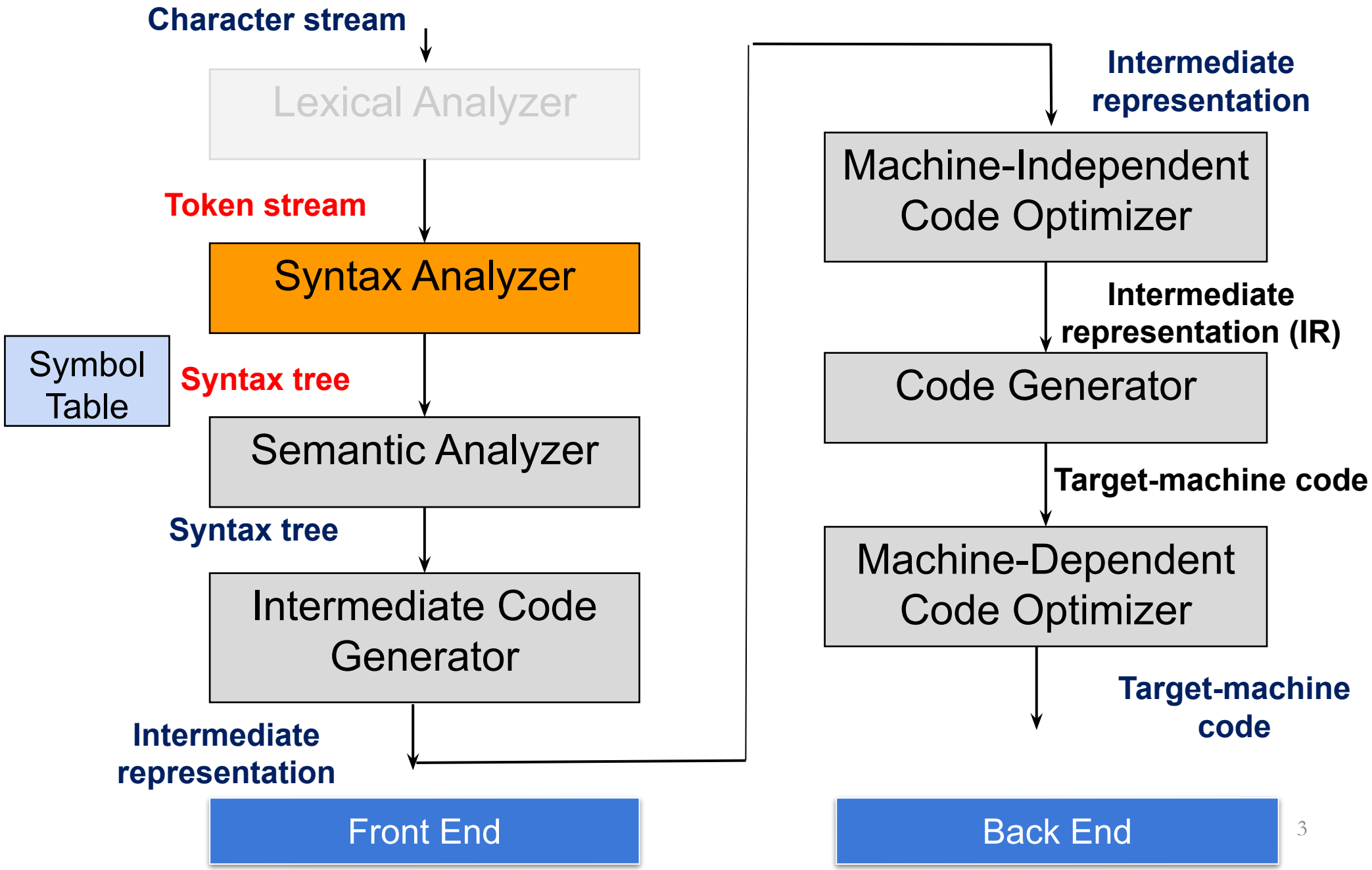
vishwesh@iitbhilai.ac.in

2023-24 M

Acknowledgement

- References for today's slides
  - *Stanford University:*
    - *https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/*
  - *Lecture notes of Prof. Amey Karkare (IIT Kanpur) and Late Prof. Sanjeev K Aggarwal (IIT Kanpur)*

Compiler Design



Derivations

- A leftmost derivation is a derivation in which each step expands the leftmost nonterminal.
- A rightmost derivation is a derivation in which each step expands the rightmost nonterminal.

Leftmost Derivation

$E \rightarrow$ `int` | E Op E | `(`E`)`

Op $\rightarrow$ `+` | `-` | `*` | `/`

E

$\Rightarrow$ E Op E

$\Rightarrow$ `int` Op E

$\Rightarrow$ `int *` E

$\Rightarrow$ `int * (`E`)`

$\Rightarrow$ `int * (`E Op E`)`

$\Rightarrow$ `int * (int` Op E`)`

$\Rightarrow$ `int * (int +` E`)`

$\Rightarrow$ `int * (int + int)`

$\Rightarrow$ `int * (int + int)`

Leftmost and Rightmost Derivations

E
⇒ E Op E
⇒ int Op E
⇒ int * E
⇒ int * (E)
⇒ int * (E Op E)
⇒ int * (int Op E)
⇒ int * (int + E)
⇒ int * (int + int)

Leftmost Derivation

E
⇒ E Op E
⇒ E Op (E)
⇒ E Op (E Op E)
⇒ E Op (E Op int)
⇒ E Op (E + int)
⇒ E Op (int + int)
⇒ E * (int + int)
⇒ int * (int + int)

Rightmost Derivation

Leftmost Derivations

```
BLOCK → STMT
      | { STMTS }

STMTS → ε
      | STMT STMTS
STMT  → EXPR;
      | if (EXPR) BLOCK
      | while (EXPR) BLOCK
      | do BLOCK while (EXPR);
      | BLOCK
      | …

EXPR  → identifier
      | constant
      | EXPR + EXPR
      | EXPR - EXPR
      | EXPR * EXPR
      | EXPR = EXPR
      | …
```

Can you derive **`id = id + constant;`**

⇒ BLOCK
⇒ STMT
⇒ EXPR;
⇒ EXPR = EXPR;
⇒ **`id =`** EXPR;
⇒ **`id =`** EXPR **`+`** EXPR;
⇒ **`id = id +`** EXPR;
⇒ **`id = id + constant;`**

Derivations

- A derivation encodes two pieces of information:
  - What productions were applied produce the resulting string from the start symbol?
  - In what order were they applied?
- Multiple derivations might use the same productions, but apply them in a different order.
- Encoding the derivation steps in a tree leads to parse-tree.

Parse Trees

$\Rightarrow$ `int * (int + int)`

E

Parse Trees

$\Rightarrow$ `int * (int + int)`

E

Parse Trees

$\Rightarrow$ `int * (int + int)`

E

E

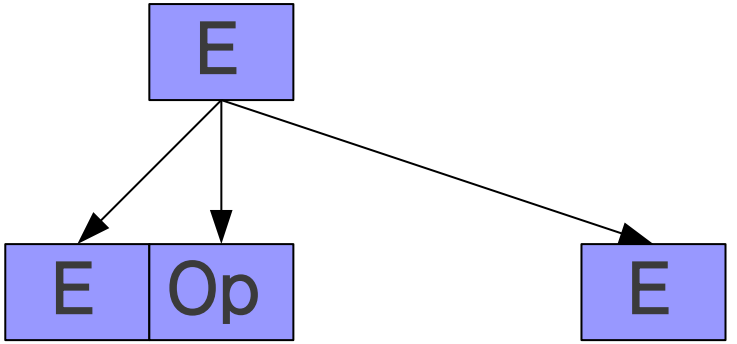
$\Rightarrow$ E Op E

Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E



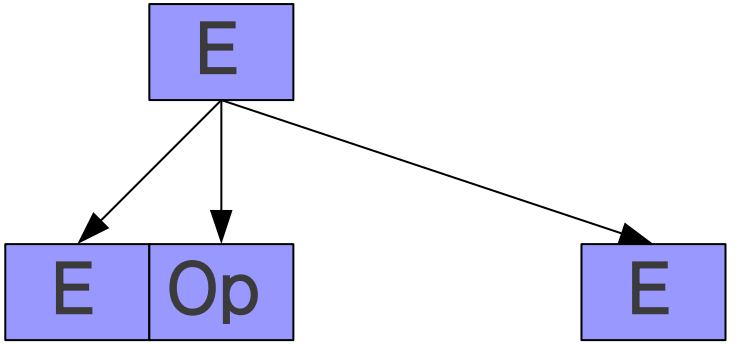
Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ `int` Op E



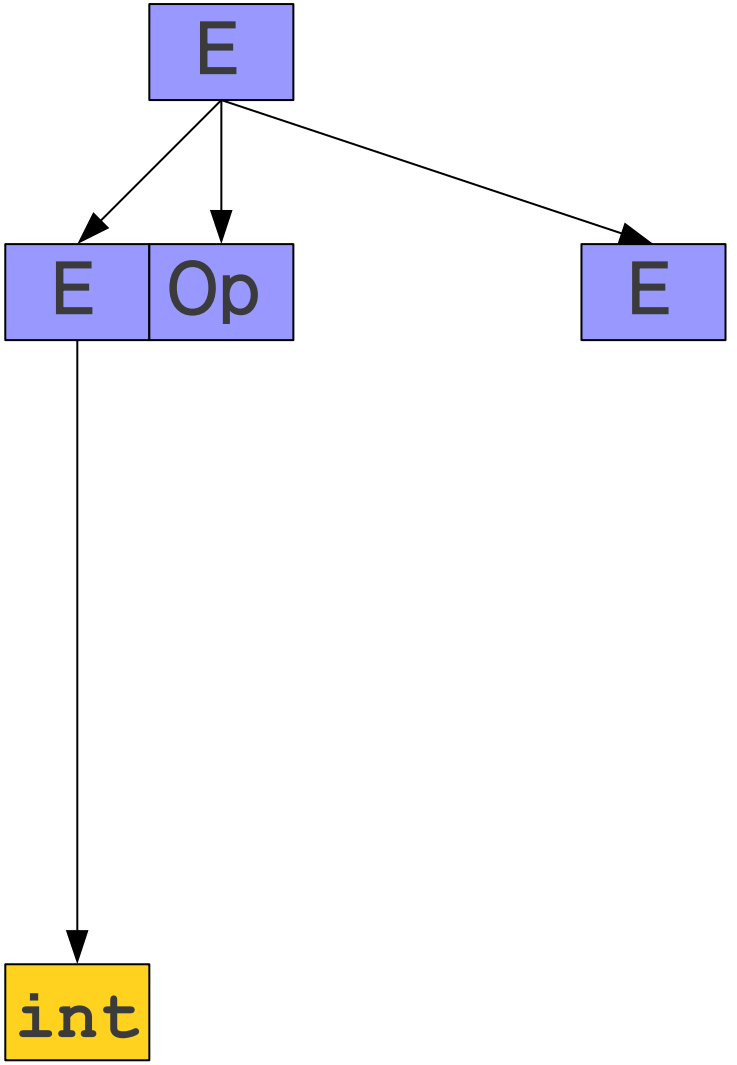
Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ `int` Op E



