

# Context-Free Grammars

# Describing Languages

- We've seen two models for the regular languages:
  - **Automata** accept precisely the strings in the language.
  - **Regular expressions** describe precisely the strings in the language.
- Finite automata **recognize** strings in the language.
  - Perform a computation to determine whether a specific string is in the language.
- Regular expressions **match** strings in the language.
  - Describe the general shape of all strings in the language.

# Context-Free Grammars

- A **context-free grammar** (or **CFG**) is an entirely different formalism for defining a class of languages.
- Goal: Give a procedure for listing off all strings in the language.
- CFGs are best explained by example...

# Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

**E** → **int**

**E** → **E Op E**

**E** → **(E)**

**Op** → **+**

**Op** → **-**

**Op** → **\***

**Op** → **/**

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

# Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

**E** → **int**

**E** → **E Op E**

**E** → **(E)**

**Op** → **+**

**Op** → **-**

**Op** → **\***

**Op** → **/**

**E**

⇒ **E Op E**

⇒ **E Op int**

⇒ **int Op int**

⇒ **int / int**

# Context-Free Grammars

- Formally, a context-free grammar is a collection of four objects:
  - A set of **nonterminal symbols** (also called **variables**),
  - A set of **terminal symbols** (the **alphabet** of the CFG)
  - A set of **production rules** saying how each nonterminal can be converted by a string of terminals and nonterminals, and
  - A **start symbol** (which must be a nonterminal) that begins the derivation.

**E** → **int**

**E** → **E Op E**

**E** → **( E )**

**Op** → **+**

**Op** → **-**

**Op** → **\***

**Op** → **/**

# Some CFG Notation

- Capital letters in **Bold Red Uppercase** will represent nonterminals.
  - i.e. **A**, **B**, **C**, **D**
- Lowercase letters in **blue monospace** will represent terminals.
  - i.e. **t**, **u**, **v**, **w**
- Lowercase Greek letters in *gray italics* will represent arbitrary strings of terminals and nonterminals.
  - i.e.  *$\alpha$* ,  *$\gamma$* ,  *$\omega$*

# A Notational Shorthand

**E**  $\rightarrow$  **int**

**E**  $\rightarrow$  **E Op E**

**E**  $\rightarrow$  **(E)**

**Op**  $\rightarrow$  **+**

**Op**  $\rightarrow$  **-**

**Op**  $\rightarrow$  **\***

**Op**  $\rightarrow$  **/**



# A Notational Shorthand

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

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

# Derivations

$\begin{aligned} E &\rightarrow E \text{ Op } E \mid \text{int} \mid (E) \\ \text{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------

$E$   
 $\Rightarrow E \text{ Op } E$   
 $\Rightarrow E \text{ Op } (E)$   
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$   
 $\Rightarrow E * (E \text{ Op } E)$   
 $\Rightarrow \text{int} * (E \text{ Op } E)$   
 $\Rightarrow \text{int} * (\text{int} \text{ Op } E)$   
 $\Rightarrow \text{int} * (\text{int} \text{ Op } \text{int})$   
 $\Rightarrow \text{int} * (\text{int} + \text{int})$

- A sequence of steps where nonterminals are replaced by the right-hand side of a production is called a **derivation**.
- If string  $\alpha$  derives string  $\omega$ , we write  $\alpha \Rightarrow^* \omega$ .
- In the example on the left, we see  $E \Rightarrow^* \text{int} * (\text{int} + \text{int})$ .

# The Language of a Grammar

- If  $G$  is a CFG with alphabet  $\Sigma$  and start symbol **S**, then the **language of  $G$**  is the set

$$\mathcal{L}(G) = \{ \omega \in \Sigma^* \mid \mathbf{S} \Rightarrow^* \omega \}$$

- That is,  $\mathcal{L}(G)$  is the set of strings derivable from the start symbol.
- Note:  $\omega$  must be in  $\Sigma^*$ , the set of strings made from terminals. Strings involving nonterminals aren't in the language.

# More Context-Free Grammars

- Chemicals!



**Form**  $\rightarrow$  **Cmp** | **Cmp Ion**

**Cmp**  $\rightarrow$  **Term** | **Term Num** | **Cmp Cmp**

**Term**  $\rightarrow$  **Elem** | **(Cmp)**

**Elem**  $\rightarrow$  **H** | **He** | **Li** | **Be** | **B** | **C** | ...

**Ion**  $\rightarrow$  **+** | **-** | **IonNum +** | **IonNum -**

**IonNum**  $\rightarrow$  **2** | **3** | **4** | ...

**Num**  $\rightarrow$  **1** | **IonNum**

# CFGs for Chemistry

**Form**  $\rightarrow$  **Cmp** | **Cmp Ion**

**Cmp**  $\rightarrow$  **Term** | **Term Num** | **Cmp Cmp**

**Term**  $\rightarrow$  **Elem** | **(Cmp)**

**Elem**  $\rightarrow$  **H** | **He** | **Li** | **Be** | **B** | **C** | ...

**Ion**  $\rightarrow$  **+** | **-** | **IonNum +** | **IonNum -**

**IonNum**  $\rightarrow$  **2** | **3** | **4** | ...

**Num**  $\rightarrow$  **1** | **IonNum**

**Form**

$\Rightarrow$  **Cmp Ion**

$\Rightarrow$  **Cmp Cmp Ion**

$\Rightarrow$  **Cmp Term Num Ion**

$\Rightarrow$  **Term Term Num Ion**

$\Rightarrow$  **Elem Term Num Ion**

$\Rightarrow$  **Mn Term Num Ion**

$\Rightarrow$  **Mn Elem Num Ion**

$\Rightarrow$  **MnO Num Ion**

$\Rightarrow$  **MnO IonNum Ion**

$\Rightarrow$  **MnO<sub>4</sub> Ion**

$\Rightarrow$  **MnO<sub>4</sub><sup>-</sup>**

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK**  
| **while** (**EXPR**) **BLOCK**  
| **do** **BLOCK** **while** (**EXPR**) ;  
| **BLOCK**  
| ...

**EXPR** → **var**  
| **const**  
| **EXPR** + **EXPR**  
| **EXPR** - **EXPR**  
| **EXPR** = **EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** **BLOCK**  
| **while** (**EXPR**) **BLOCK**  
| **do** **BLOCK** **while** (**EXPR**) ;  
| **BLOCK**  
| ...

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** **STMTS**  
| **do** **BLOCK** **while** (**EXPR**); }  
| **BLOCK**  
| ...

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...



# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** **STMT**  
| **do** **BLOCK** **while** (**EXPR**) ; **STMTS**  
| **BLOCK** }  
| ...

**EXPR** → **var**  
| **const**  
| **EXPR** + **EXPR**  
| **EXPR** - **EXPR**  
| **EXPR** = **EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** **EXPR;**  
| **do** **BLOCK** **while** (**EXPR**); **STMTS**  
| **BLOCK** }  
| ...

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** **EXPR = EXPR;**  
| **do** **BLOCK** **while** (**EXPR**); **STMTS**  
| **BLOCK** }  
| ...

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** **var = EXPR;**  
| **do** **BLOCK** **while** (**EXPR**); **STMTS**  
| **BLOCK** }  
| ...

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** **var = EXPR \* EXPR;**  
| **do** **BLOCK** **while** (**EXPR**); **STMTS**  
| **BLOCK** }  
| ...

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** **var = var \* EXPR;**  
| **do** **BLOCK** **while** (**EXPR**); **STMTS**  
| **BLOCK** }  
| ...

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** **var = var \* var;**  
| **do** **BLOCK** **while** (**EXPR**); **STMTS**  
| **BLOCK** }  
| ...

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** **var = var \* var;**  
| **do** **BLOCK** **while** (**EXPR**); **STMT**  
| **BLOCK** **STMTS**  
| ... }

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...



# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK**     **var = var \* var;**  
| **do** **BLOCK** **while** (**EXPR**);   **if** (**var**) **BLOCK**  
| **BLOCK**                         **STMTS**  
| ...                             }

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK**     **var = var \* var;**  
| **do** **BLOCK** **while** (**EXPR**);   **if** (**var**) **STMT**  
| **BLOCK**                       **STMTS**  
| ...                            }

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK**     **var = var \* var;**  
| **do** **BLOCK** **while** (**EXPR**);   **if** (**var**) **EXPR;**  
| **BLOCK**                       **STMTS**  
| ...                         }

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK**      **var = var \* var;**  
| **do** **BLOCK** **while** (**EXPR**);    **if** (**var**) **EXPR = EXPR;**  
| **BLOCK**                            **STMTS**  
| ...                                }

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK**     **var = var \* var;**  
| **do** **BLOCK** **while** (**EXPR**);   **if** (**var**) **var = EXPR;**  
| **BLOCK**                       **STMTS**  
| ...                         }

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK**     **var = var \* var;**  
| **do** **BLOCK** **while** (**EXPR**);   **if** (**var**) **var = const;**  
| **BLOCK**                       **STMTS**  
| ...                         }

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK**     **var = var \* var;**  
| **do** **BLOCK** **while** (**EXPR**);   **if** (**var**) **var = const;**  
| **BLOCK**                         **STMT**  
| ...                             **STMTS**  
                                  }

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK**     **var = var \* var;**  
| **do** **BLOCK** **while** (**EXPR**);   **if** (**var**) **var = const;**  
| **BLOCK**                       **while** (**var**) **BLOCK**  
| ...                           **STMTS**  
                                  }

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...



# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR;**  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** {  
| **do** **BLOCK** **while** (**EXPR**);  
| **BLOCK**  
| ...  
| }  
| }  
| }  
| }

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** {  
| **do** **BLOCK** **while** (**EXPR**);  
| **BLOCK** **STMT**  
| ... **STMTS**

**EXPR** → **var** }  
| **const** **STMTS**  
| **EXPR** + **EXPR** }  
| **EXPR** - **EXPR** }  
| **EXPR** = **EXPR** }  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** {  
| **do** **BLOCK** **while** (**EXPR**);  
| **BLOCK**  
| ...  
| **var** = **var** \* **var**;  
| **if** (**var**) **var** = **const**;  
| **while** (**var**) {  
|     **EXPR**;  
|     **STMTS**

**EXPR** → **var** }  
| **const** **STMTS**  
| **EXPR** + **EXPR** }  
| **EXPR** - **EXPR**  
| **EXPR** = **EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** {  
| **do** **BLOCK** **while** (**EXPR**);  
| **BLOCK**  
| ...  
|  
| **var** = **var** \* **var**;  
| **if** (**var**) **var** = **const**;  
| **while** (**var**) {  
|     **EXPR** = **EXPR**;  
|     **STMTS**

**EXPR** → **var** }  
| **const** **STMTS**  
| **EXPR** + **EXPR** }  
| **EXPR** - **EXPR**  
| **EXPR** = **EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK**      **var = var \* var;**  
| **do** **BLOCK** **while** (**EXPR**);    **if** (**var**) **var = const;**  
| **BLOCK**                            **while** (**var**) {  
| ...                                     **var = EXPR;**  
                                       **STMTS**

**EXPR** → **var**                                 **}**  
| **const**                                    **STMTS**  
| **EXPR + EXPR**                            **}**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** {  
| **do** **BLOCK** **while** (**EXPR**);  
| **BLOCK**  
| ...  
|

**var** = **var** \* **var**;  
**if** (**var**) **var** = **const**;  
**while** (**var**) {  
    **var** = **EXPR** + **EXPR**;  
    **STMTS**

**EXPR** → **var**  
| **const**  
| **EXPR** + **EXPR**  
| **EXPR** - **EXPR**  
| **EXPR** = **EXPR**  
| ...

}  
**STMTS**  
}

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** {  
| **do** **BLOCK** **while** (**EXPR**);  
| **BLOCK**  
| ...  
|  
| **var** = **var** \* **var**;  
| **if** (**var**) **var** = **const**;  
| **while** (**var**) {  
|  
| **var** = **var** + **EXPR**;  
| **STMTS**  
|  
| }

**EXPR** → **var**  
| **const**  
| **EXPR** + **EXPR**  
| **EXPR** - **EXPR**  
| **EXPR** = **EXPR**  
| ...  
|  
| }

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** {  
| **do** **BLOCK** **while** (**EXPR**);  
| **BLOCK**  
| ...  
| **STMTS**

**EXPR** → **var**  
| **const**  
| **EXPR** + **EXPR**  
| **EXPR** - **EXPR**  
| **EXPR** = **EXPR**  
| ...



# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** {  
| **do** **BLOCK** **while** (**EXPR**);  
| **BLOCK**  
| ...  
| }  
| }  
| **var** = **var** \* **var**;  
| **if** (**var**) **var** = **const**;  
| **while** (**var**) {  
|     **var** = **var** + **const**;  
| }  
| **STMTS**

**EXPR** → **var**  
| **const**  
| **EXPR** + **EXPR**  
| **EXPR** - **EXPR**  
| **EXPR** = **EXPR**  
| ...

# CFGs for Programming Languages

**BLOCK** → **STMT**  
| { **STMTS** }

**STMTS** →  $\epsilon$   
| **STMT STMTS**

**STMT** → **EXPR**;  
| **if** (**EXPR**) **BLOCK** {  
| **while** (**EXPR**) **BLOCK** {  
| **do** **BLOCK** **while** (**EXPR**);  
| **BLOCK**  
| ...  
| }  
| }

**EXPR** → **var**  
| **const**  
| **EXPR + EXPR**  
| **EXPR - EXPR**  
| **EXPR = EXPR**  
| ...

# Context-Free Languages

- A language  $L$  is called a **context-free language** (or CFL) iff there is a CFG  $G$  such that  $L = \mathcal{L}(G)$ .
- Questions:
  - What languages are context-free?
  - How are context-free and regular languages related?

# From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

**S**  $\rightarrow$  **a\*b**

# From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

**S**  $\rightarrow$  **A****b**

# From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

$$\begin{aligned} S &\rightarrow Ab \\ A &\rightarrow Aa \mid \epsilon \end{aligned}$$

# From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

**S**  $\rightarrow$  **a (b | c\*)**

# From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

**S**  $\rightarrow$  **aX**

**X**  $\rightarrow$  **(b | c\*)**



# From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

**S** → **aX**

**X** → **b** | **c\***

# From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

**S** → **a****X**

**X** → **b** | **C**

# From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

**S** → **aX**

**X** → **b** | **C**

**C** → **Cc** | **ε**

# Regular Languages and CFLs

- ***Theorem:*** Every regular language is context-free.
- ***Proof Idea:*** Use the construction from the previous slides to convert a regular expression for  $L$  into a CFG for  $L$ . ■

# The Language of a Grammar

- Consider the following CFG  $G$ :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this generate?

a	a	a	a	a	a	S	b	b	b	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---

# The Language of a Grammar

- Consider the following CFG  $G$ :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this generate?

a	a	a	a	a	a	b	b	b	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---

# The Language of a Grammar

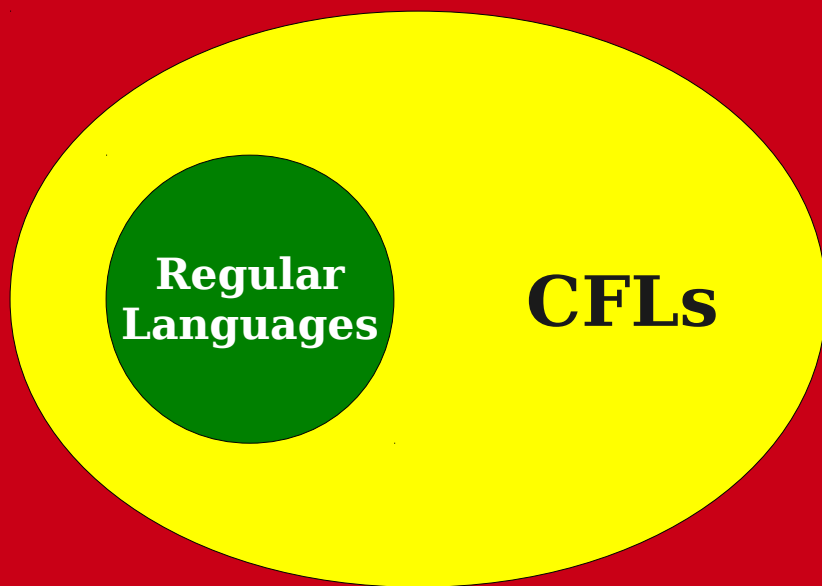
- Consider the following CFG  $G$ :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this generate?

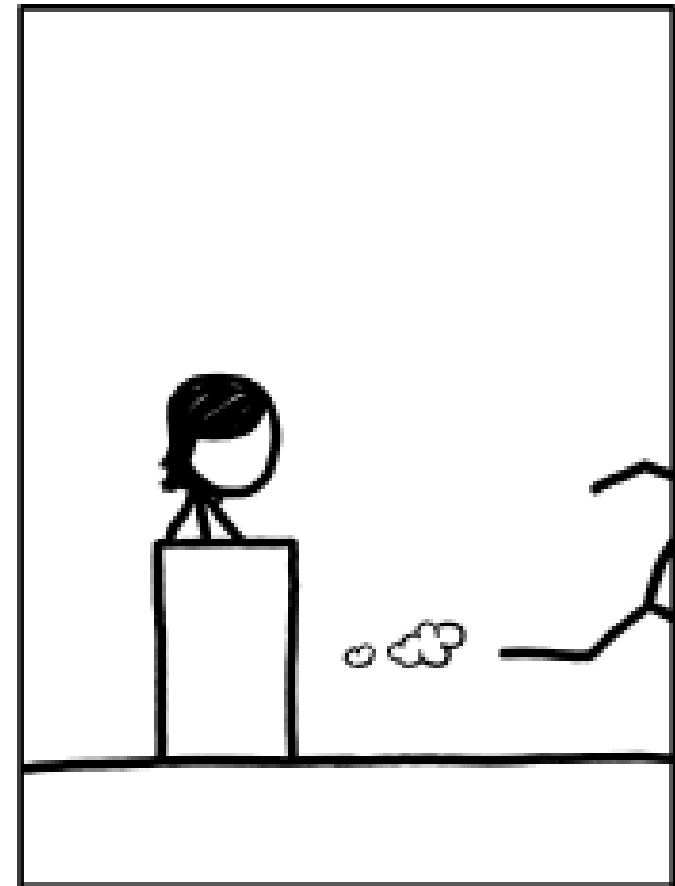
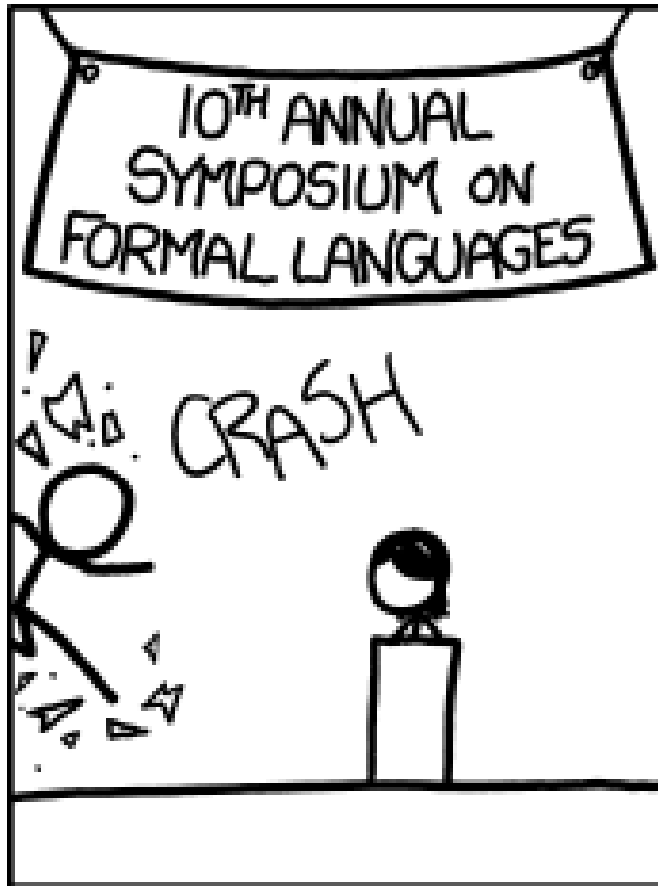
a	a	a	a	a	a	b	b	b	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---