Parse Trees

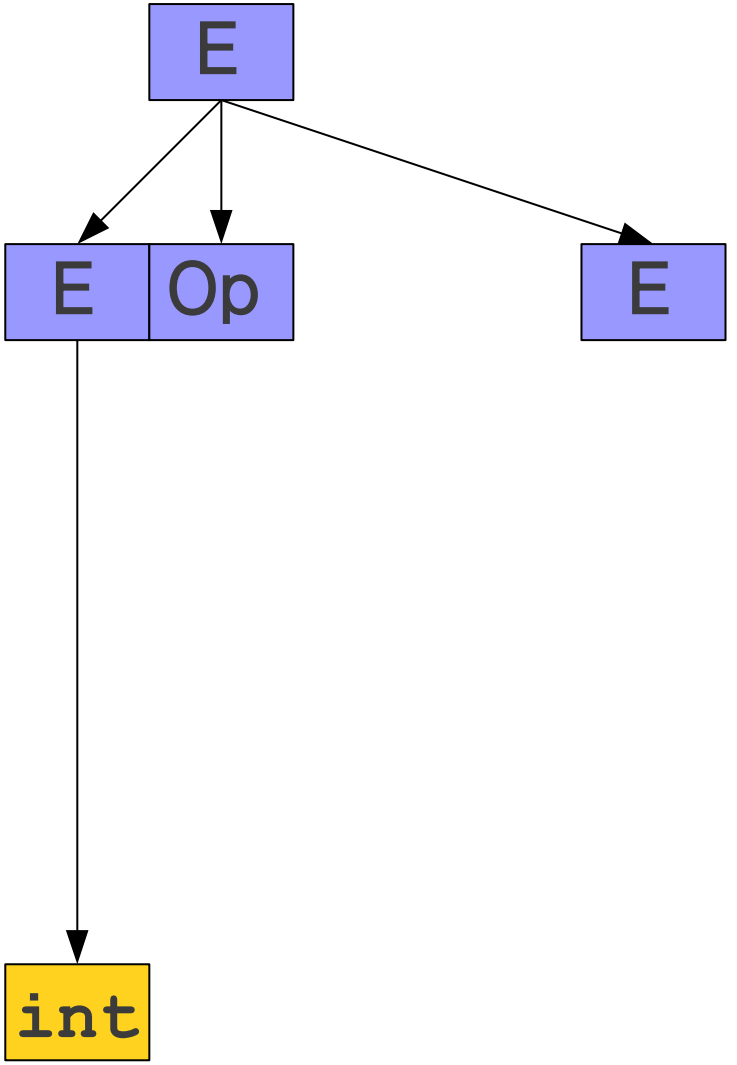
⇒ `int * (int + int)`

E

⇒ E Op E

⇒ `int` Op E

⇒ `int *` E



Parse Trees

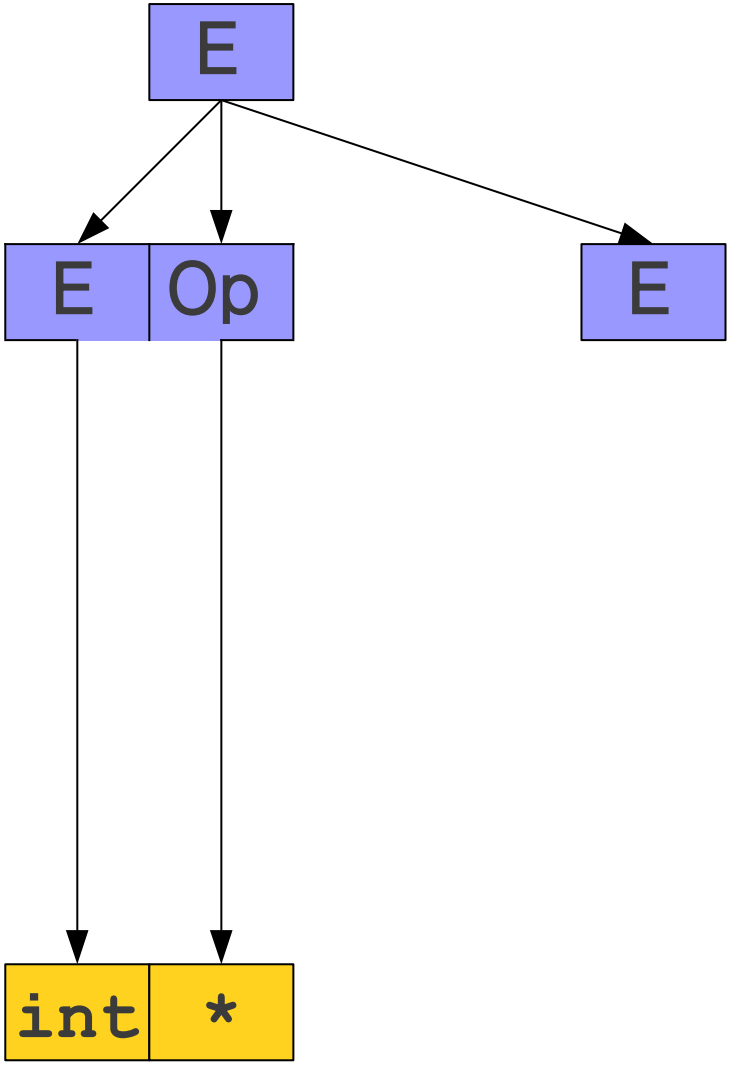
⇒ `int * (int + int)`

E

⇒ E Op E

⇒ `int` Op E

⇒ `int *` E



Parse Trees

⇒ `int * (int + int)`

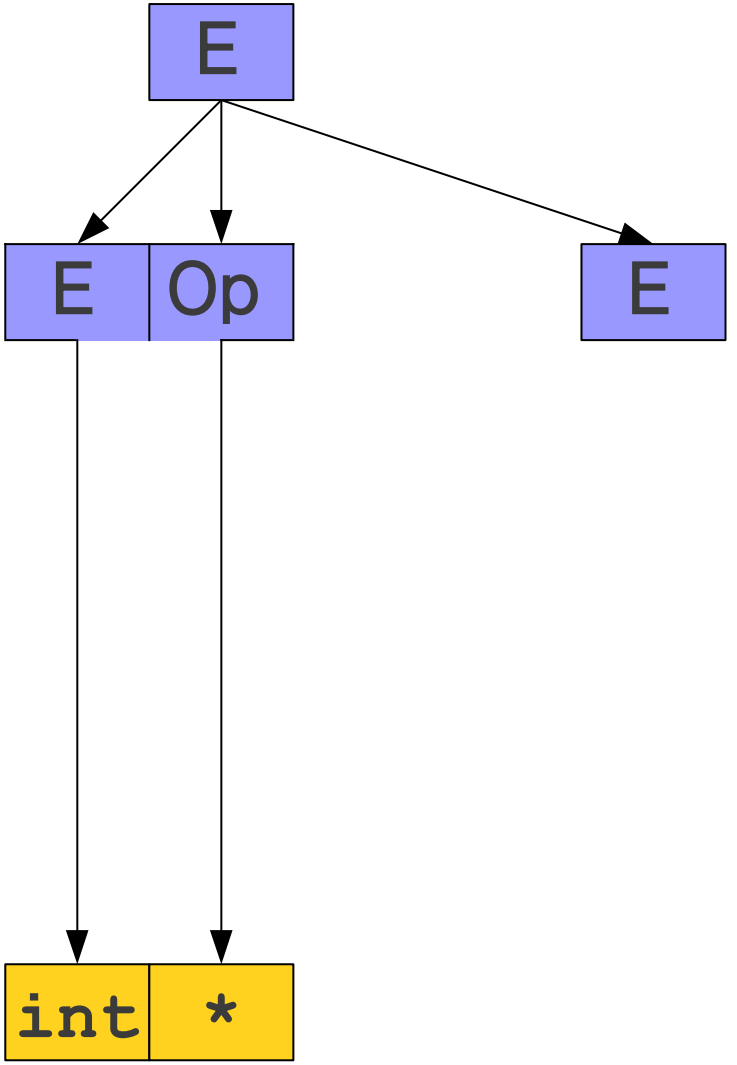
E

⇒ E Op E

⇒ `int` Op E

⇒ `int *` E

⇒ `int * (`E`)`



Parse Trees

⇒ `int * (int + int)`

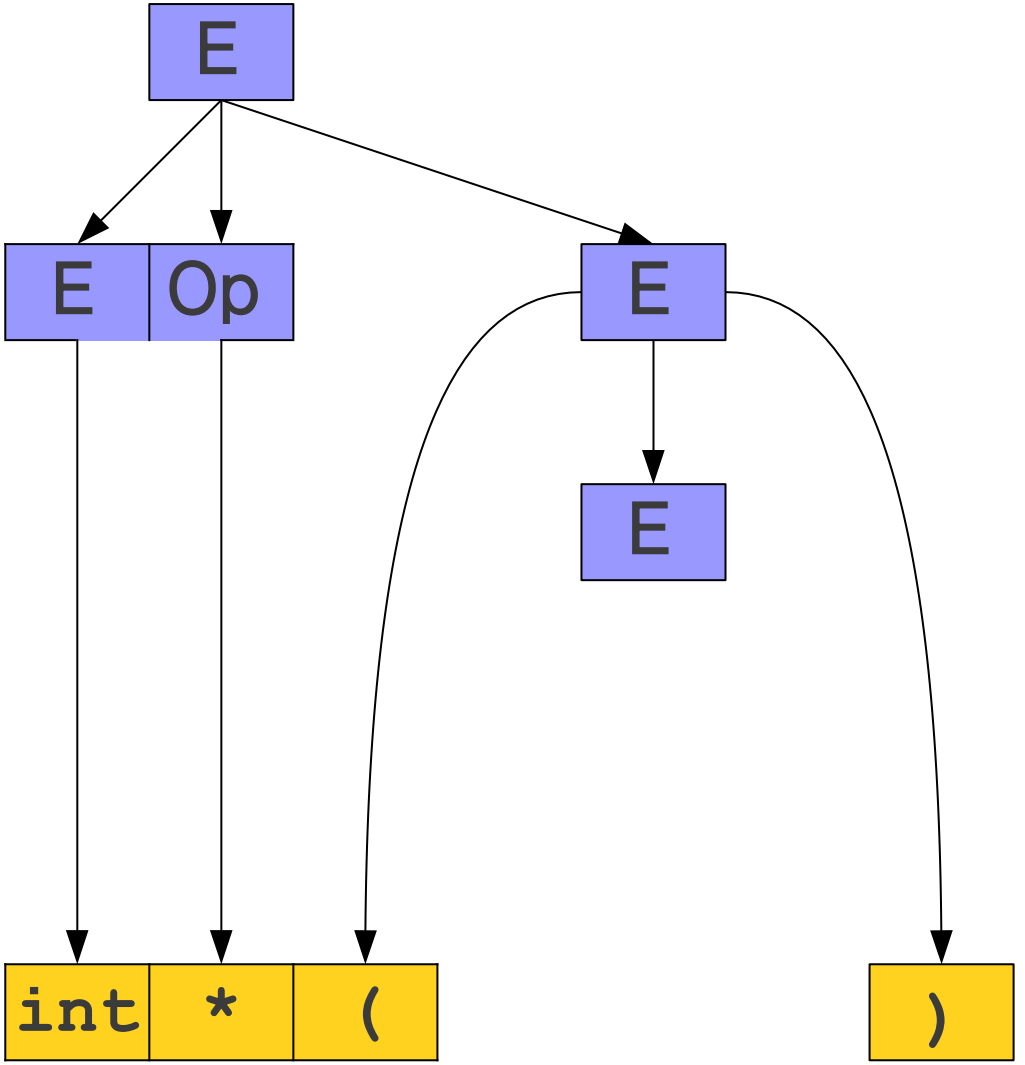
E

⇒ E Op E

⇒ `int` Op E

⇒ `int *` E

⇒ `int * (`E`)`



Parse Trees

⇒ `int * (int + int)`

E

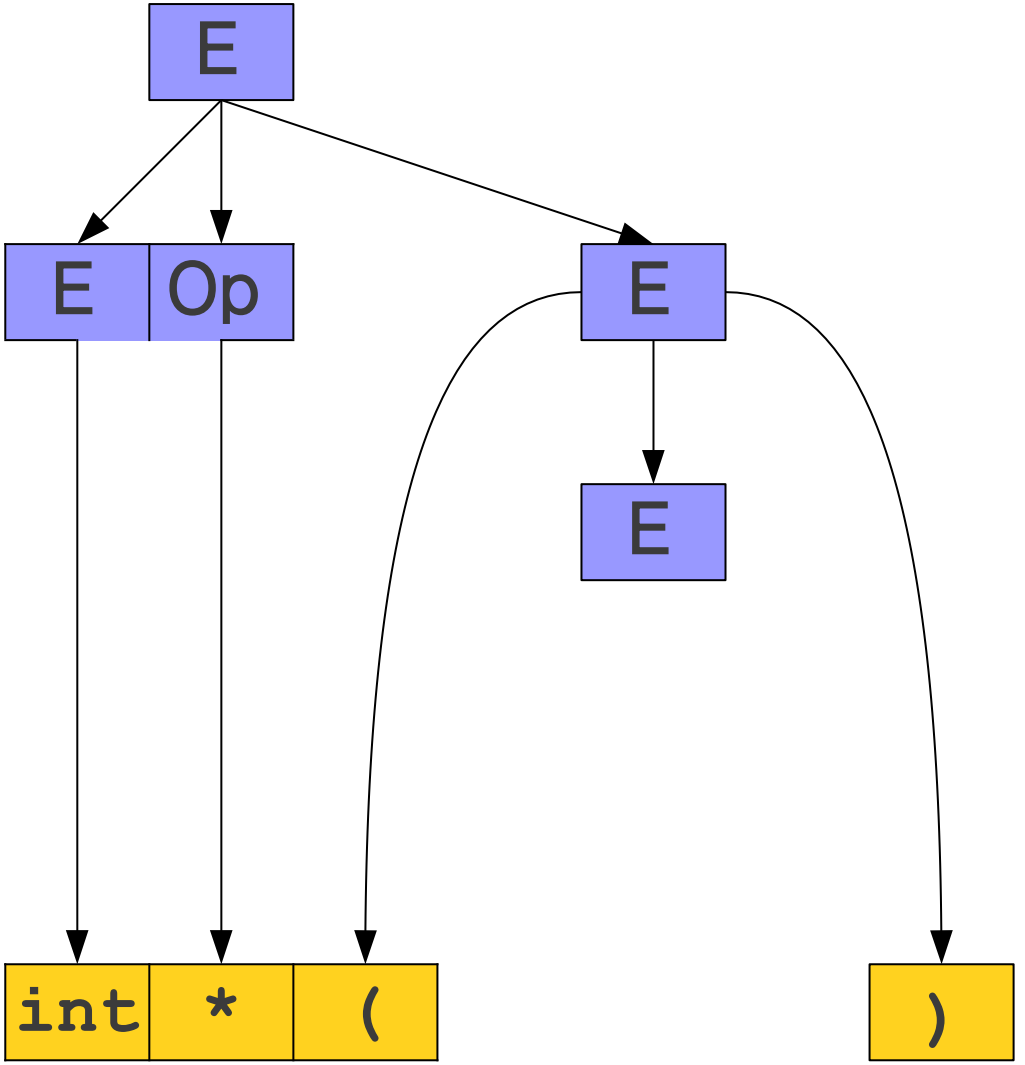
⇒ E Op E

⇒ `int` Op E

⇒ `int *` E

⇒ `int * (`E`)`

⇒ `int * (`E Op E`)`



Parse Trees

⇒ `int * (int + int)`

E

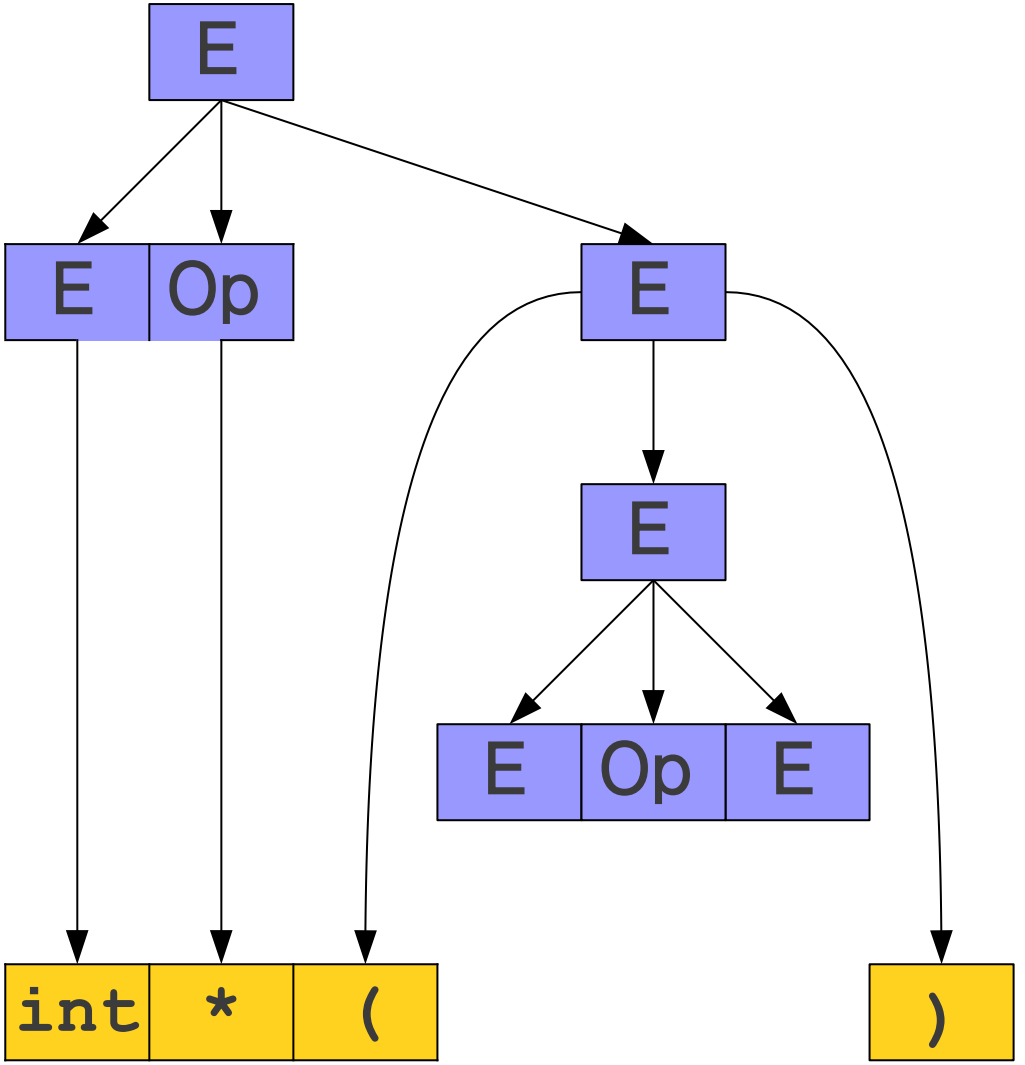
⇒ E Op E

⇒ `int` Op E

⇒ `int *` E

⇒ `int * (`E`)`

⇒ `int * (`E Op E`)`



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

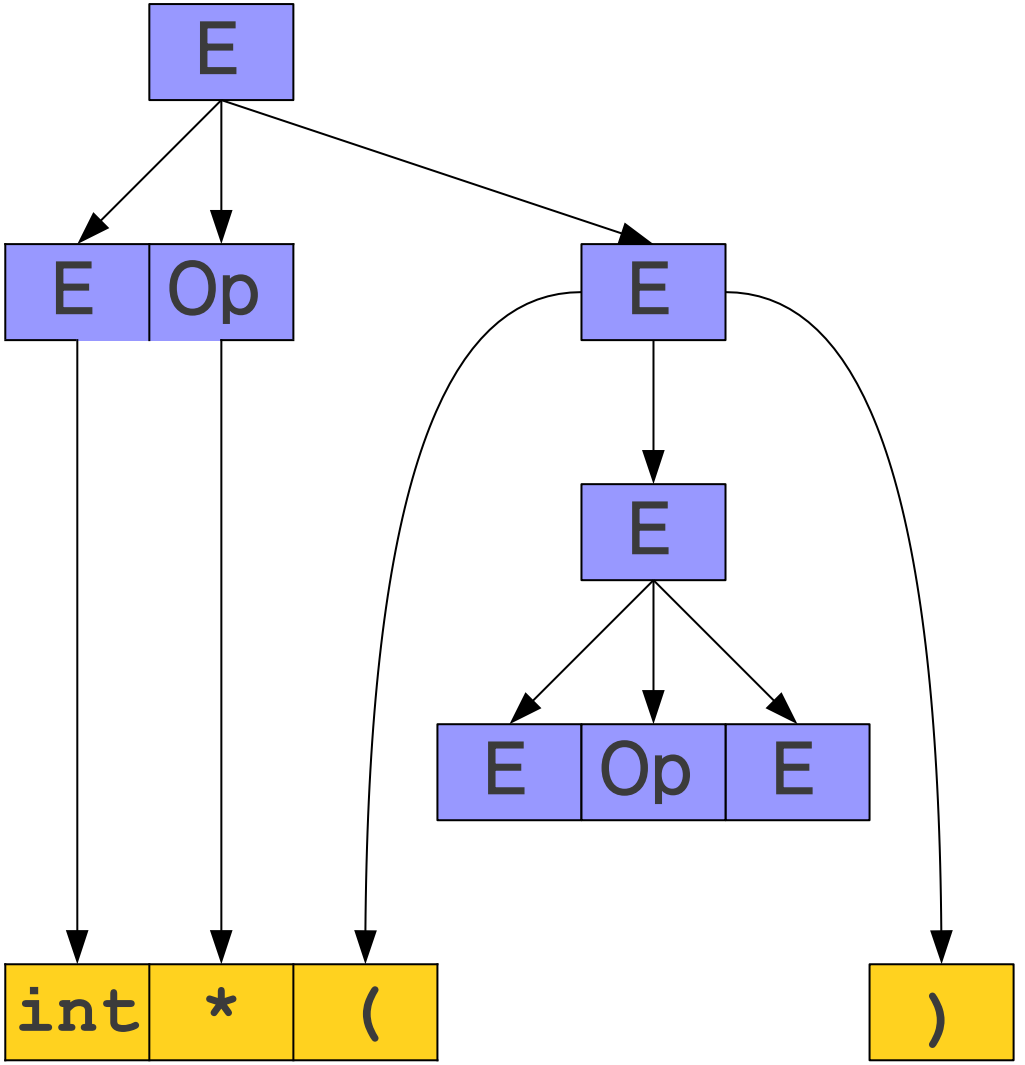
⇒ `int` Op E

⇒ `int *` E

⇒ `int * (`E`)`

⇒ `int * (`E Op E`)`

⇒ `int * (int` Op E`)`



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

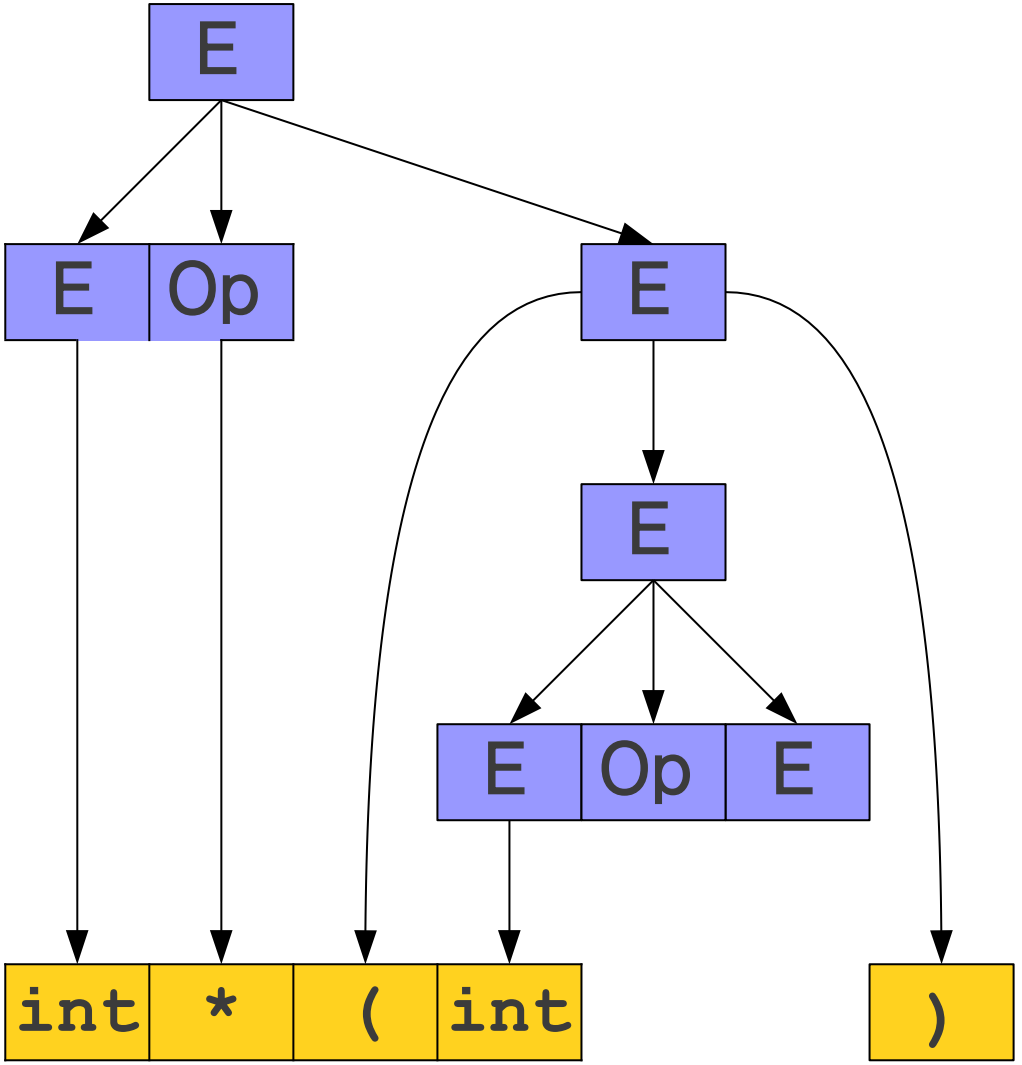
⇒ `int` Op E

⇒ `int *` E

⇒ `int * (`E`)`

⇒ `int * (`E Op E`)`

⇒ `int * (int` Op E`)`



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ `int` Op E

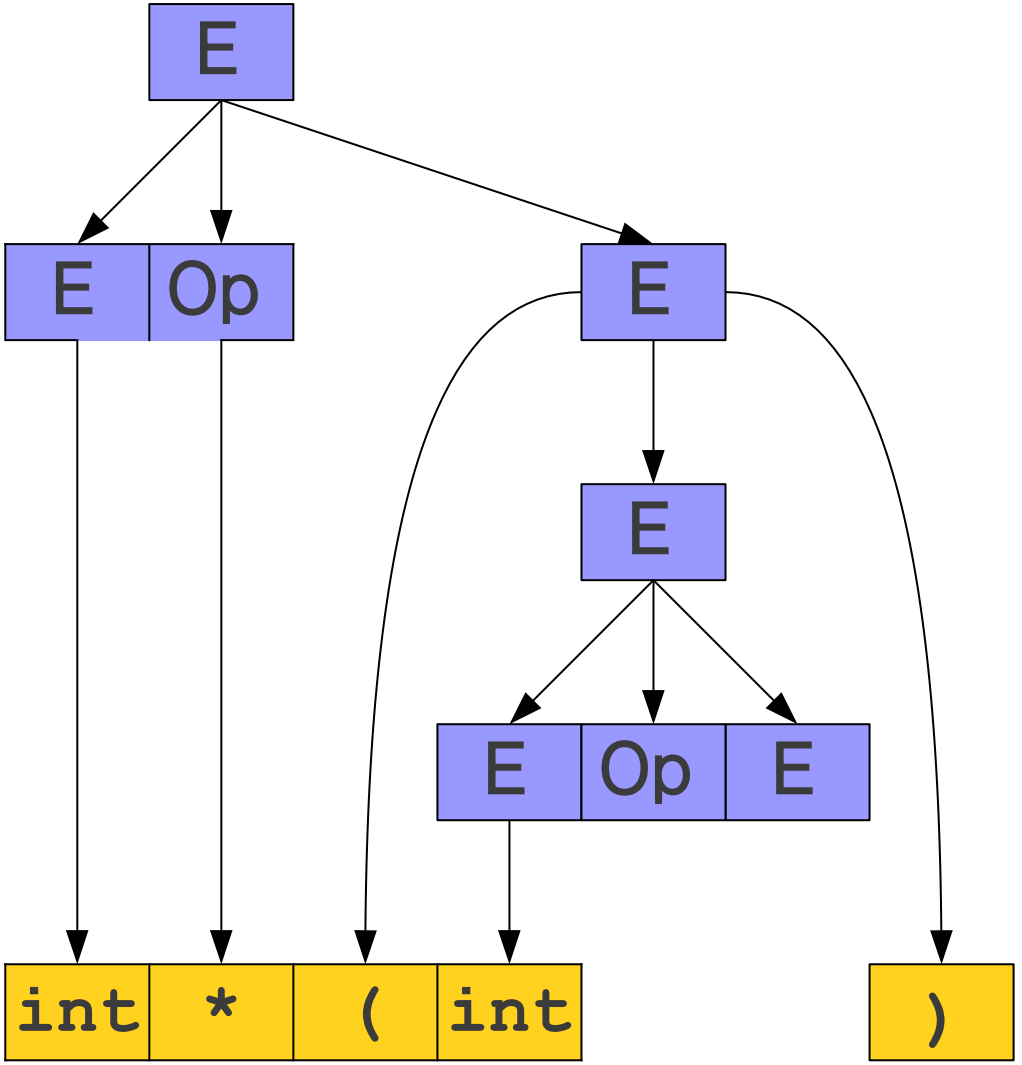
⇒ `int *` E

⇒ `int * (`E`)`

⇒ `int * (`E Op E`)`

⇒ `int * (int` Op E`)`

⇒ `int * (int +` E`)`



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ `int` Op E

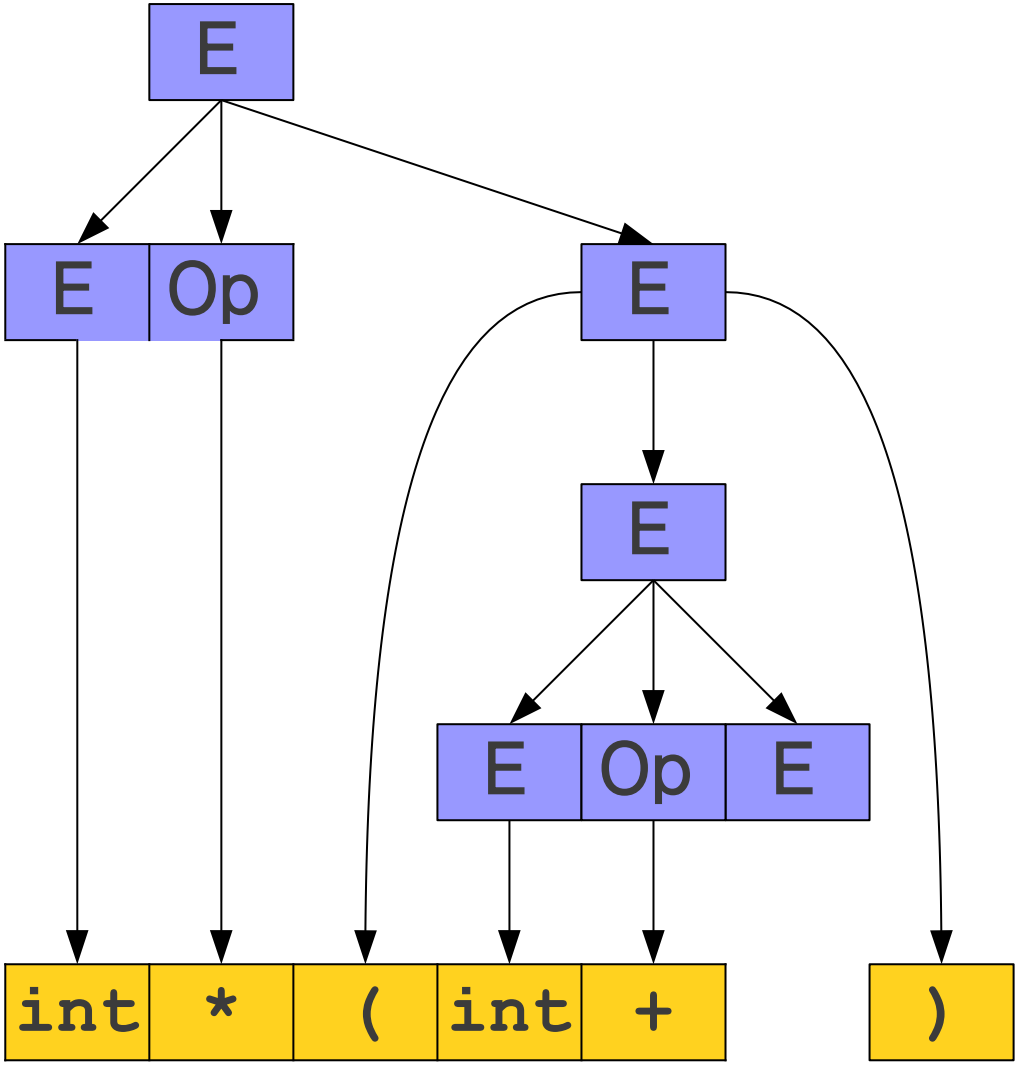
⇒ `int *` E

⇒ `int * (`E`)`

⇒ `int * (`E Op E`)`

⇒ `int * (int` Op E`)`

⇒ `int * (int +` E`)`



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ `int` Op E

⇒ `int *` E

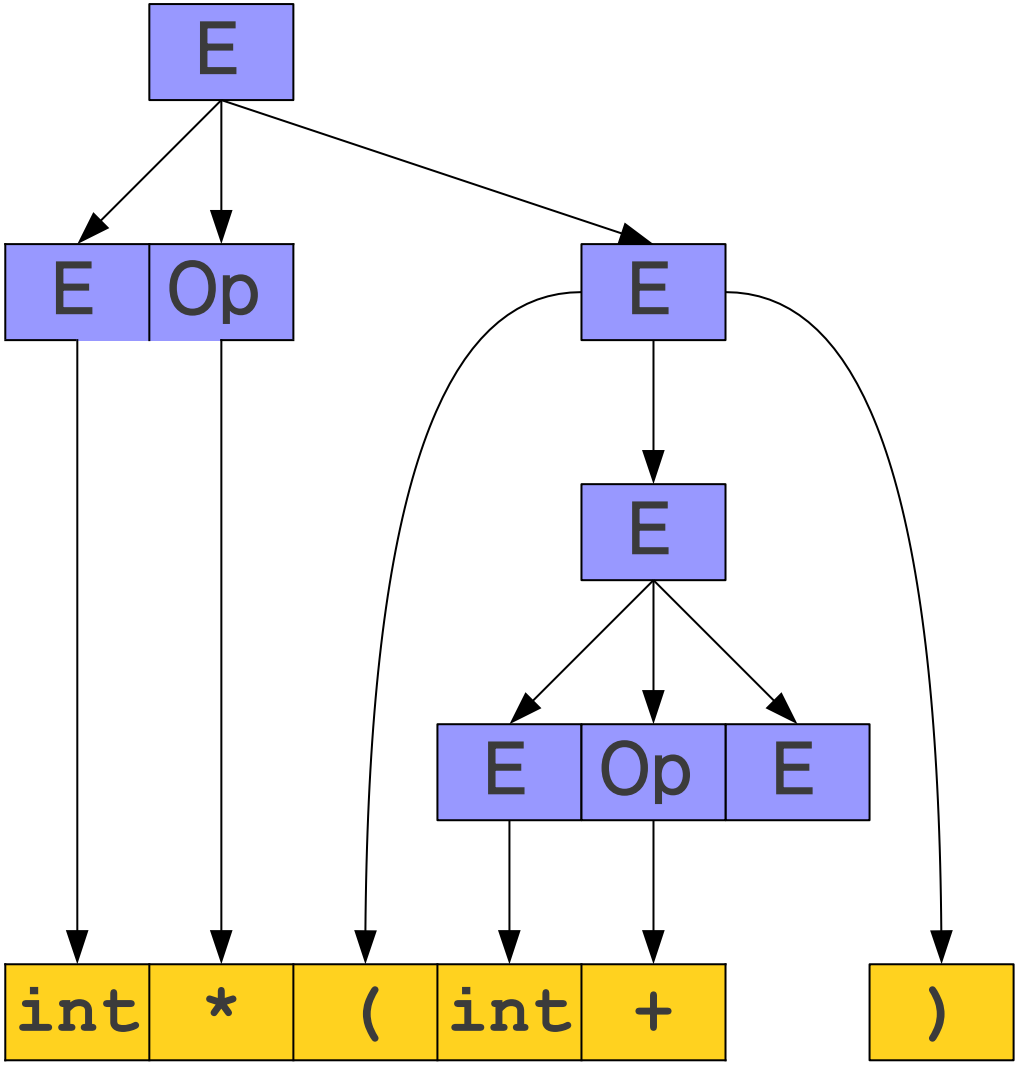