$$\mathcal{L}(G) = \{ a^n b^n \mid n \in \mathbb{N} \}$$



**All Languages**





<http://xkcd.com/1090/>

# Designing CFGs

- Like designing DFAs, NFAs, and regular expressions, designing CFGs is a craft.
- When thinking about CFGs:
  - **Think recursively:** Build up bigger structures from smaller ones.
  - **Have a construction plan:** Know in what order you will build up the string.

# Designing CFGs

- Let  $\Sigma = \{a, b\}$  and let  $L = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}$
- We can design a CFG for  $L$  by thinking inductively:
  - Base case:  $\epsilon$ ,  $a$ , and  $b$  are palindromes.
  - If  $w$  is a palindrome, then  $awa$  and  $bwb$  are palindromes.

$$S \rightarrow \epsilon \mid a \mid b \mid aSa \mid bSb$$

# Designing CFGs

- Let  $\Sigma = \{ (, ) \}$  and let  $L = \{ w \in \Sigma^* \mid w \text{ is a string of balanced parentheses} \}$
- We can think about how we will build strings in this language as follows:
  - The empty string is balanced.
  - Any two strings of balanced parentheses can be concatenated.
  - Any string of balanced parentheses can be parenthesized.

$$S \rightarrow SS \mid (S) \mid \epsilon$$

# Designing CFGs: Watch Out!

- When designing CFGs, remember that each nonterminal can be expanded out independently of the others.
- Let  $\Sigma = \{a, \overset{?}{=}\}$  and let  $L = \{a^n \overset{?}{=} a^n \mid n \in \mathbb{N}\}$ . Is the following a CFG for  $L$ ?

- $S \rightarrow X \overset{?}{=} X$

- $X \rightarrow aX \mid \varepsilon$

$$\begin{aligned} &S \\ \Rightarrow &X \overset{?}{=} X \\ \Rightarrow &aX \overset{?}{=} X \\ \Rightarrow &aaX \overset{?}{=} X \\ \Rightarrow &aa \overset{?}{=} X \\ \Rightarrow &aa \overset{?}{=} aX \\ \Rightarrow &aa \overset{?}{=} a \end{aligned}$$

# Finding a Build Order

- Let  $\Sigma = \{a, \underline{a}\}$  and let  $L = \{a^n \underline{a} a^n \mid n \in \mathbb{N}\}$ .
- To build a CFG for  $L$ , we need to be more clever with how we construct the string.
- **Idea:** Build from the ends inward.
- Gives this grammar:  $S \rightarrow aSa \mid \underline{a}$

$S$   
 $\Rightarrow aSa$   
 $\Rightarrow aaSaa$   
 $\Rightarrow aaaSaaa$   
 $\Rightarrow aaa \underline{a} aaa$

# Designing CFGs: A Caveat

- Let  $\Sigma = \{\mathbf{l}, \mathbf{r}\}$  and let  $L = \{w \in \Sigma^* \mid w \text{ has the same number of } \mathbf{l}'\text{'s and } \mathbf{r}'\text{'s}\}$
- Is this a grammar for  $L$ ?

$$\mathbf{S} \rightarrow \mathbf{lSr} \mid \mathbf{rSl} \mid \epsilon$$

- Can you derive the string  $\mathbf{lrrl}$ ?

# Designing CFGs: A Caveat

- When designing a CFG for a language, make sure that it
  - generates all the strings in the language and
  - never generates a string outside the language.
- The first of these can be tricky – make sure to test your grammars!
- You'll design your own CFG for this language on the next problem set.



# CFG Caveats II

- Is the following grammar a CFG for the language  $\{ \mathbf{a}^n \mathbf{b}^n \mid n \in \mathbb{N} \}$ ?

$$\mathbf{S} \rightarrow \mathbf{aSb}$$

- What strings can you derive?
  - Answer: **None!**
- What is the language of the grammar?
  - Answer:  $\emptyset$
- When designing CFGs, make sure your recursion actually terminates!

# Parse Trees

# Parse Trees

**E**

$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E} \mathbf{Op} \mathbf{E} \mid \text{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

**E**

**E**

$$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E} \mathbf{Op} \mathbf{E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$$

# Parse Trees

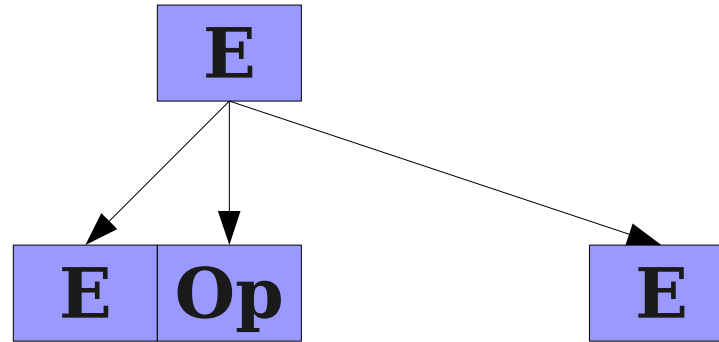
**E**

**E**  
 $\Rightarrow$  **E Op E**

**E**  $\rightarrow$  **E Op E** | **int** | (**E**)  
**Op**  $\rightarrow$  + | \* | - | /

# Parse Trees

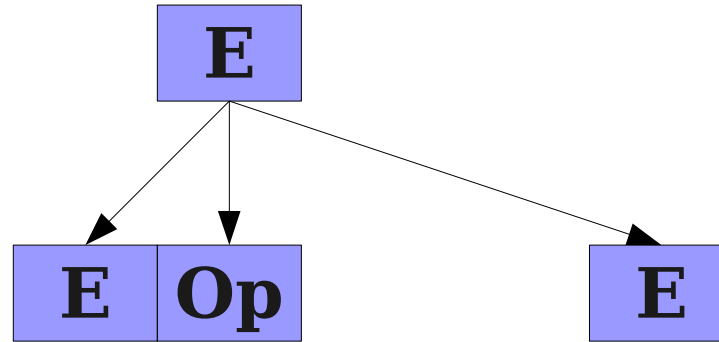
**E**  
 $\Rightarrow$  **E Op E**



$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E Op E} \mid \text{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
--------------------------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

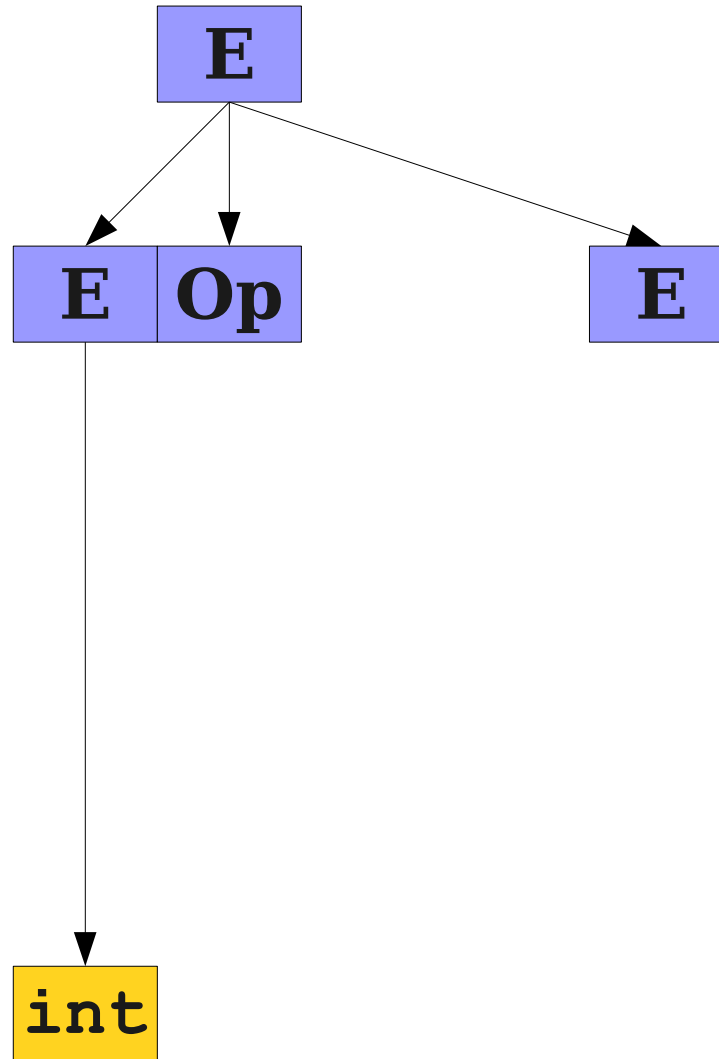
**E**  
⇒ **E Op E**  
⇒ **int Op E**



$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E\ Op\ E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

**E**  
 $\Rightarrow$  **E Op E**  
 $\Rightarrow$  **int Op E**

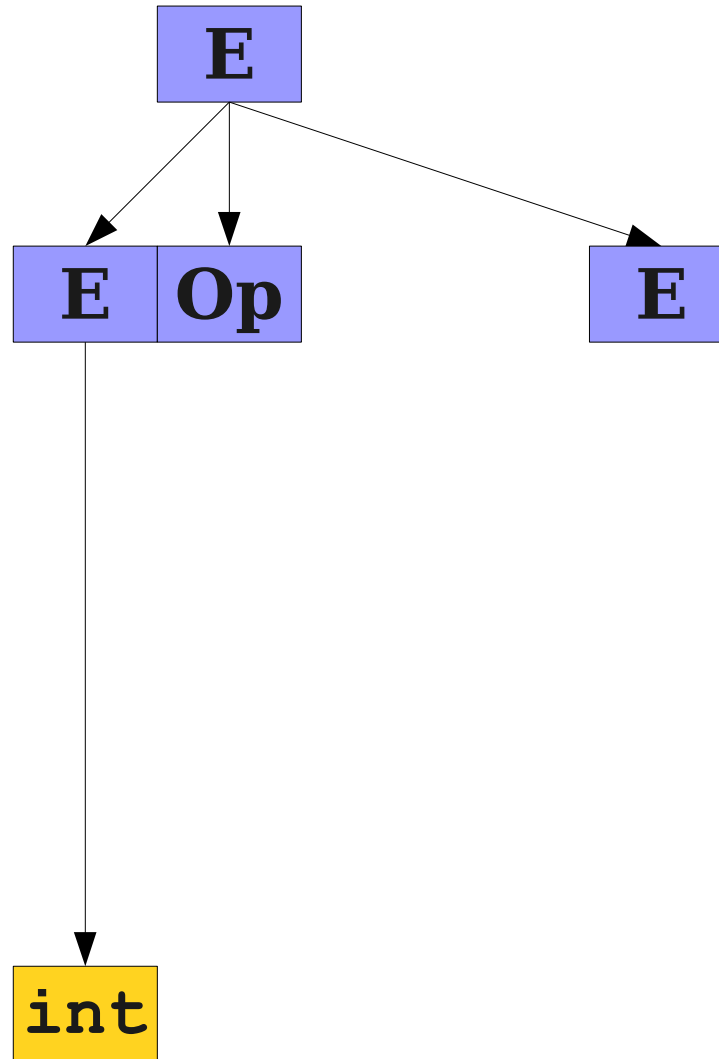


$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E\ Op\ E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------------------------------



# Parse Trees

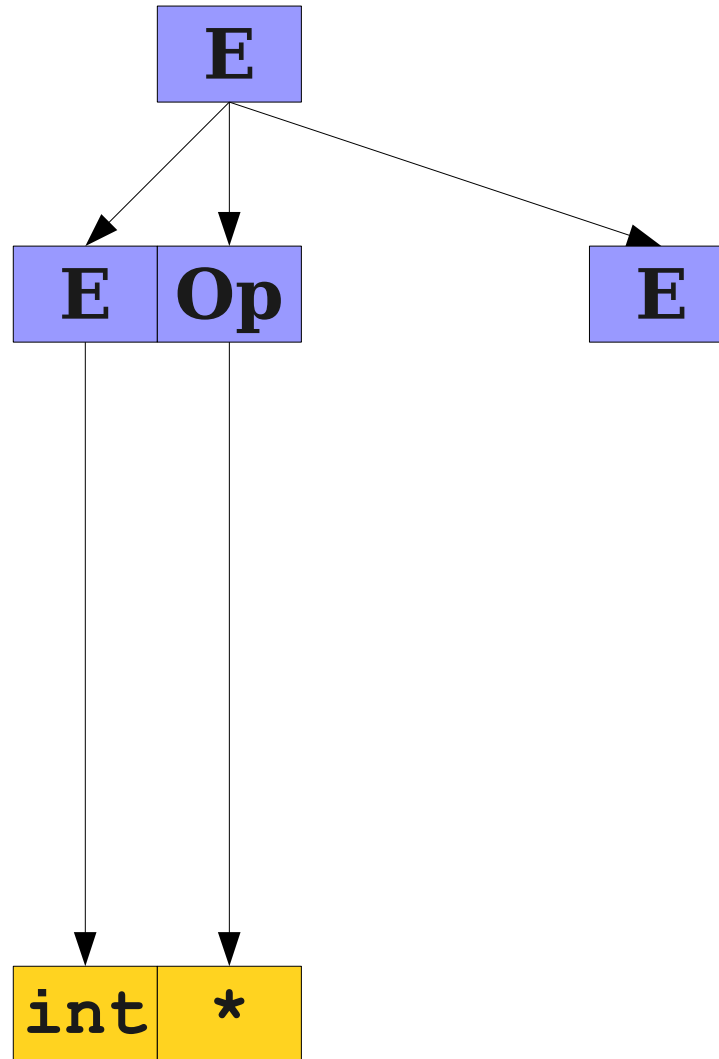
**E**  
⇒ **E Op E**  
⇒ **int Op E**  
⇒ **int \* E**



$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E\ Op\ E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

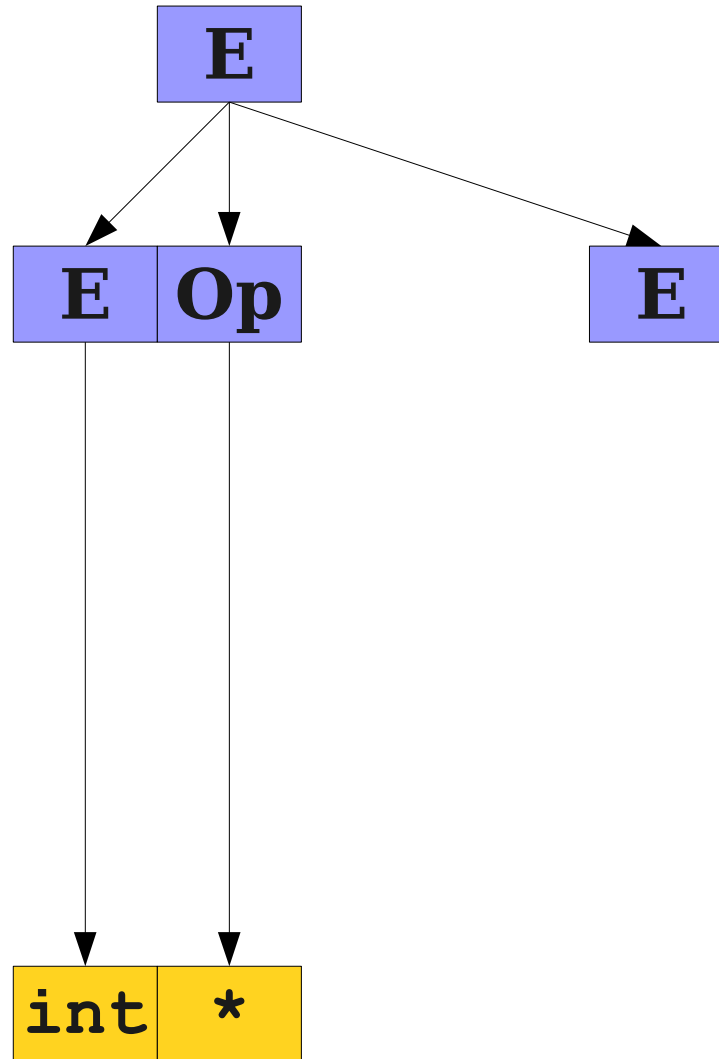
**E**  
⇒ **E Op E**  
⇒ **int Op E**  
⇒ **int \* E**



$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E Op E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
----------------------------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