⇒ `int * (`E`)`

⇒ `int * (`E Op E`)`

⇒ `int * (int` Op E`)`

⇒ `int * (int +` E`)`

⇒ `int * (int + int)`



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ `int` Op E

⇒ `int *` E

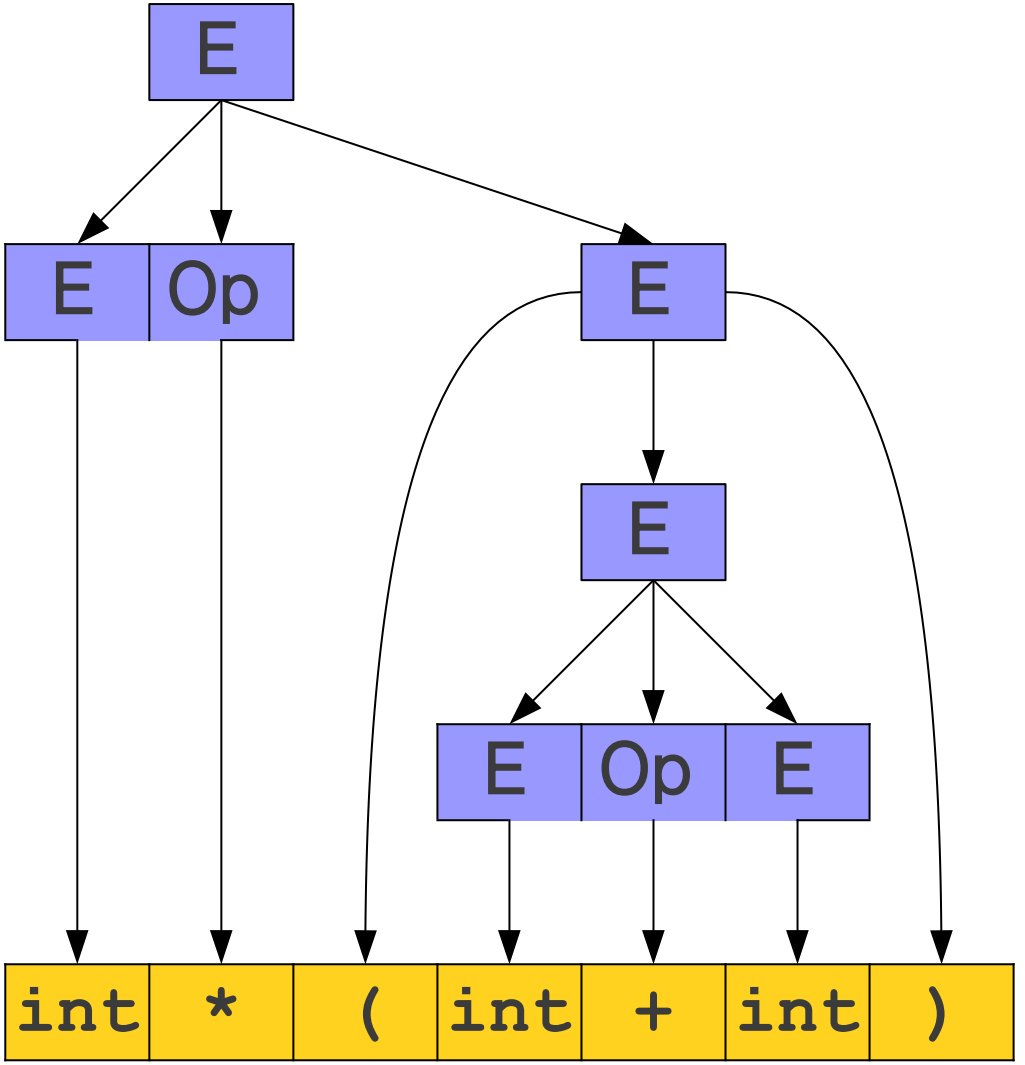
⇒ `int * (`E`)`

⇒ `int * (`E Op E`)`

⇒ `int * (int` Op E`)`

⇒ `int * (int +` E`)`

⇒ `int * (int + int)`



Parse Trees

$\Rightarrow$ `int * (int + int)`

E

Parse Trees

$\Rightarrow$ `int * (int + int)`

E

Parse Trees

E

$\Rightarrow$ `int * (int + int)`

E

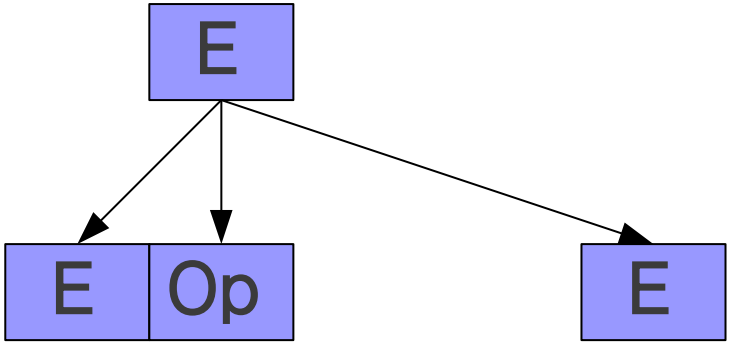
$\Rightarrow$ E Op E

Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E



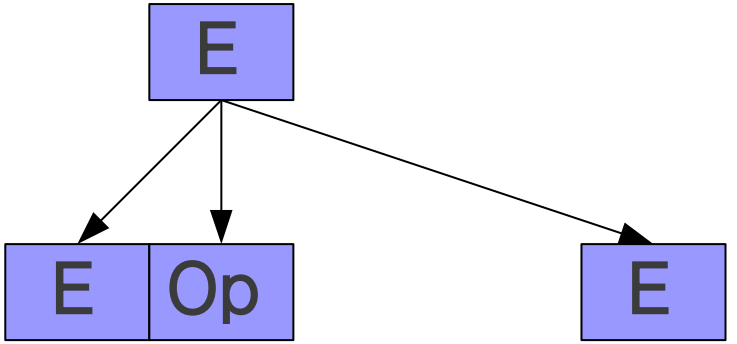
Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ E Op `(`E`)`



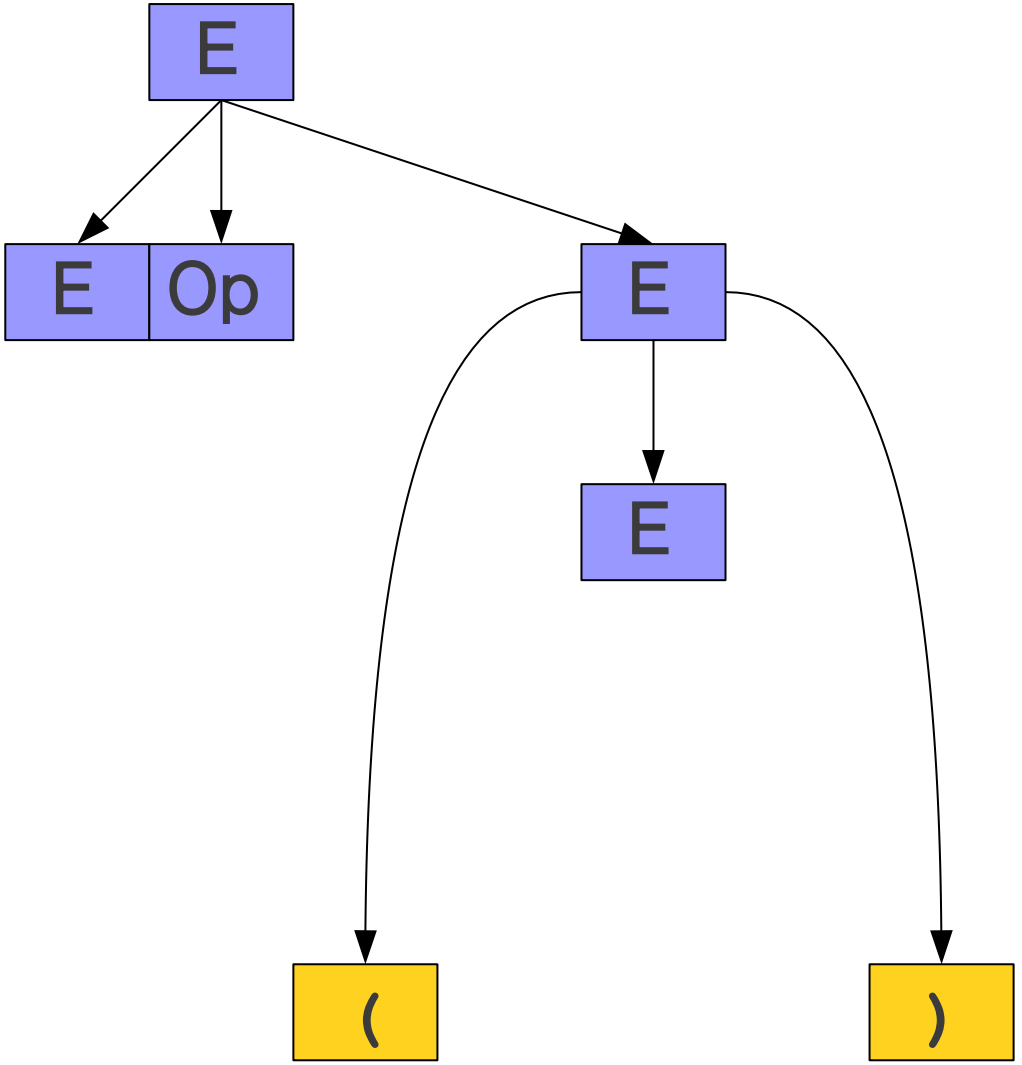
Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ E Op `(`E`)`



Parse Trees

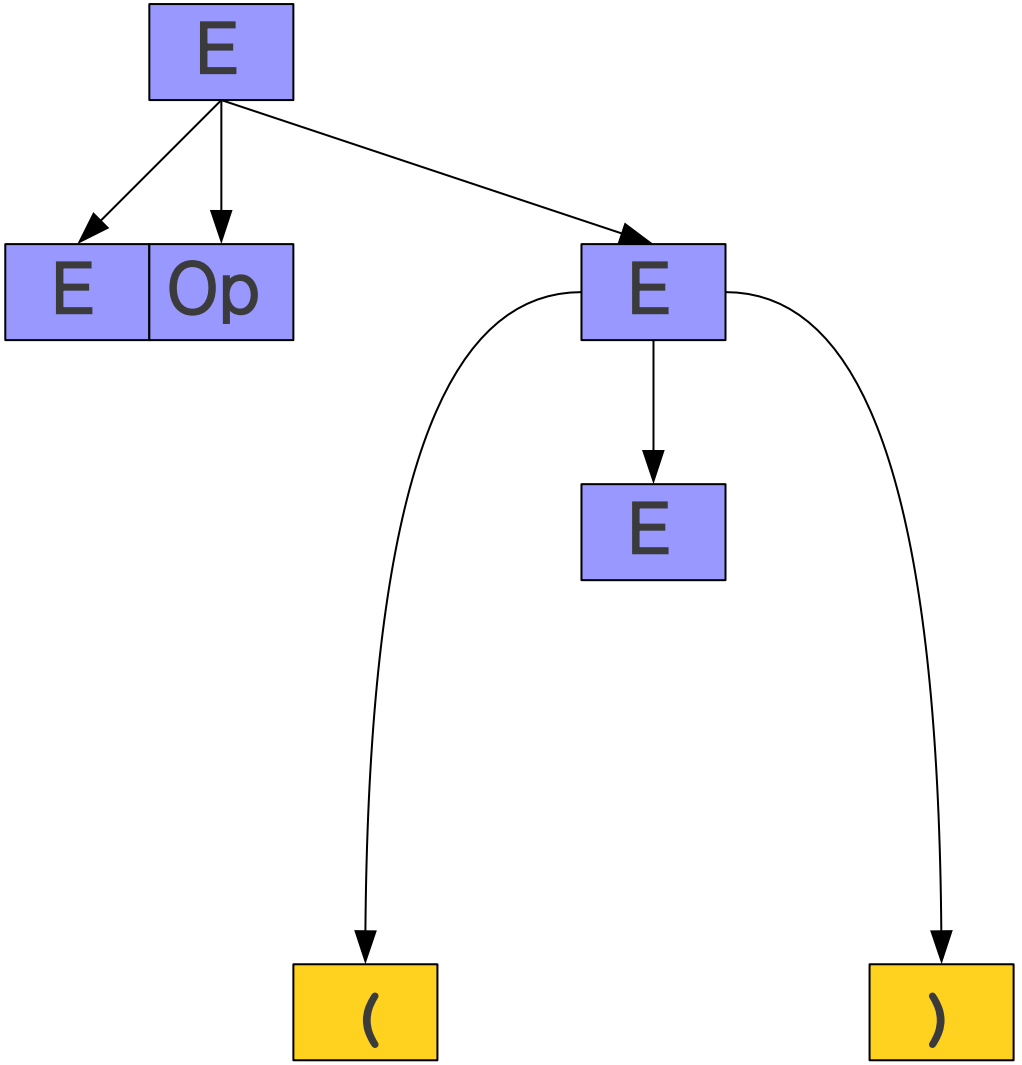
⇒ `int * (int + int)`

E

⇒ E Op E

⇒ E Op `(`E`)`

⇒ E Op `(`E Op E`)`



Parse Trees

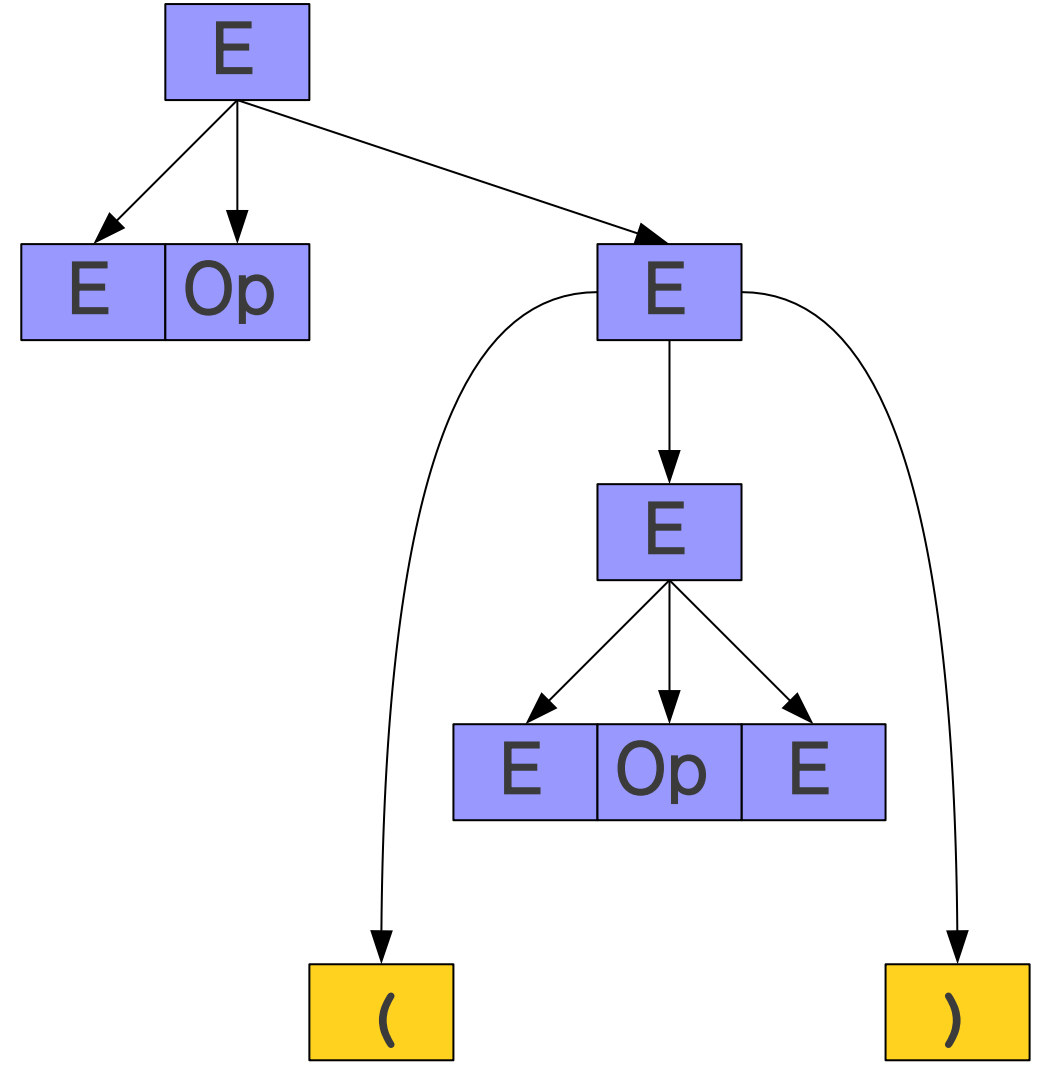
⇒ `int * (int + int)`

E

⇒ E Op E

⇒ E Op `(`E`)`

⇒ E Op `(`E Op E`)`



Parse Trees

⇒ `int * (int + int)`

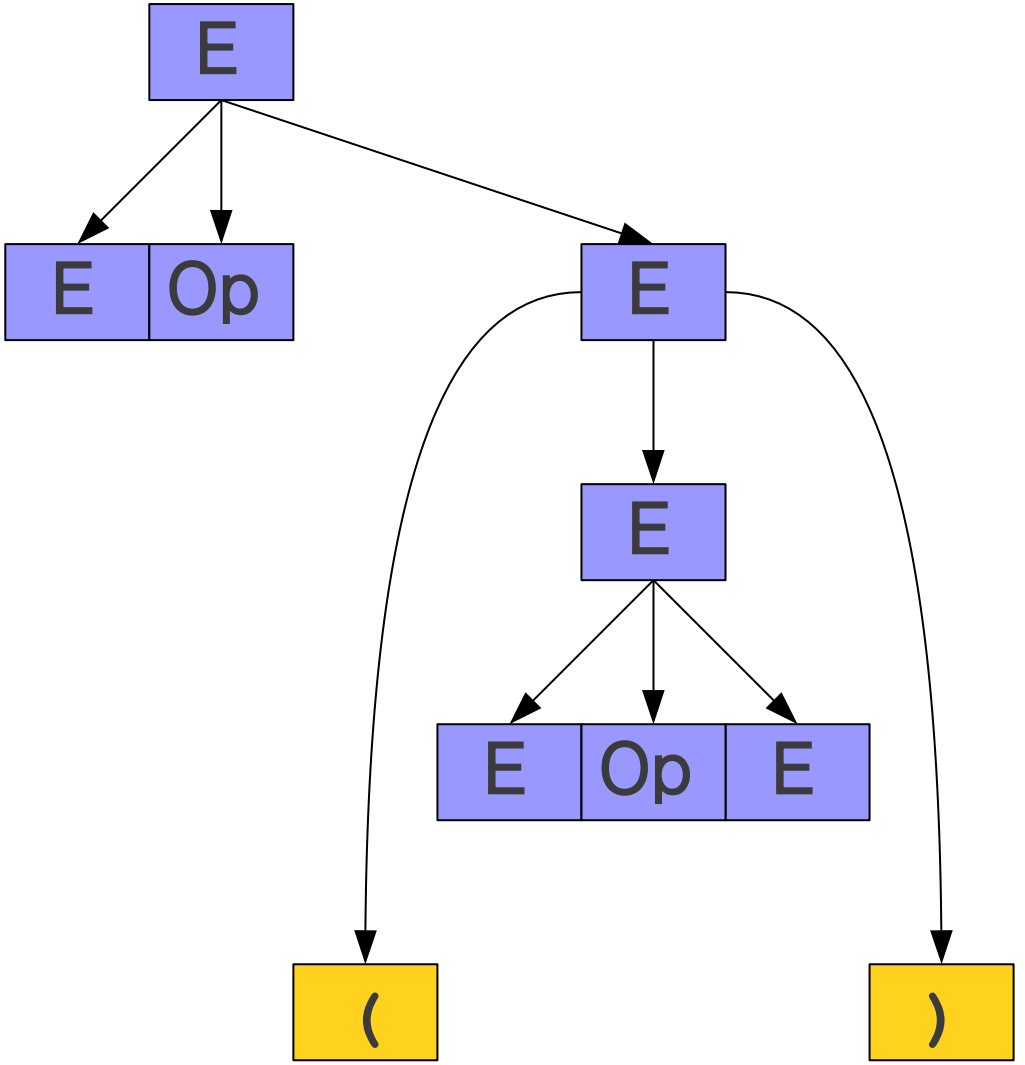
E

⇒ E Op E

⇒ E Op `(`E`)`

⇒ E Op `(`E Op E`)`

⇒ E Op `(`E Op `int)`



Parse Trees

⇒ `int * (int + int)`

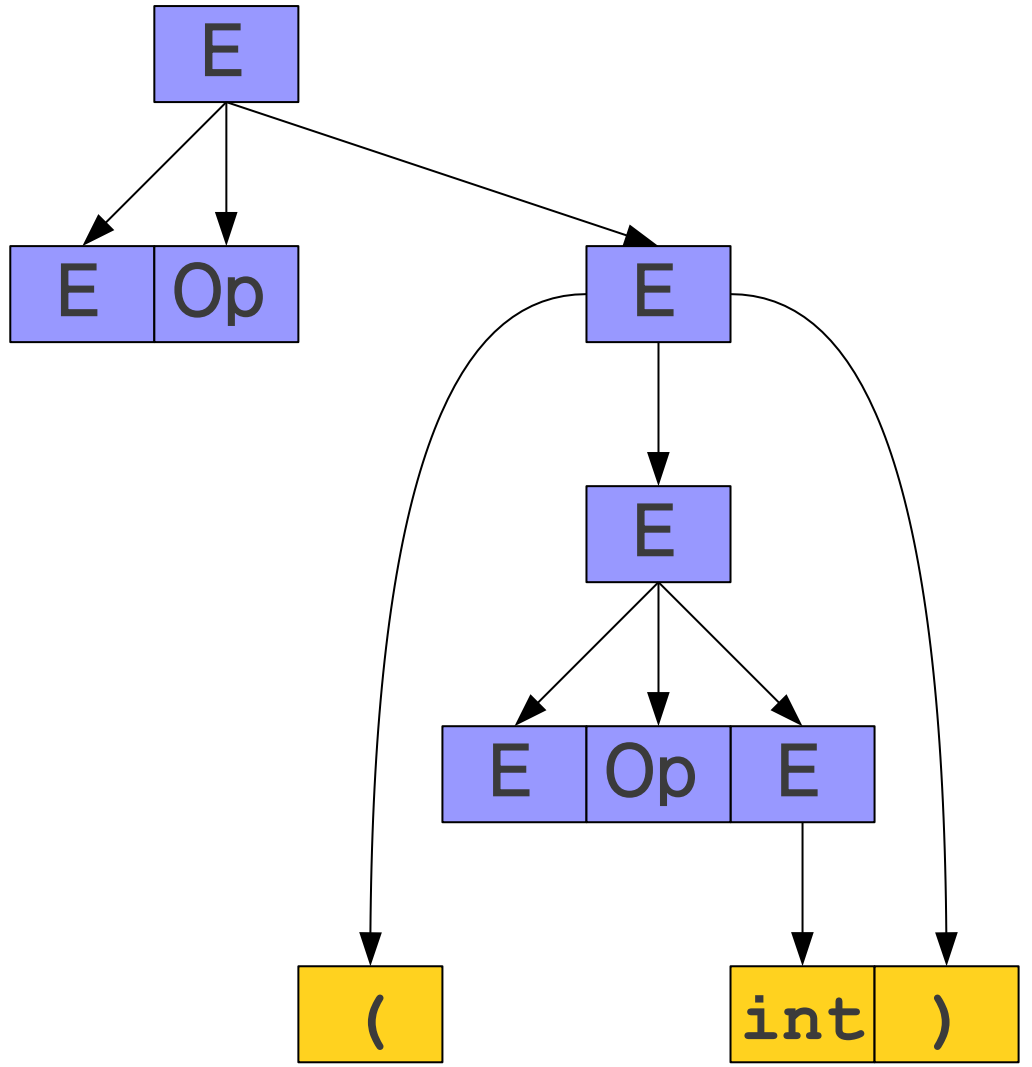
E

⇒ E Op E

⇒ E Op `(`E`)`

⇒ E Op `(`E Op E`)`

⇒ E Op `(`E Op `int)`



Parse Trees

⇒ `int * (int + int)`

E

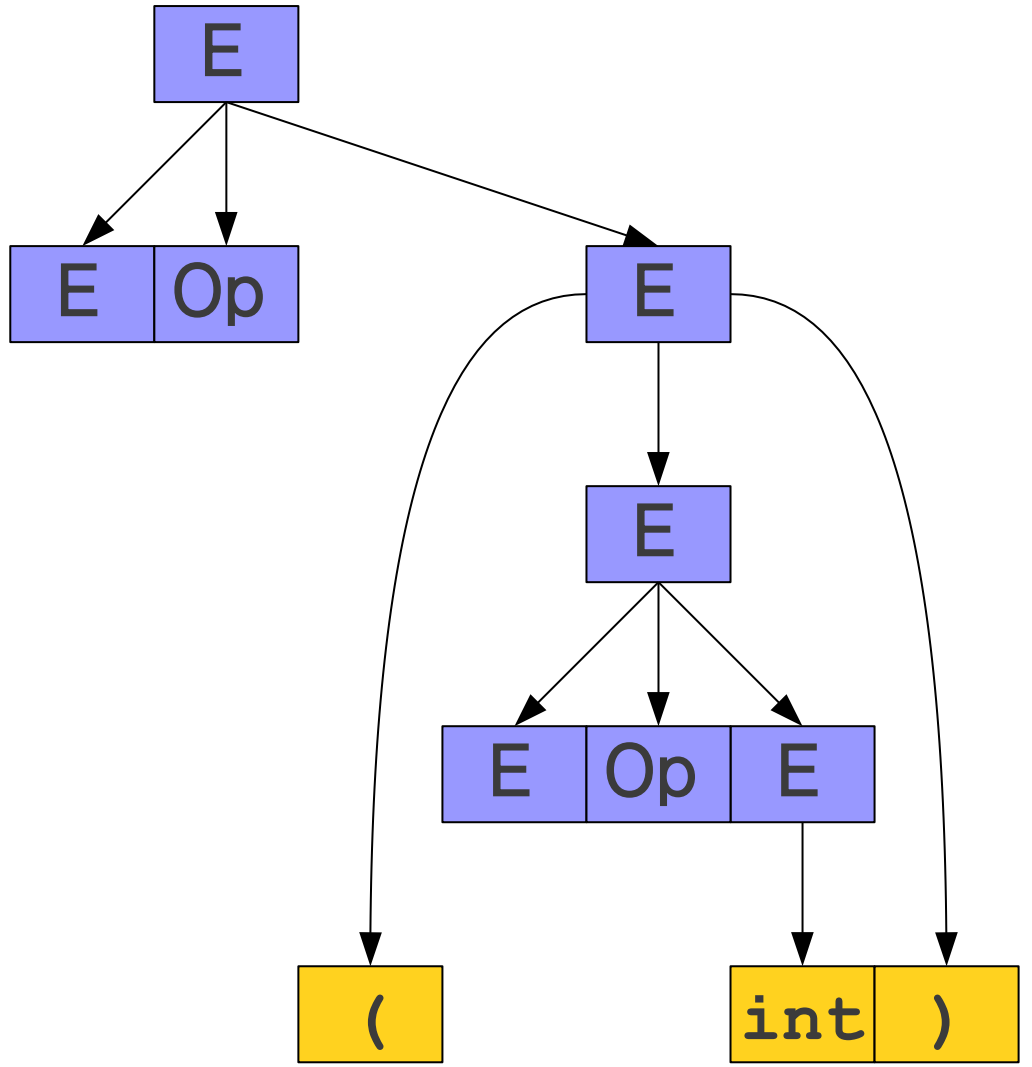
⇒ E Op E

⇒ E Op `(`E`)`

⇒ E Op `(`E Op E`)`

⇒ E Op `(`E Op `int)`

⇒ E Op `(`E `+ int)`



Parse Trees

⇒ `int * (int + int)`

E

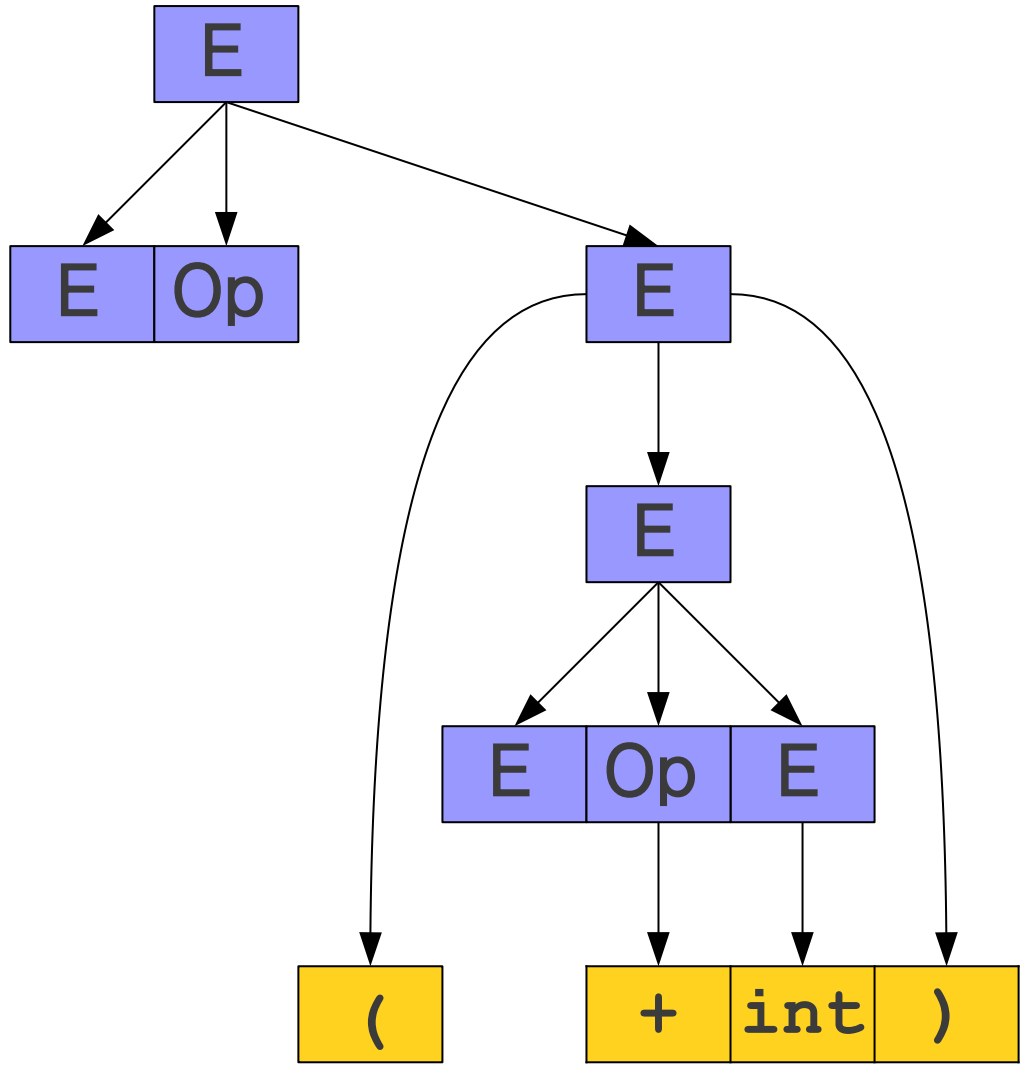
⇒ E Op E

⇒ E Op `(`E`)`