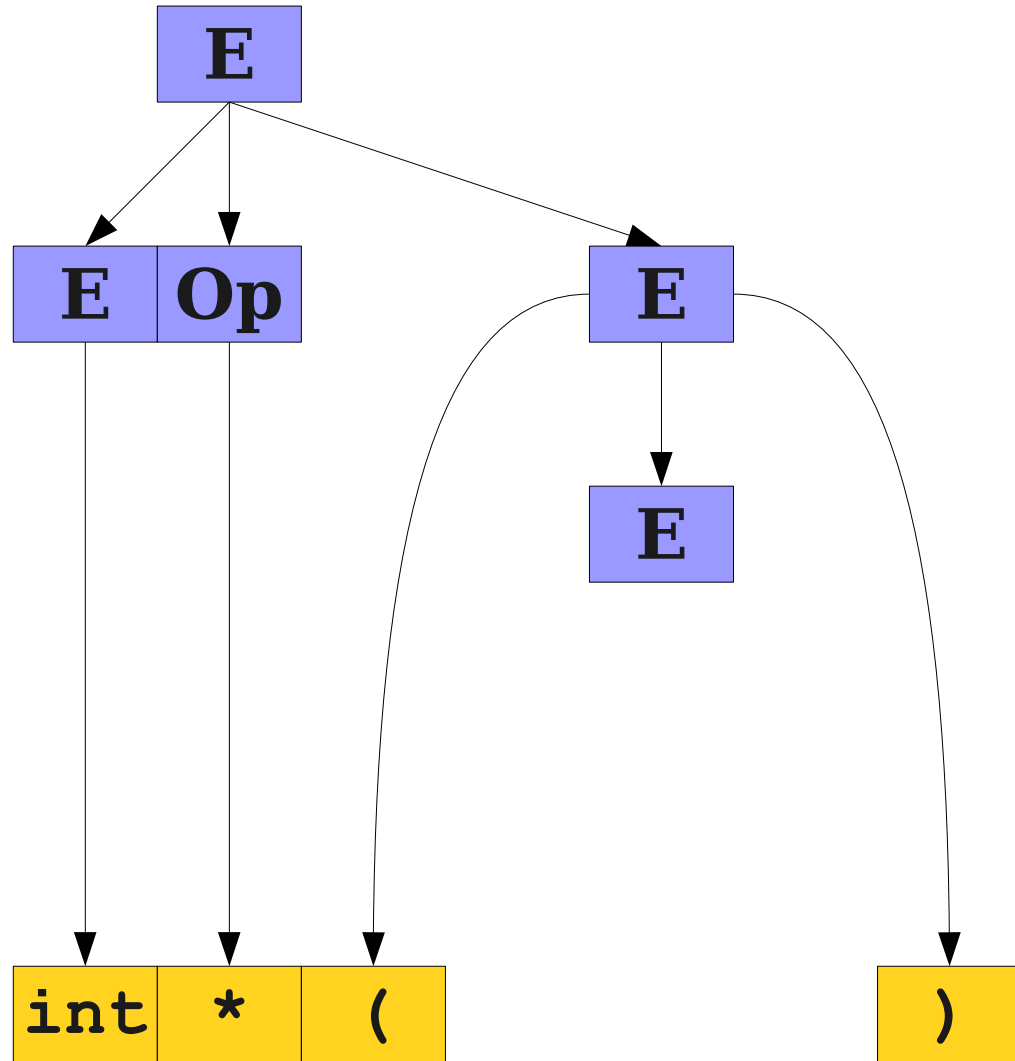
**E**  
⇒ **E Op E**  
⇒ **int Op E**  
⇒ **int \* E**  
⇒ **int \* (E)**



**E** → **E Op E** | **int** | (**E**)  
**Op** → **+** | **\*** | **-** | **/**

# Parse Trees

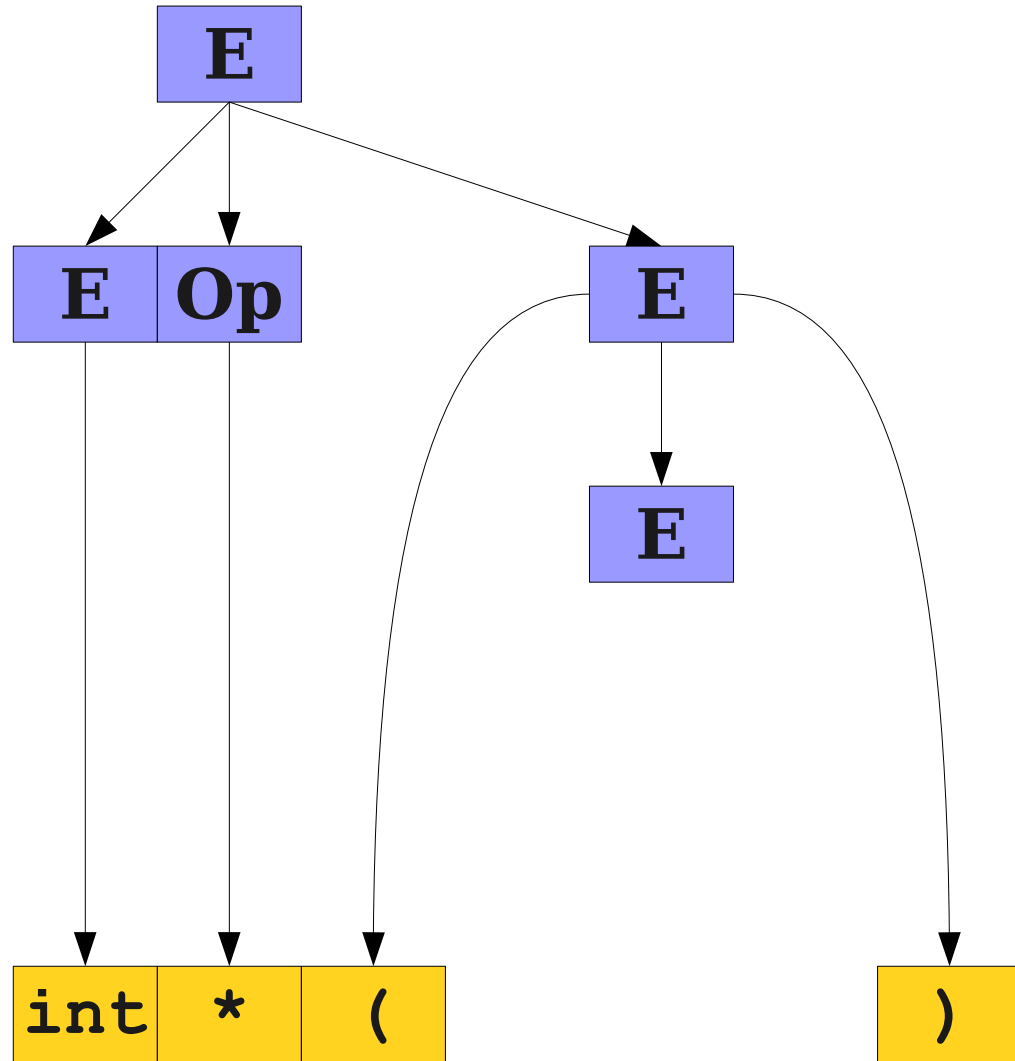
**E**  
⇒ **E Op E**  
⇒ **int Op E**  
⇒ **int \* E**  
⇒ **int \* (E)**



**E** → **E Op E** | **int** | (**E**)  
**Op** → + | \* | - | /

# Parse Trees

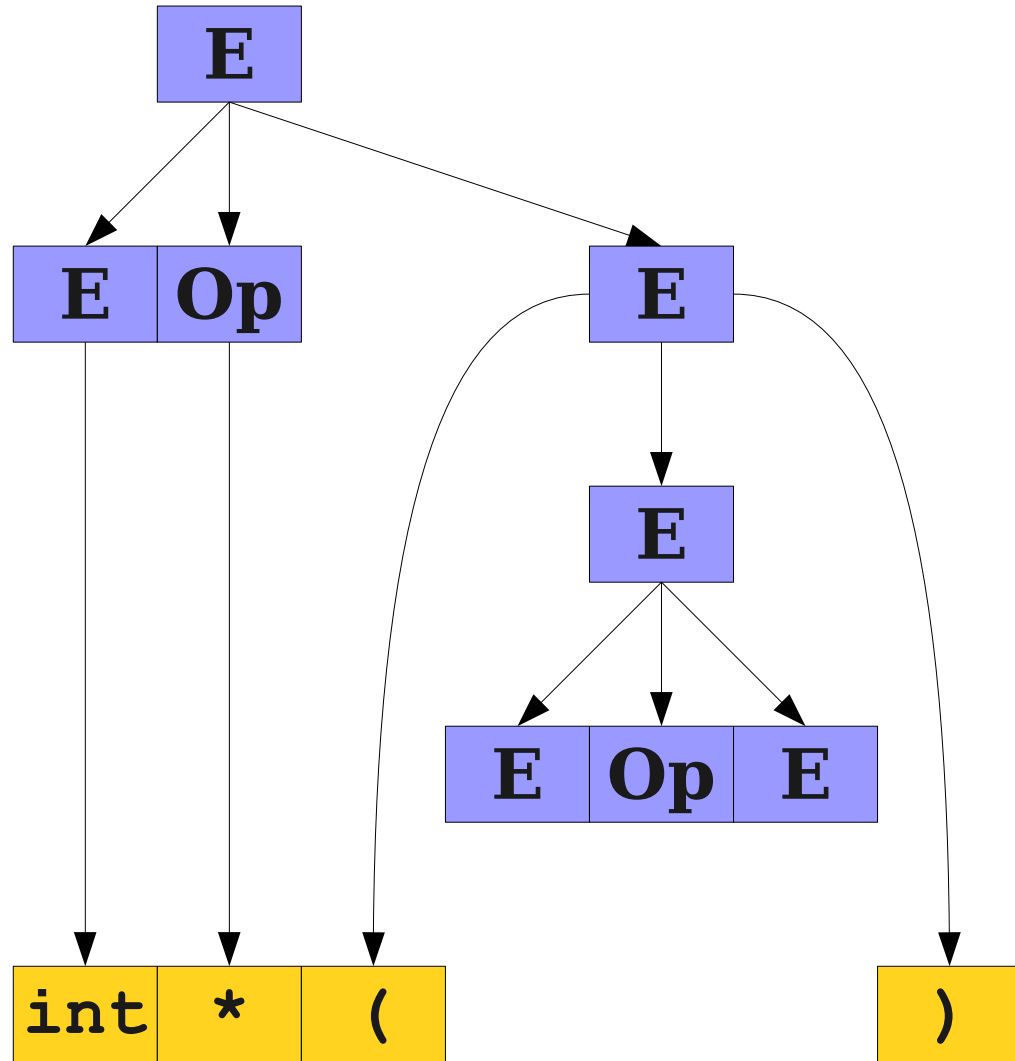
**E**  
⇒ **E Op E**  
⇒ **int Op E**  
⇒ **int \* E**  
⇒ **int \* (E)**  
⇒ **int \* (E Op E)**