⇒ E Op `(`E Op E`)`

⇒ E Op `(`E Op `int)`

⇒ E Op `(`E `+ int)`



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

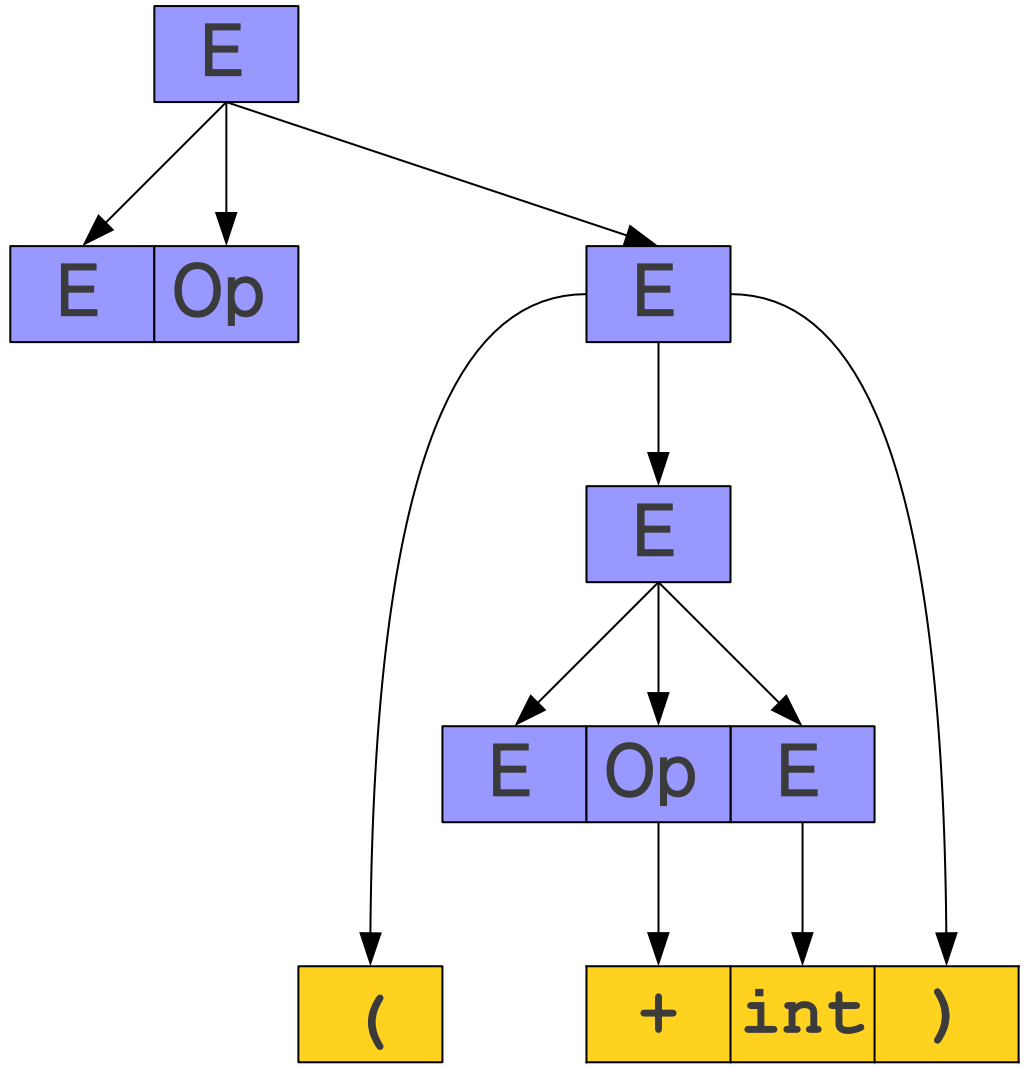
⇒ E Op `(`E`)`

⇒ E Op `(`E Op E`)`

⇒ E Op `(`E Op `int)`

⇒ E Op `(`E `+ int)`

⇒ E Op `(int + int)`



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

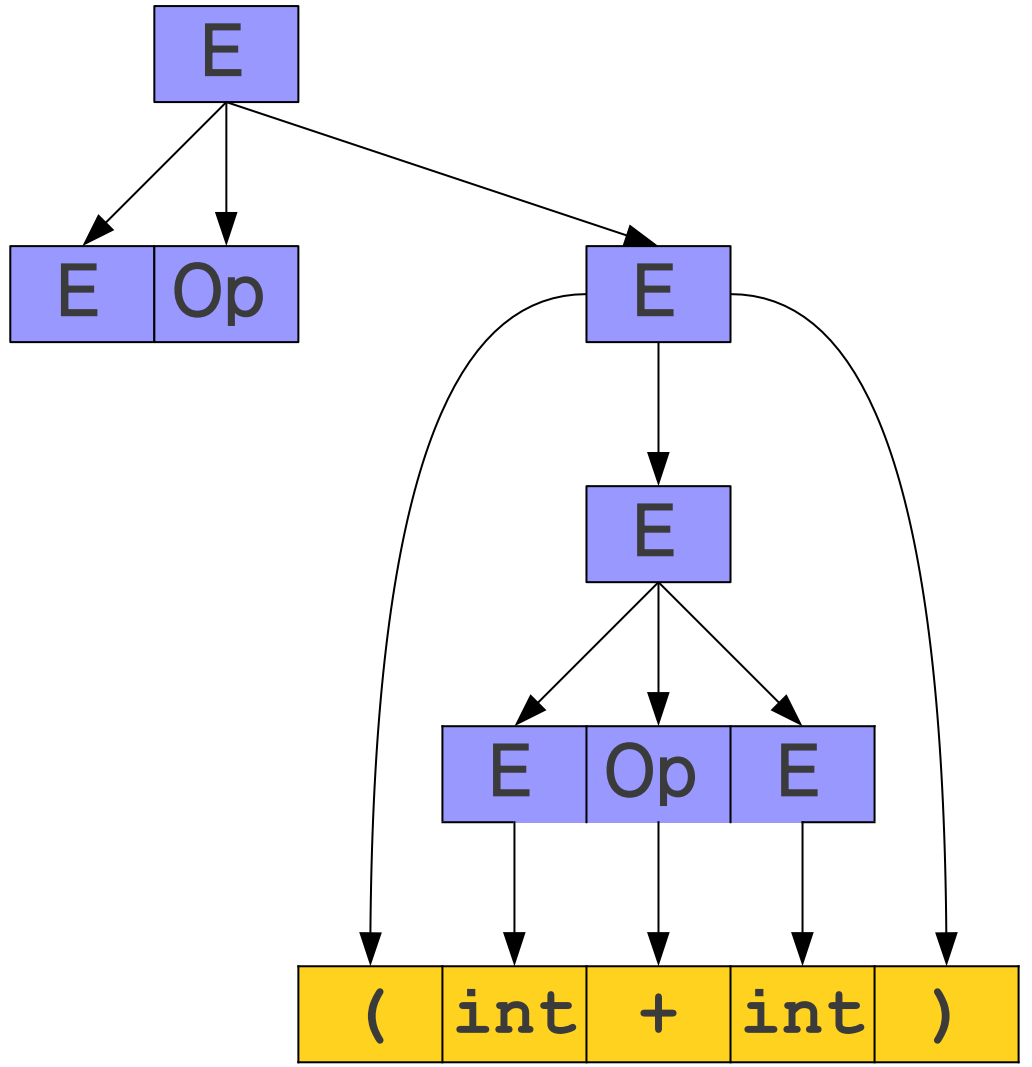
⇒ E Op `(`E`)`

⇒ E Op `(`E Op E`)`

⇒ E Op `(`E Op `int)`

⇒ E Op `(`E `+ int)`

⇒ E Op `(int + int)`



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ E Op (E)

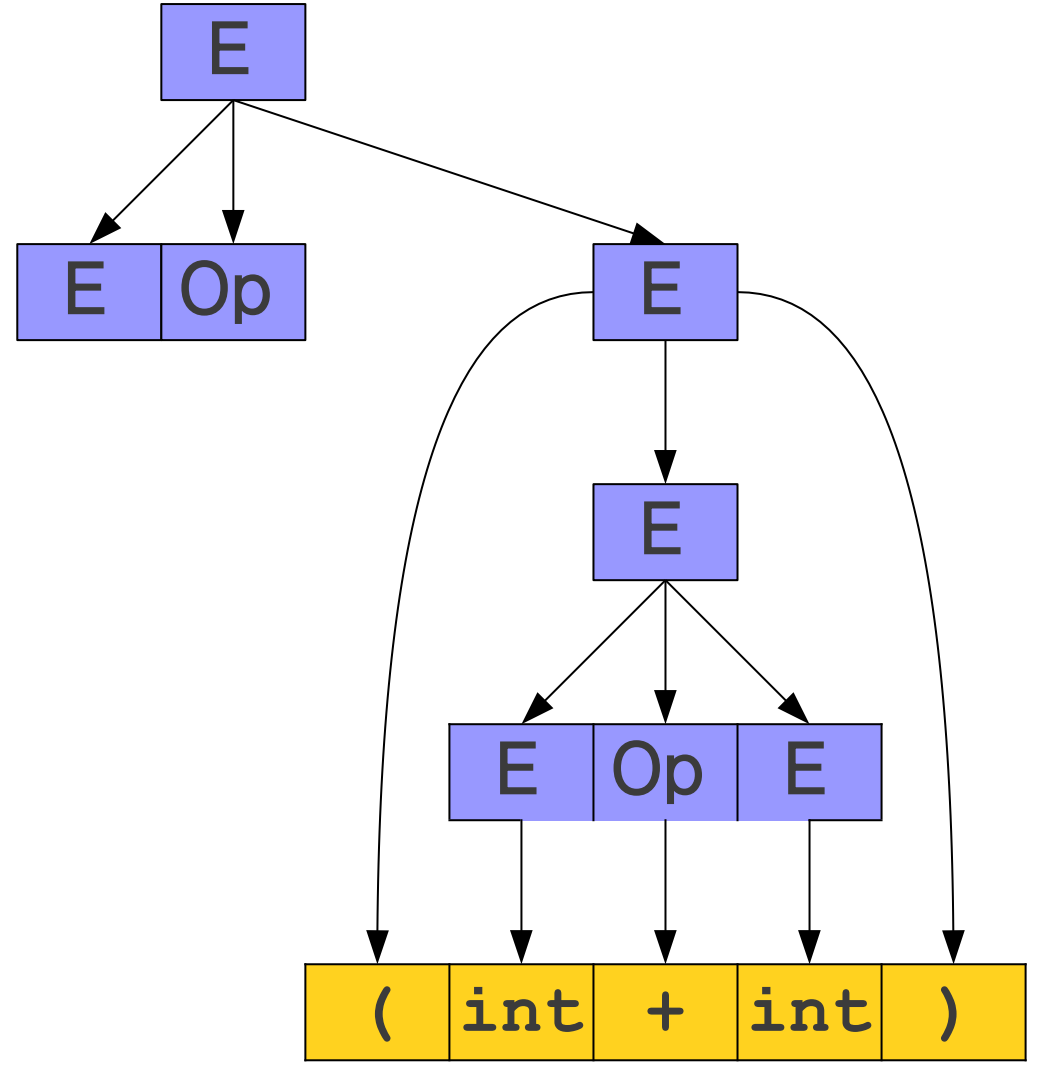
⇒ E Op (E Op E)

⇒ E Op (E Op `int`)

⇒ E Op (E + `int`)

⇒ E Op (`int + int`)

⇒ E * (`int + int`)



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ E Op (E)

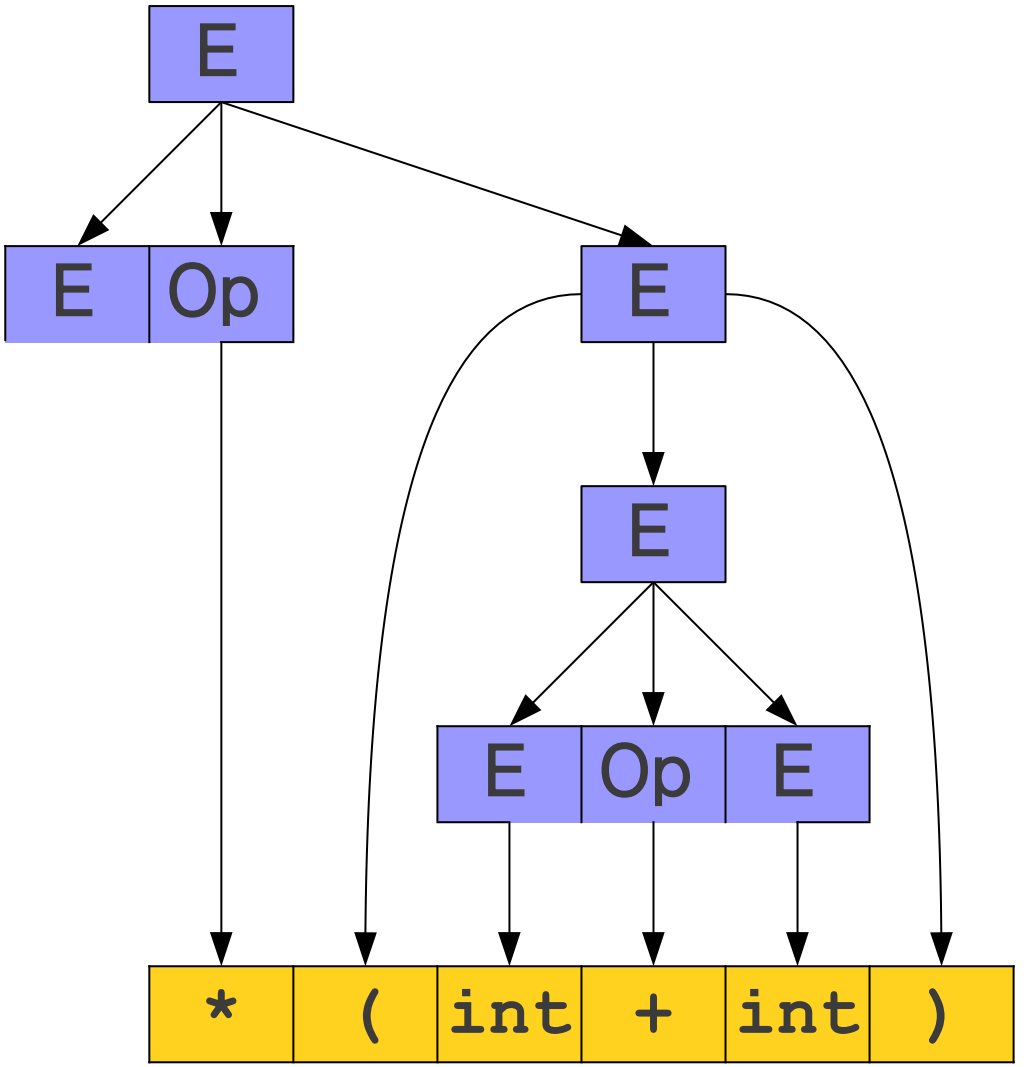
⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)

⇒ E * (int + int)