**E** → **E Op E** | **int** | **(E)**  
**Op** → **+** | **\*** | **-** | **/**

# Parse Trees

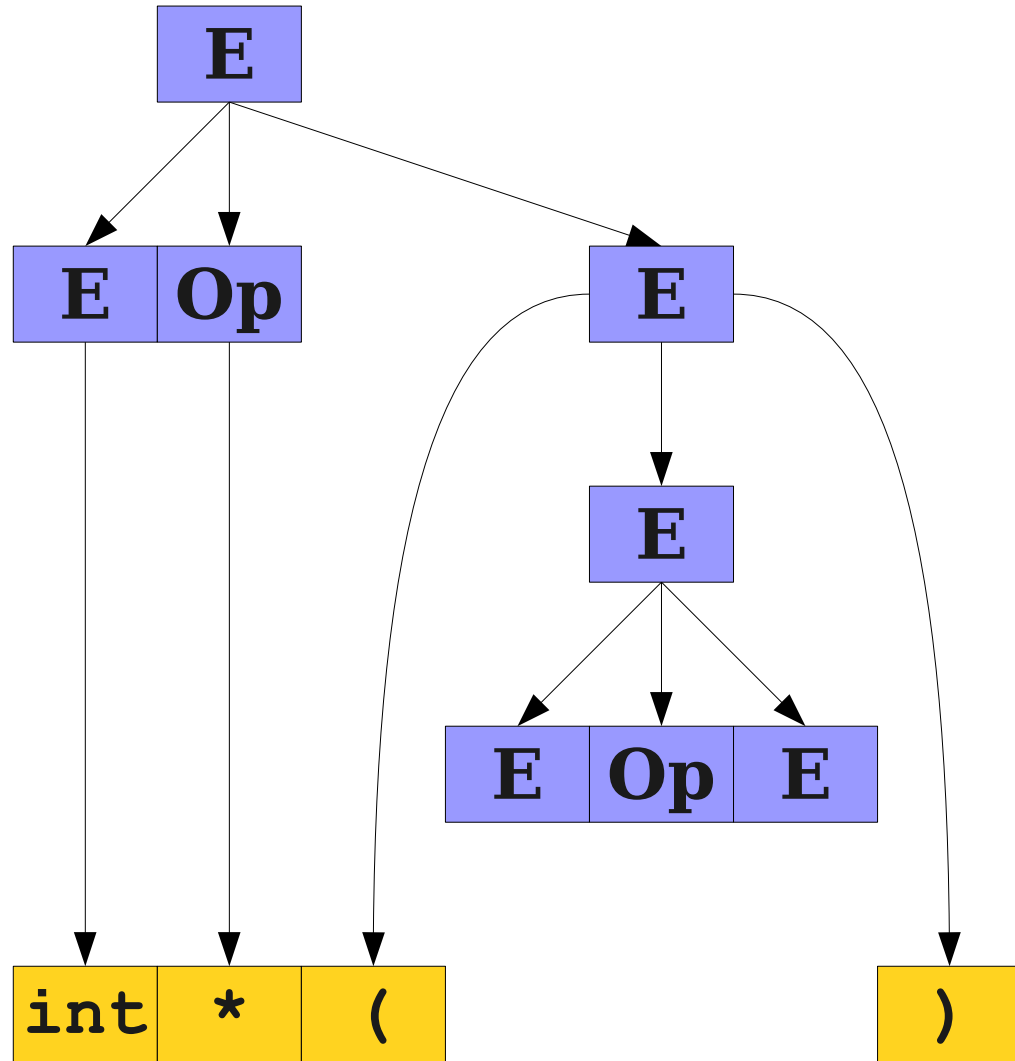
**E**  
⇒ **E Op E**  
⇒ **int Op E**  
⇒ **int \* E**  
⇒ **int \* (E)**  
⇒ **int \* (E Op E)**



**E** → **E Op E** | **int** | **(E)**  
**Op** → **+** | **\*** | **-** | **/**

# Parse Trees

**E**  
⇒ **E Op E**  
⇒ **int Op E**  
⇒ **int \* E**  
⇒ **int \* (E)**  
⇒ **int \* (E Op E)**  
⇒ **int \* (int Op E)**



$\mathbf{E} \rightarrow \mathbf{E\ Op\ E} \mid \mathbf{int} \mid (\mathbf{E})$
$\mathbf{Op} \rightarrow + \mid * \mid - \mid /$

# Parse Trees

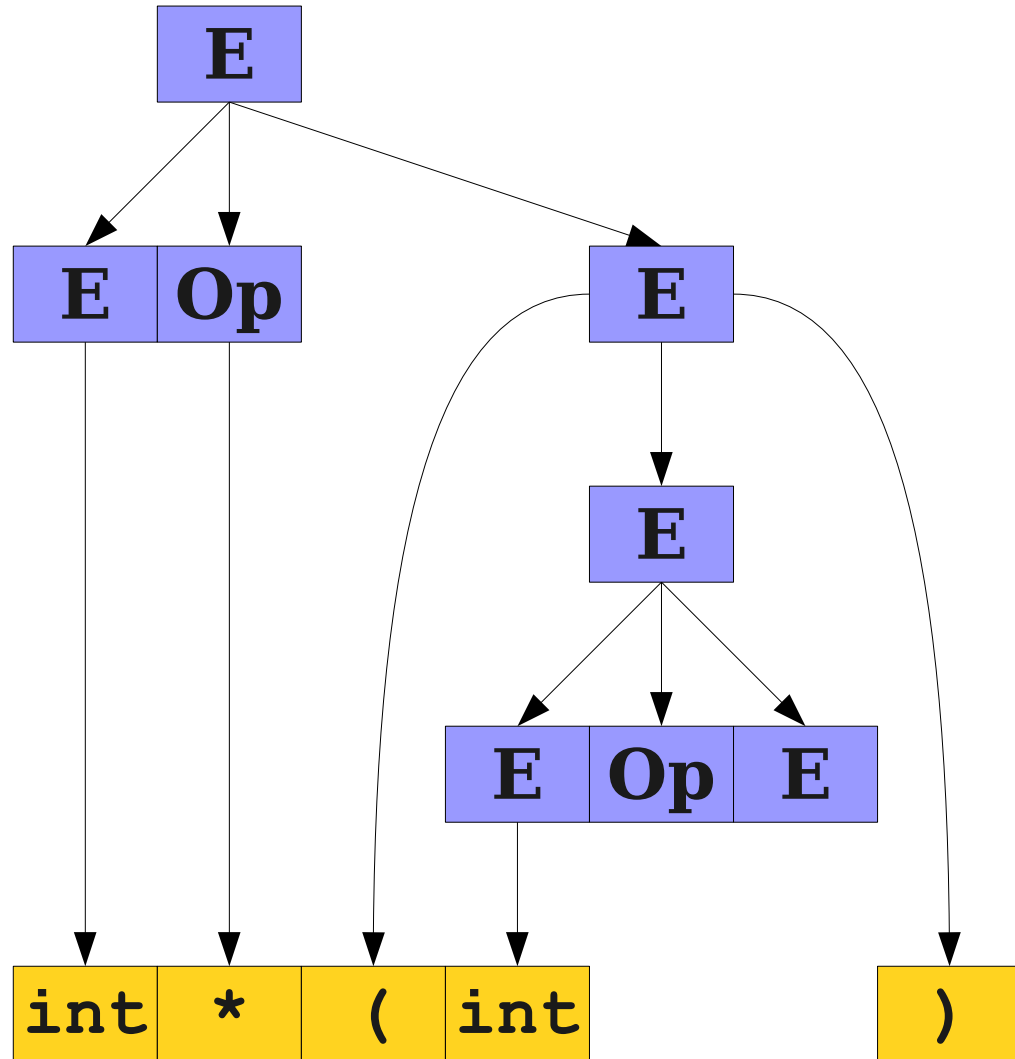
# E

$$\Rightarrow \mathbf{E} \mathbf{O_p} \mathbf{E}$$

⇒ **int Op E**

$\Rightarrow \text{int} * \mathbf{E}$

⇒ int \* (E)

$$\Rightarrow \text{int} * (\text{E Op E})$$
$$\Rightarrow \text{int} * (\text{int Op E})$$

$$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E} \mathbf{Op} \mathbf{E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$$



# Parse Trees

# E

**⇒ E Op E**

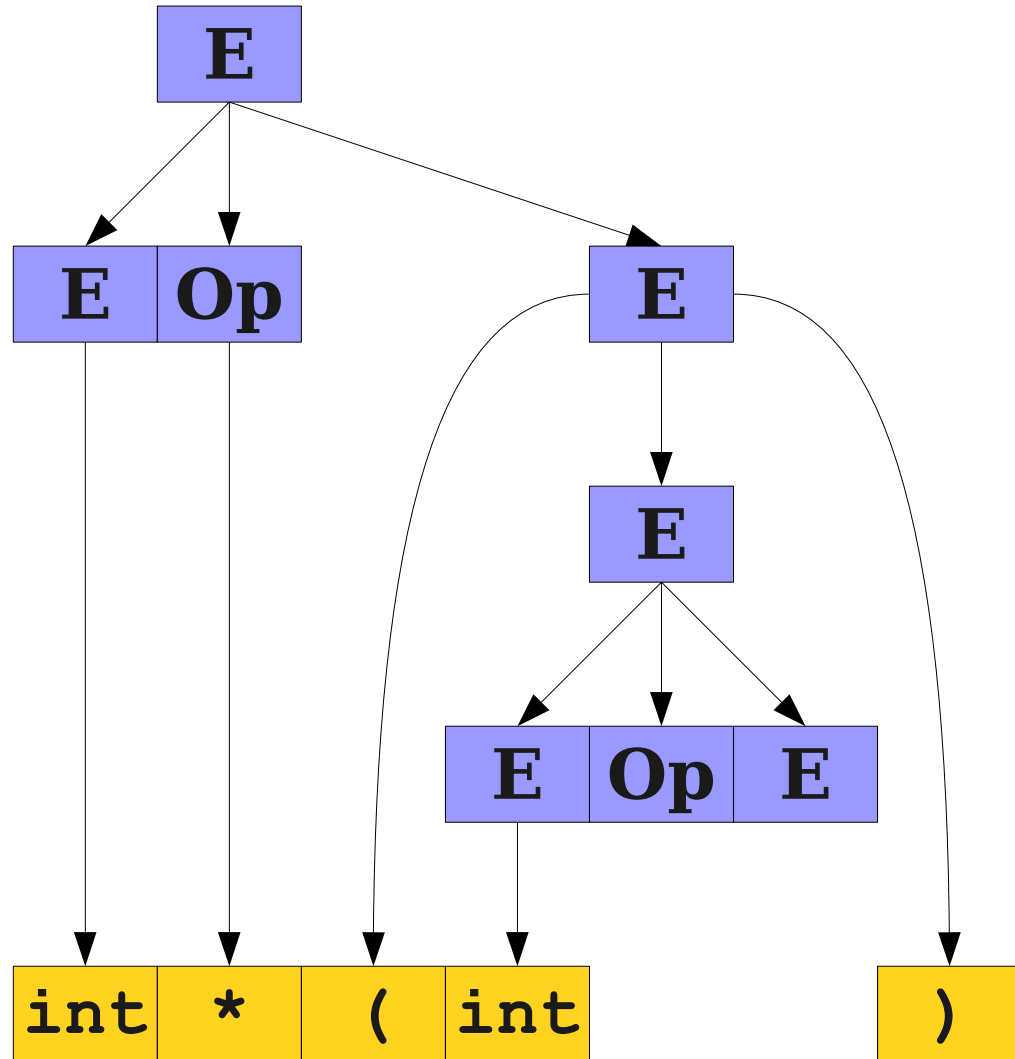
$\Rightarrow$  **int Op E**

$\Rightarrow \text{int} * \mathbf{E}$

⇒ int \* (E)

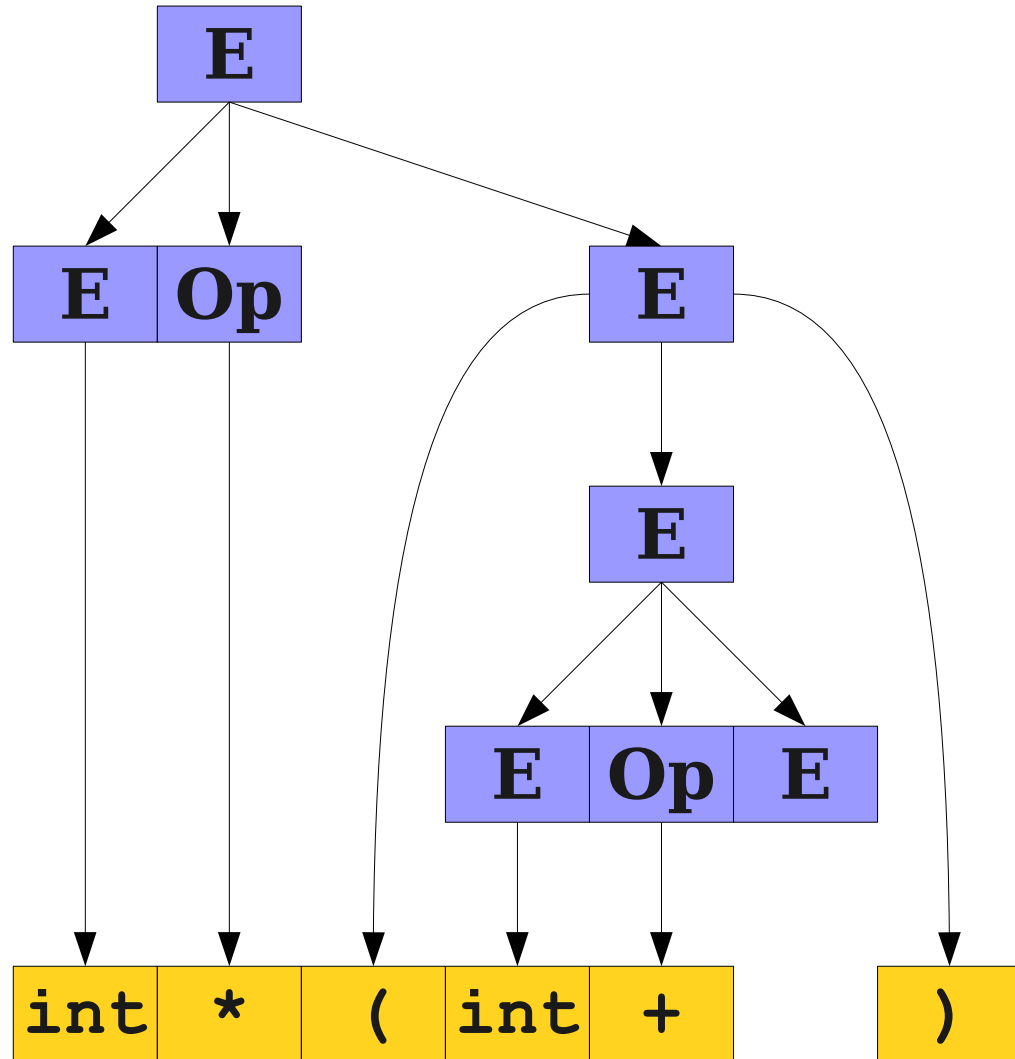
$$\Rightarrow \text{int} * (\text{E Op E})$$
$$\Rightarrow \text{int} * (\text{int Op E})$$

⇒ int \* (int + E)


$$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E} \mathbf{Op} \mathbf{E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$$

# Parse Trees

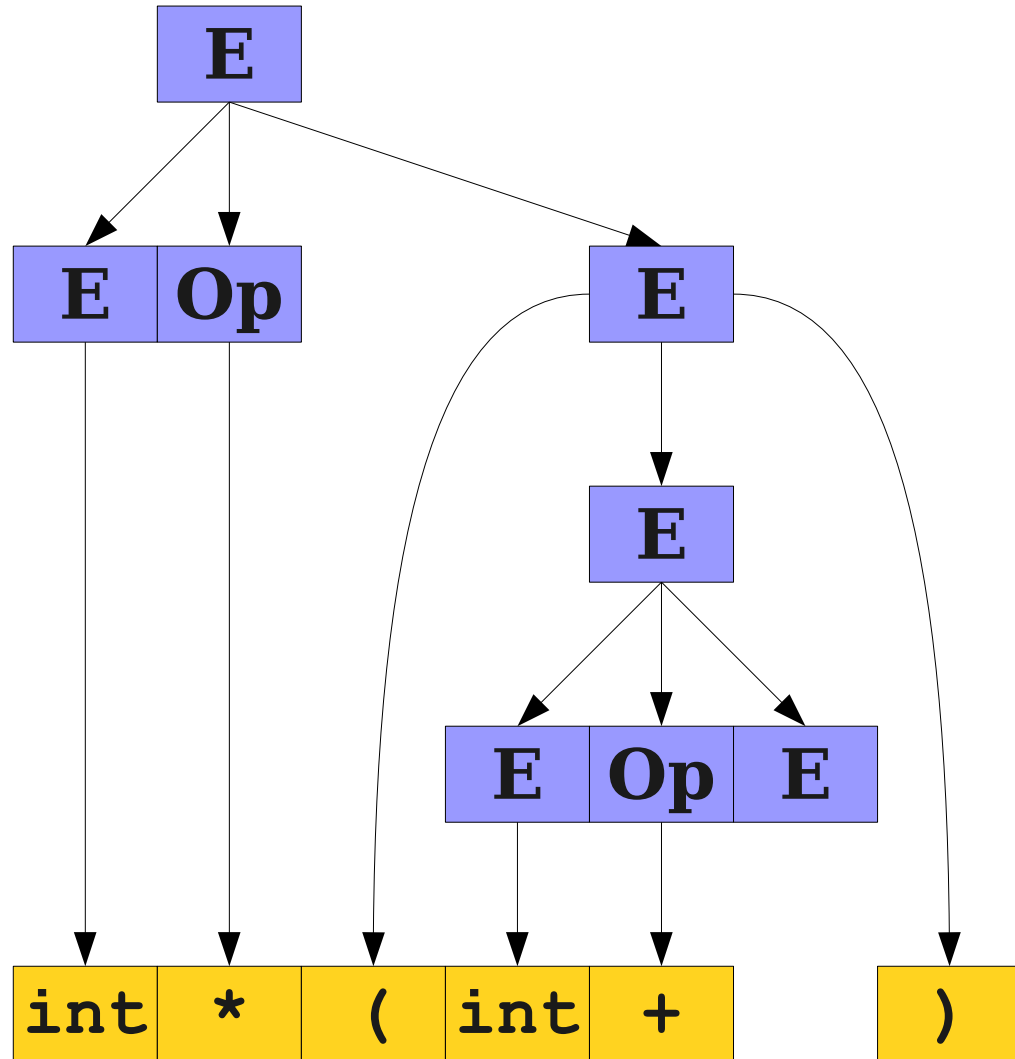
**E**  
⇒ **E Op E**  
⇒ int **Op E**  
⇒ int \* **E**  
⇒ int \* (**E**)  
⇒ int \* (**E Op E**)  
⇒ int \* (int **Op E**)  
⇒ int \* (int + **E**)