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

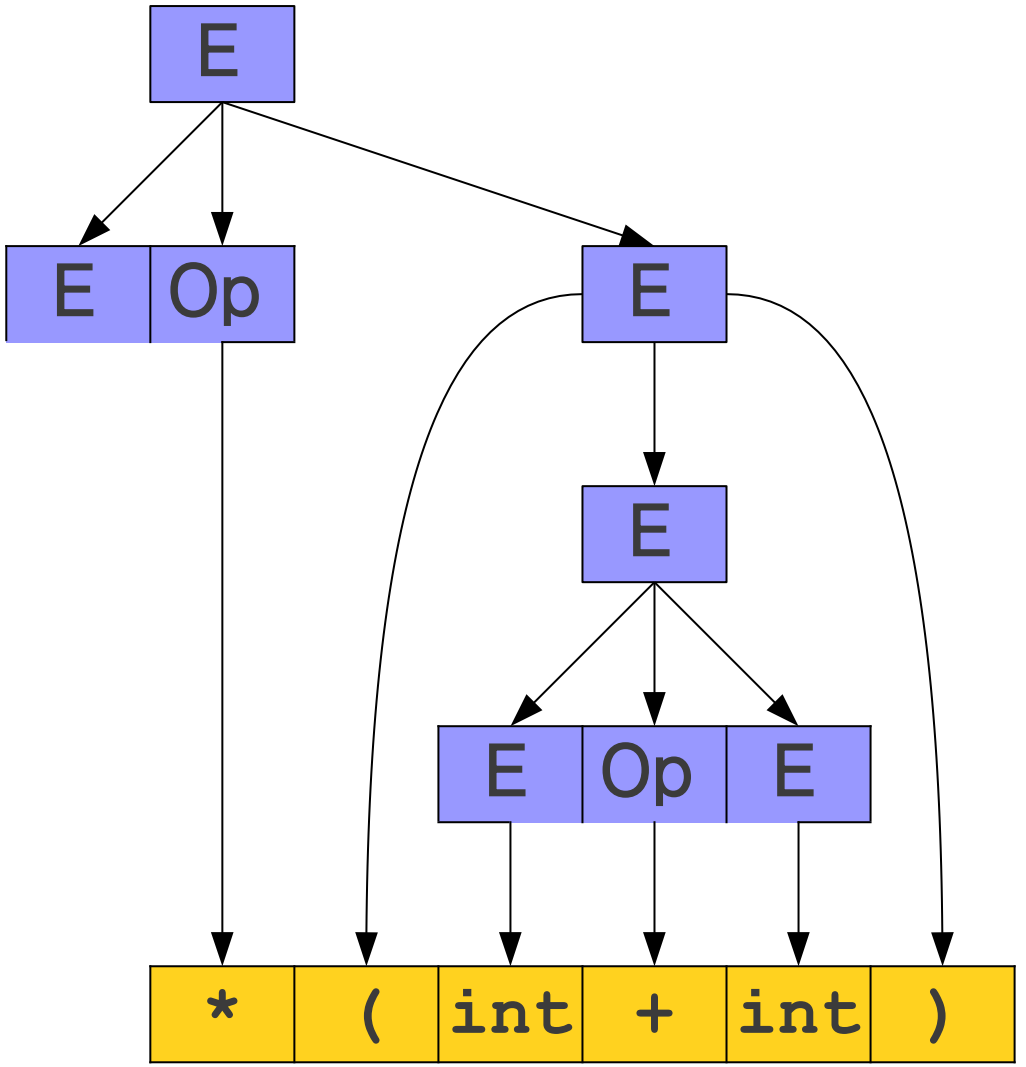
⇒ E Op (E Op `int`)

⇒ E Op (E + `int`)

⇒ E Op (`int + int`)

⇒ E * (`int + int`)

⇒ `int * (int + int)`



Parse Trees

⇒ `int * (int + int)`

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

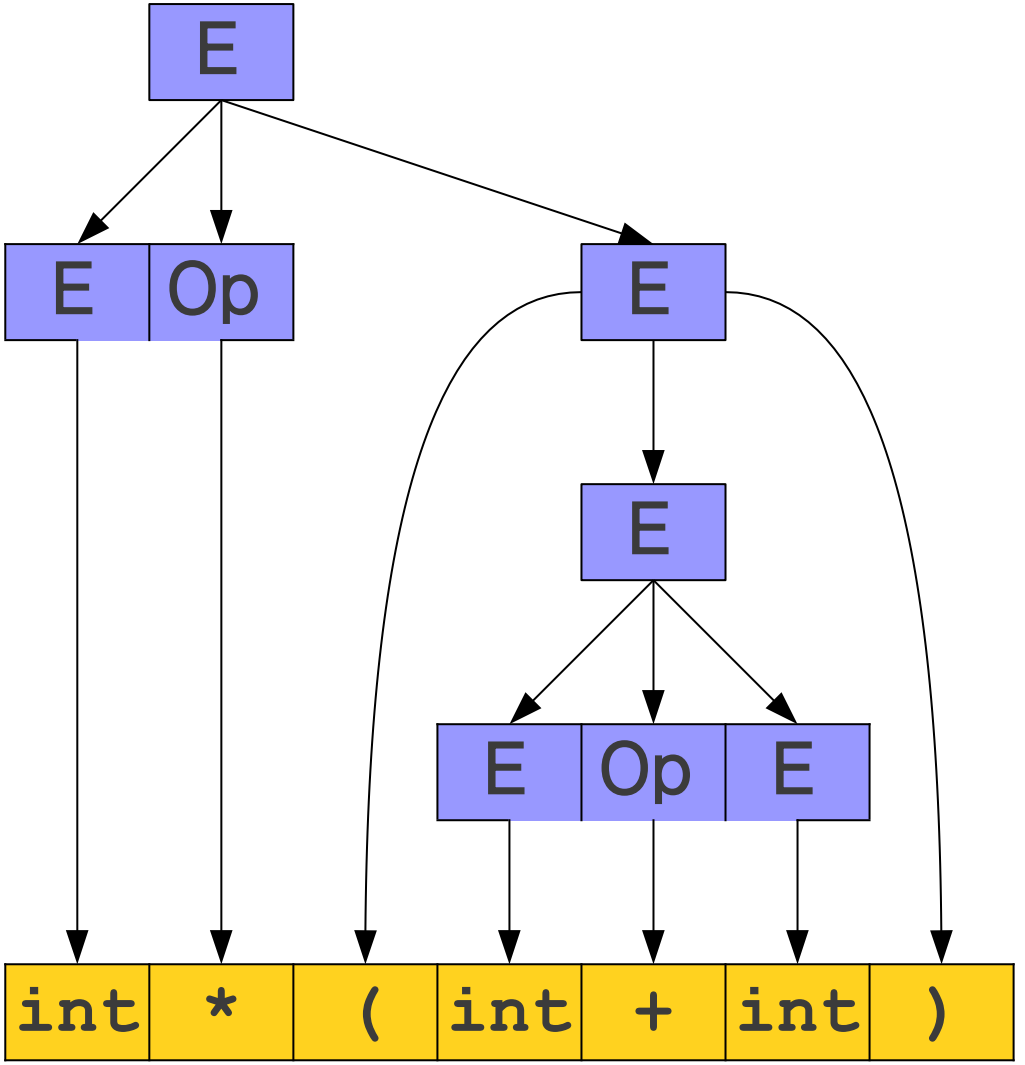
⇒ E Op (E Op int)

⇒ E Op (E + int)

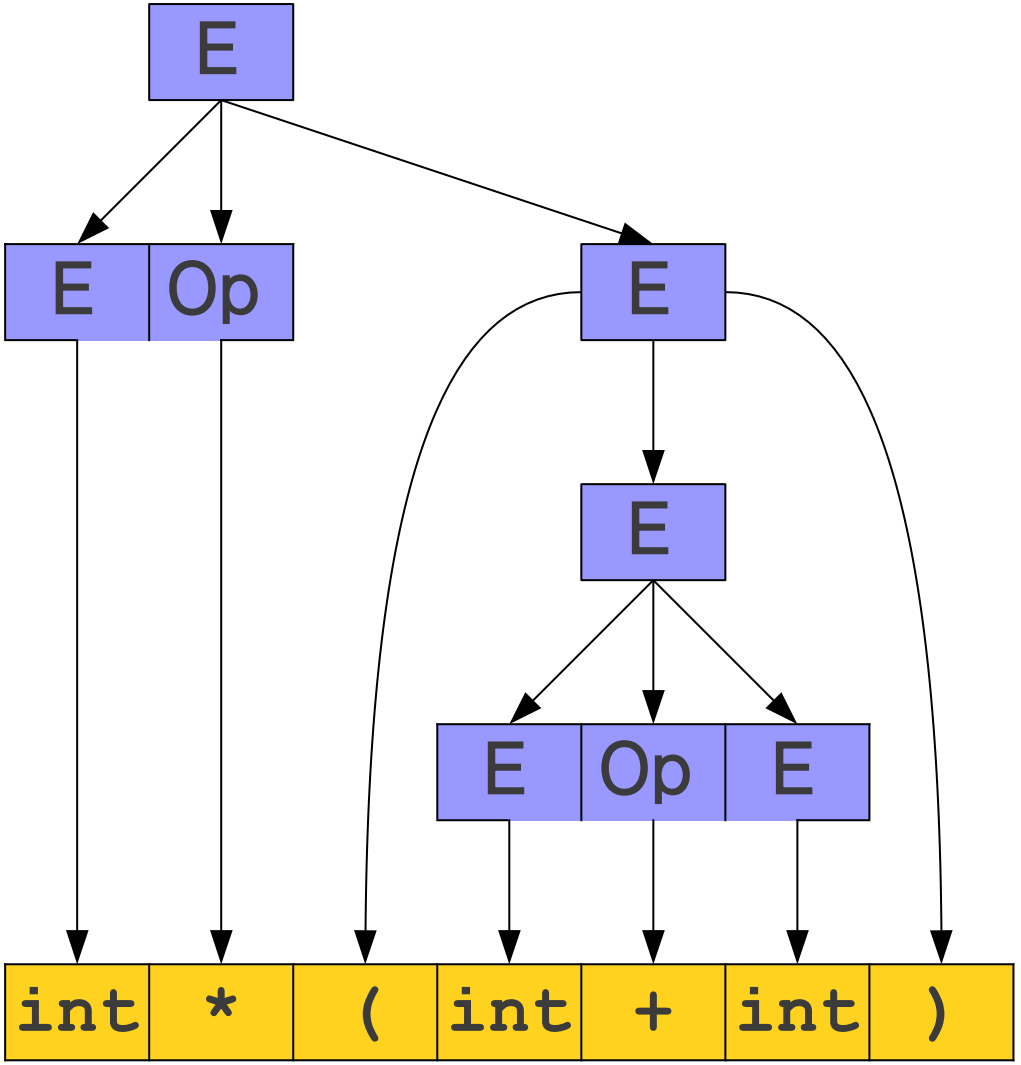
⇒ E Op (int + int)

⇒ E * (int + int)

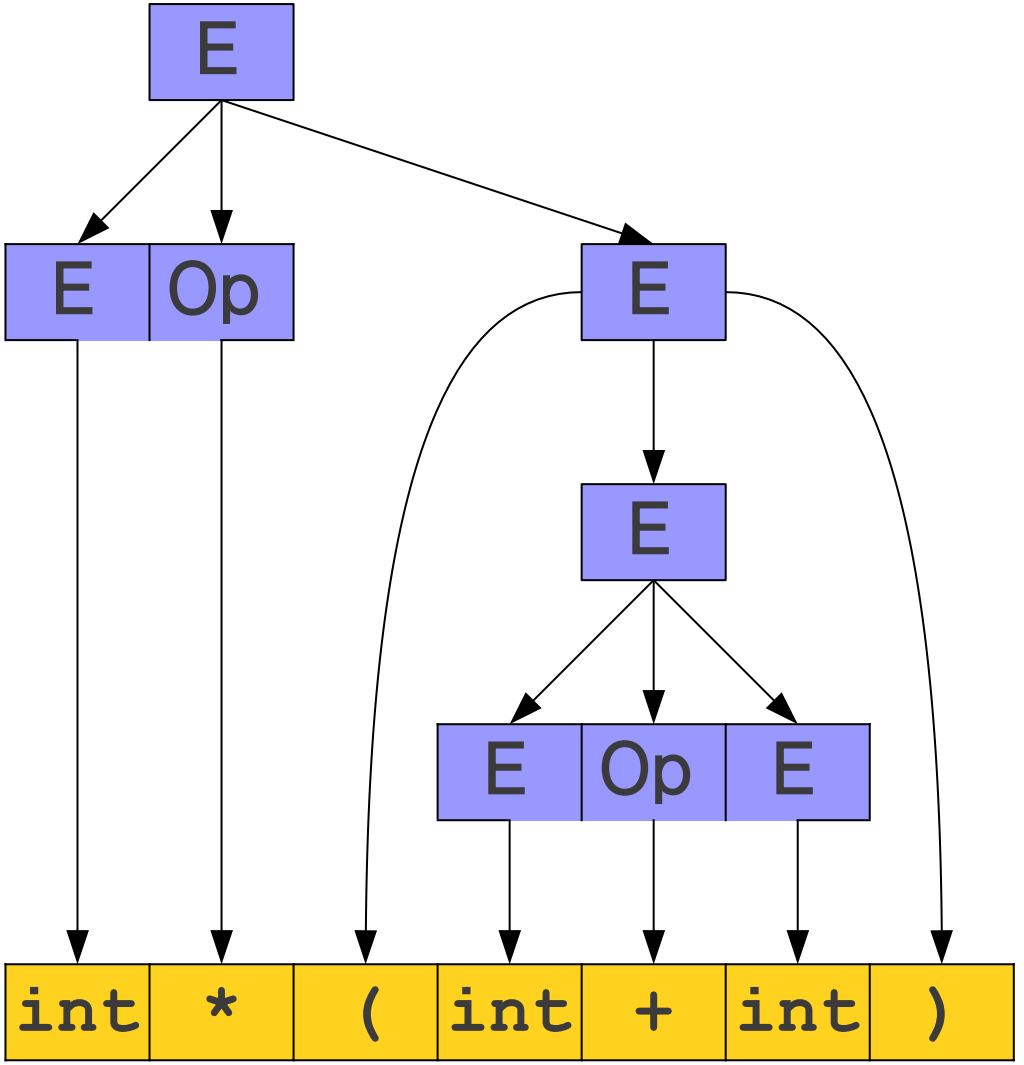
⇒ `int * (int + int)`



For Comparison



Parse Tree with Leftmost Derivation



Parse Tree with Rightmost Derivation

Parse Trees

- A **parse tree** is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Encodes what productions are used, not the order in which those productions are applied.

The Goal of Parsing

- Goal of syntax analysis: Recover the structure described by a series of tokens.
- If language is described as a CFG, goal is to recover a parse tree for the the input string.
- We'll discuss how to do this next week.

Challenges in Parsing

Challenge

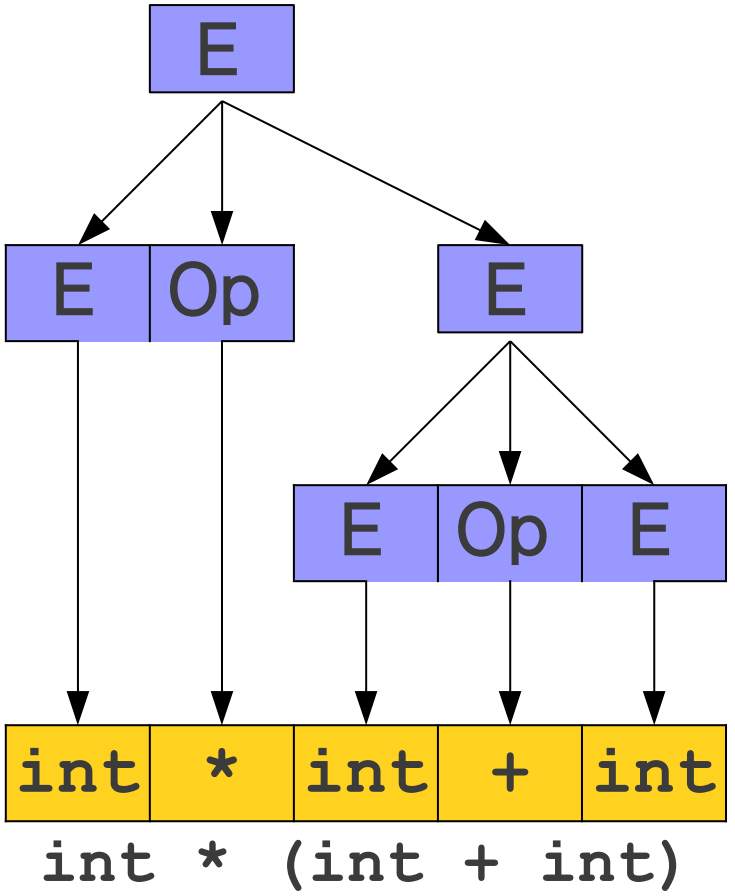
E → `int` | E Op E

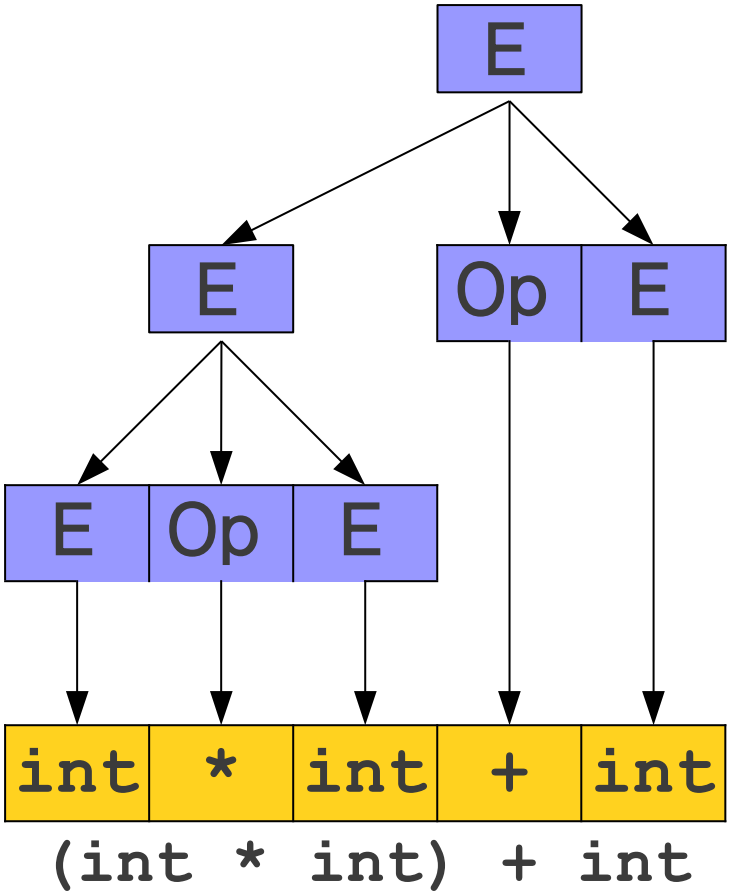
Op → `+` | `*` |

```
int * int + int
```

Can you construct Parse Tree?

A Serious Problem





Ambiguity

- A CFG is said to be **ambiguous** if there is at least one string with two or more parse trees.

Is Ambiguity a Problem?

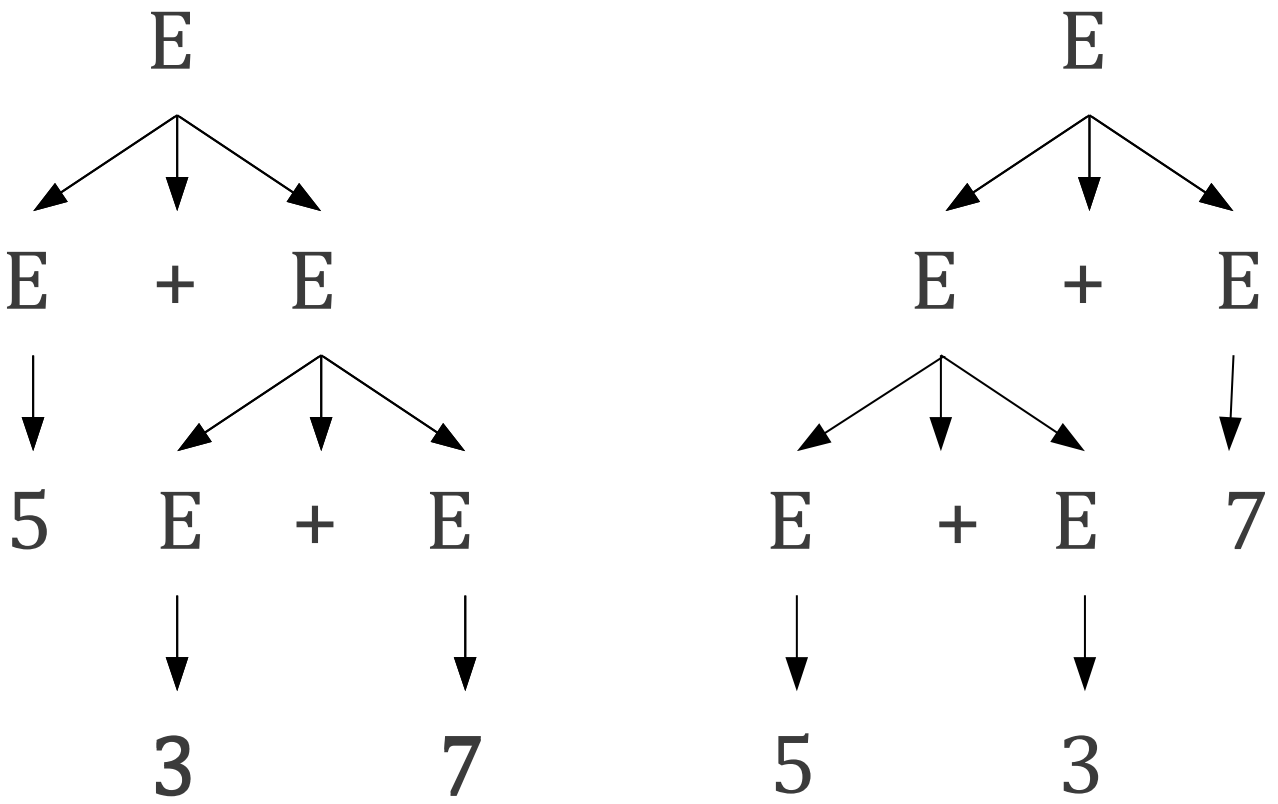
- Depends on **semantics**.

$$E \rightarrow \texttt{int} \mid E \texttt{ + } E$$

Is Ambiguity a Problem?

- Depends on semantics.

$$E \rightarrow \texttt{int} \mid E \texttt{ + } E$$



Is Ambiguity a Problem?

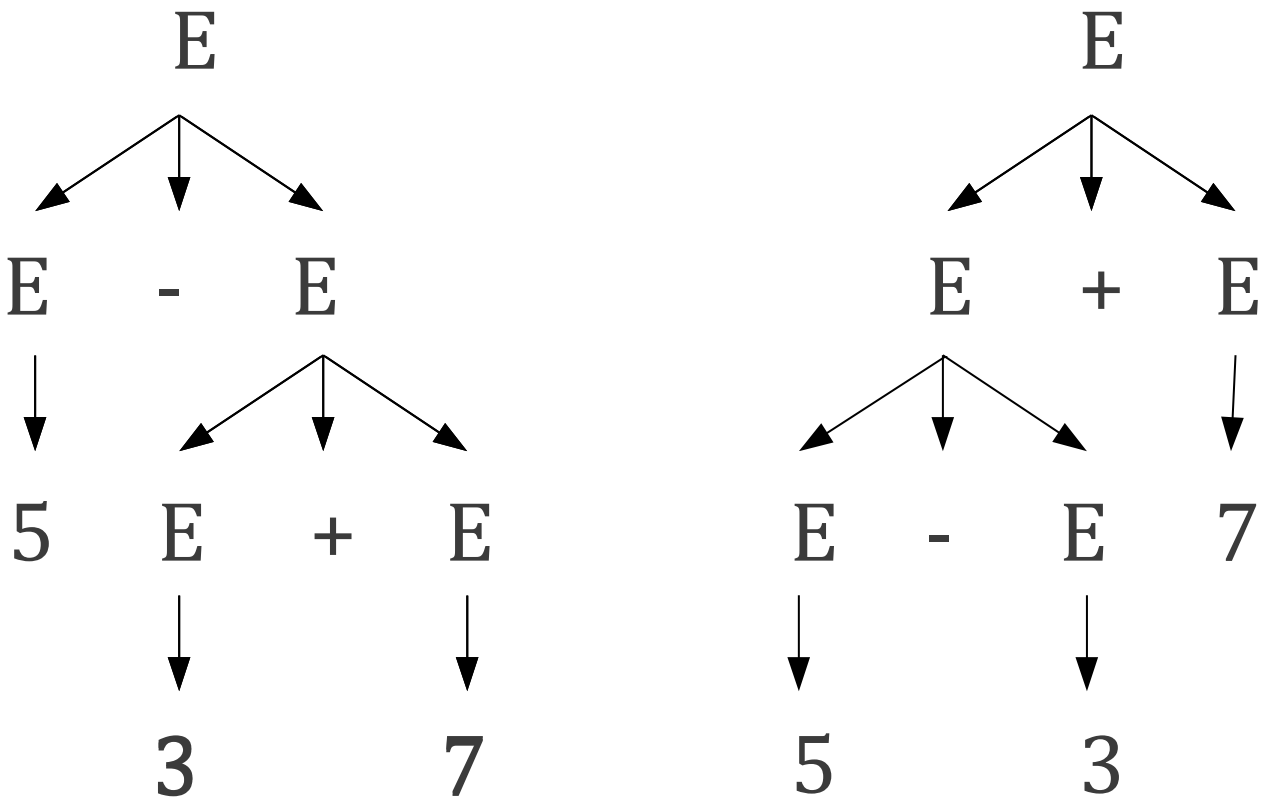
- Depends on semantics.

$$E \rightarrow \texttt{int} \mid E \texttt{ + } E \mid E \texttt{ - } E$$

Is Ambiguity a Problem?

- Depends on semantics.

$$E \rightarrow \texttt{int} \mid E + E \mid E - E$$



Ambiguity

- Ambiguity is problematic because meaning of the programs can be incorrect
- Ambiguity can be handled in several ways
  - Enforce associativity and precedence
  - Rewrite the grammar (cleanest way)
- There is no algorithm to convert automatically any ambiguous grammar to an unambiguous grammar accepting the same language
- Worse, there are inherently ambiguous languages!

Ambiguity in Programming Lang.

- Dangling else problem

  stmt → if expr stmt

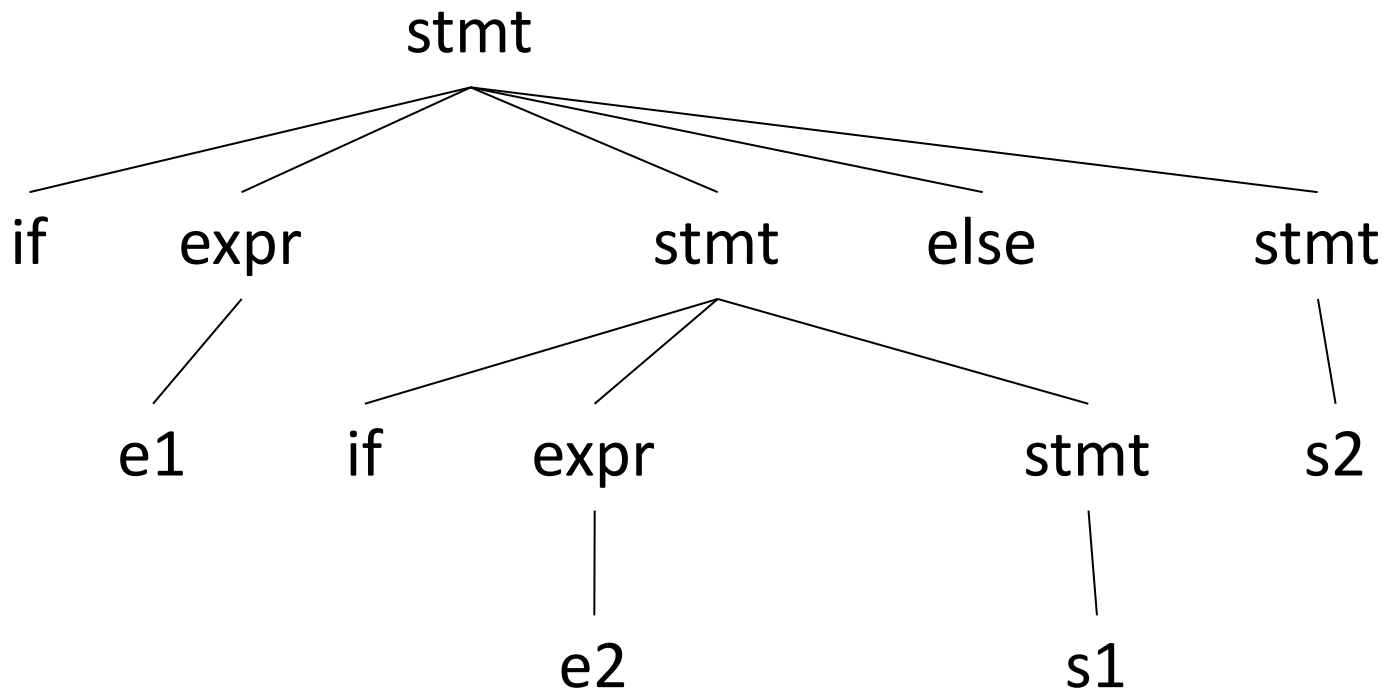
  | if expr stmt else stmt

- For this grammar, the string

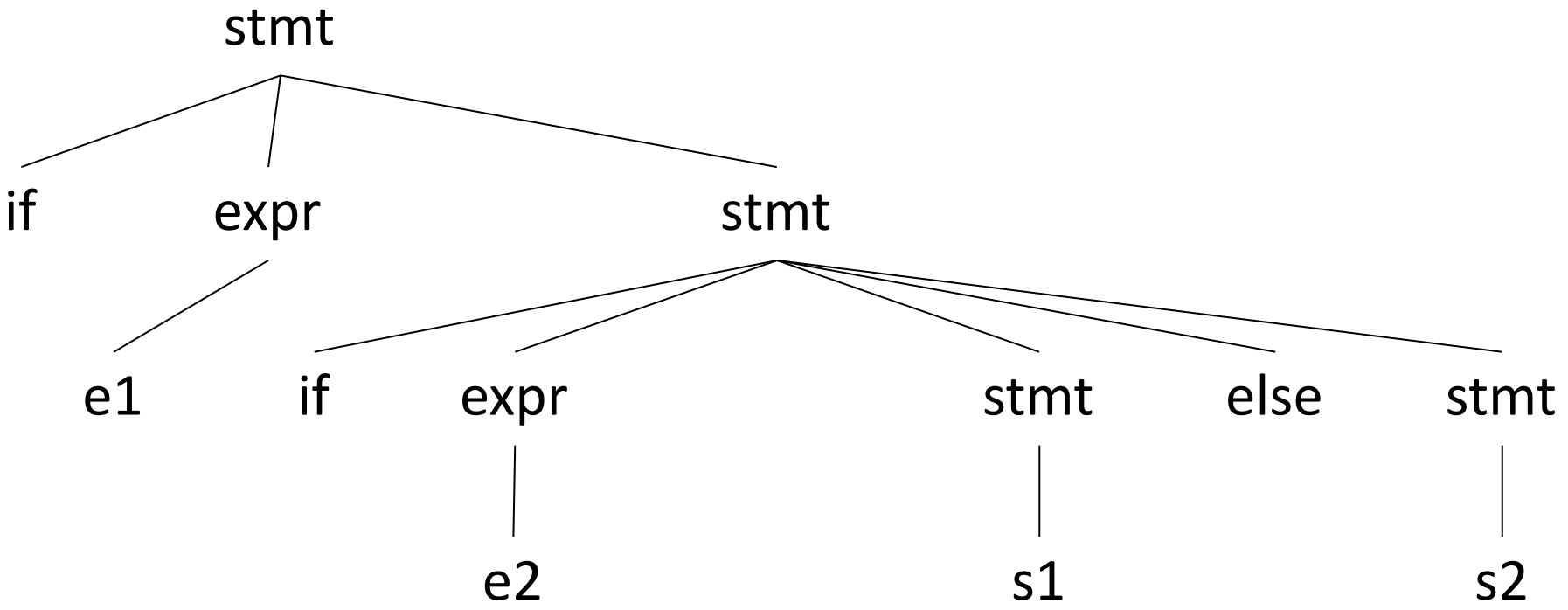
  if e1 if e2 then s1 else s2

  has two parse trees

```
if e1
   if e2
      s1
else s2
```




```
if e1
   if e2
      s1
   else s2
```



Resolving dangling else problem

- General rule: match each **else** with the closest previous **unmatched if**.

Precedence

- String a+5*2 has two possible interpretations because of two different parse trees corresponding to
  (a+5)*2 and a+(5*2)
- Precedence determines the correct interpretation.
- Next lectures, we will see more details about precedence/associativity on resolving the ambiguity

Summary

- A **parse tree** shows how a string can be **derived** from a grammar.
- A grammar is **ambiguous** if it can derive the same string multiple ways.
- Next time: Parsing algorithms
  - Top-Down Parsing
  - Bottom-Up Parsing