**E** → **E Op E** | int | (**E**)  
**Op** → + | \* | - | /

# Parse Trees

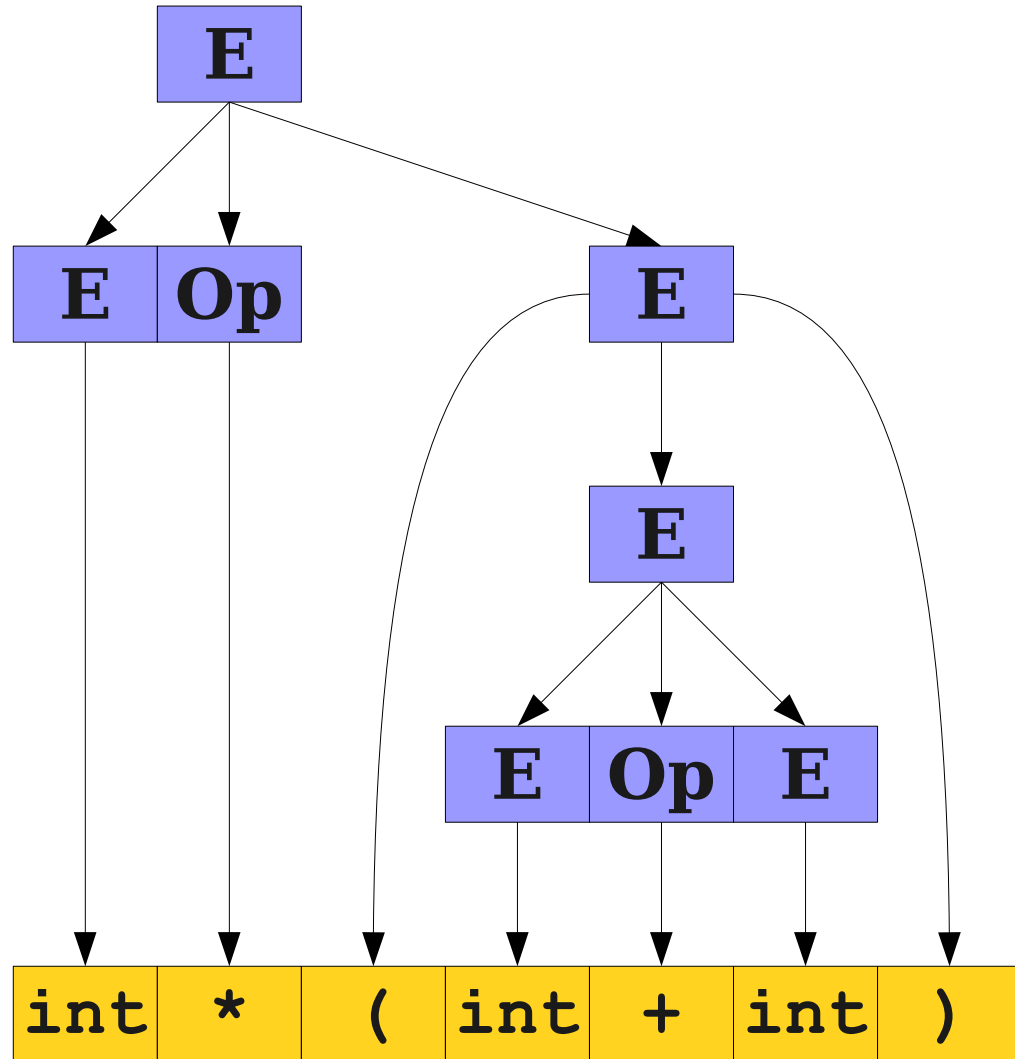
**E**  
⇒ **E Op E**  
⇒ int **Op E**  
⇒ int \* **E**  
⇒ int \* (**E**)  
⇒ int \* (**E Op E**)  
⇒ int \* (int **Op E**)  
⇒ int \* (int + **E**)  
⇒ int \* (int + int)



**E** → **E Op E** | int | (**E**)  
**Op** → + | \* | - | /

# Parse Trees

**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)



**E** → **E Op E** | int | (**E**)  
**Op** → + | \* | - | /

# Parse Trees

**E**

$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E} \mathbf{Op} \mathbf{E} \mid \text{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

**E**

**E**

$$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E} \mathbf{Op} \mathbf{E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$$

# Parse Trees

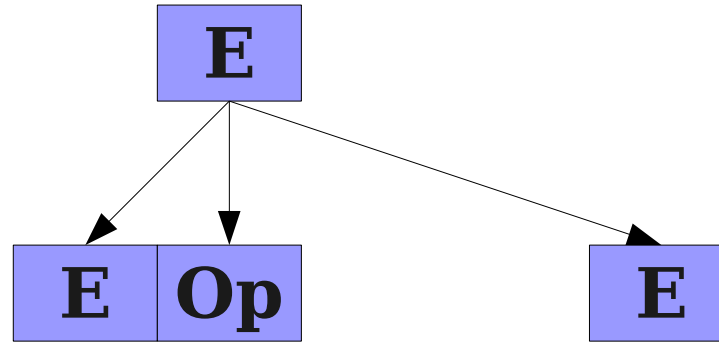
**E**

**E**  
 $\Rightarrow$  **E Op E**

**E**  $\rightarrow$  **E Op E** | **int** | (**E**)  
**Op**  $\rightarrow$  + | \* | - | /

# Parse Trees

**E**  
 $\Rightarrow$  **E Op E**

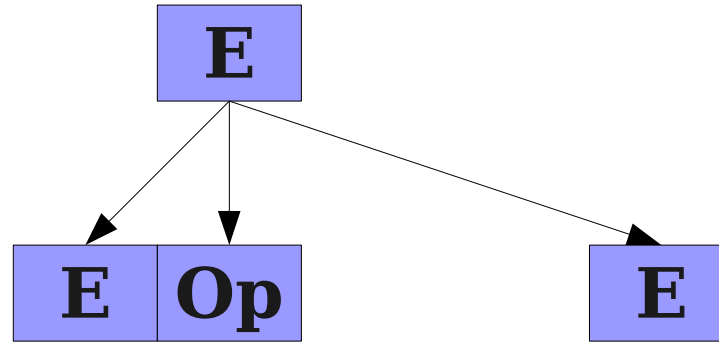


**E**  $\rightarrow$  **E Op E** | **int** | (**E**)  
**Op**  $\rightarrow$  + | \* | - | /



# Parse Trees

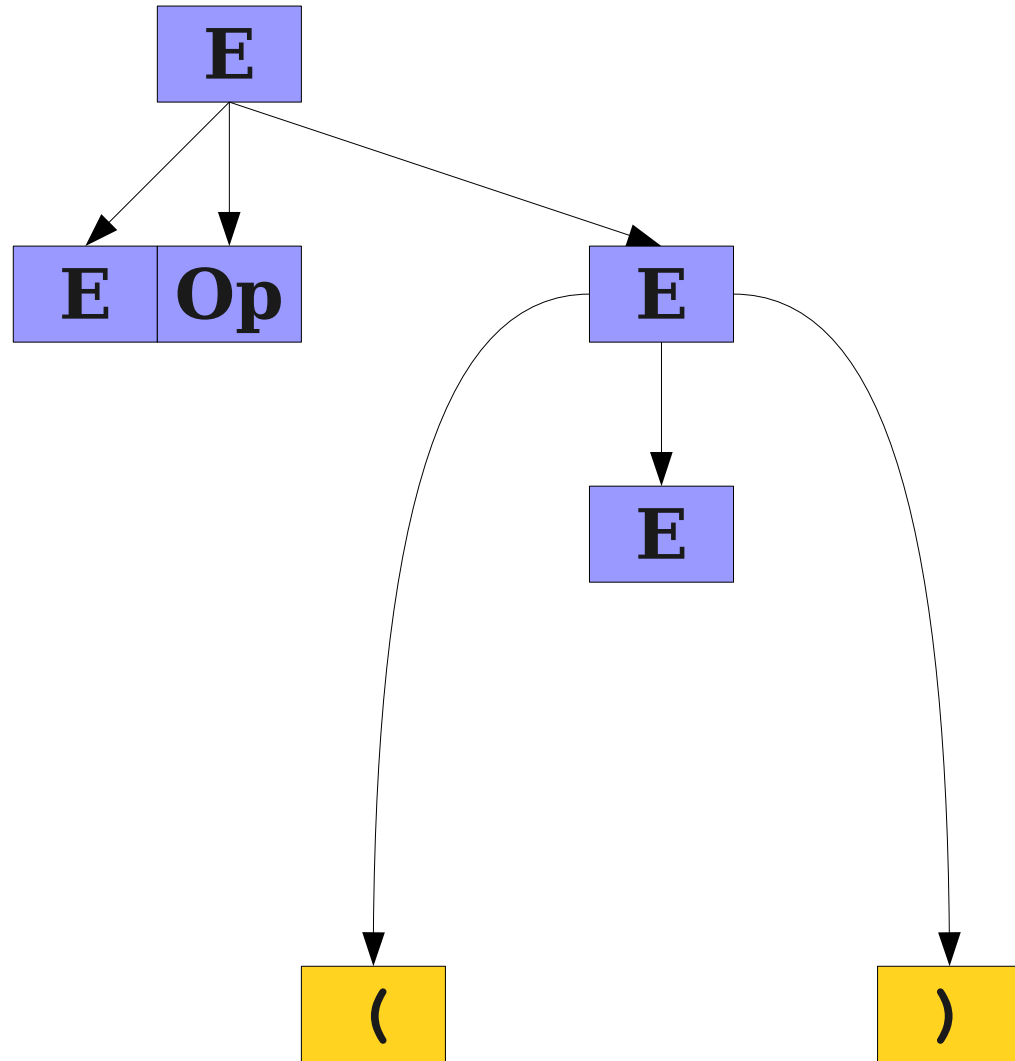
**E**  
 $\Rightarrow$  **E Op E**  
 $\Rightarrow$  **E Op (E)**



$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E Op E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
----------------------------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

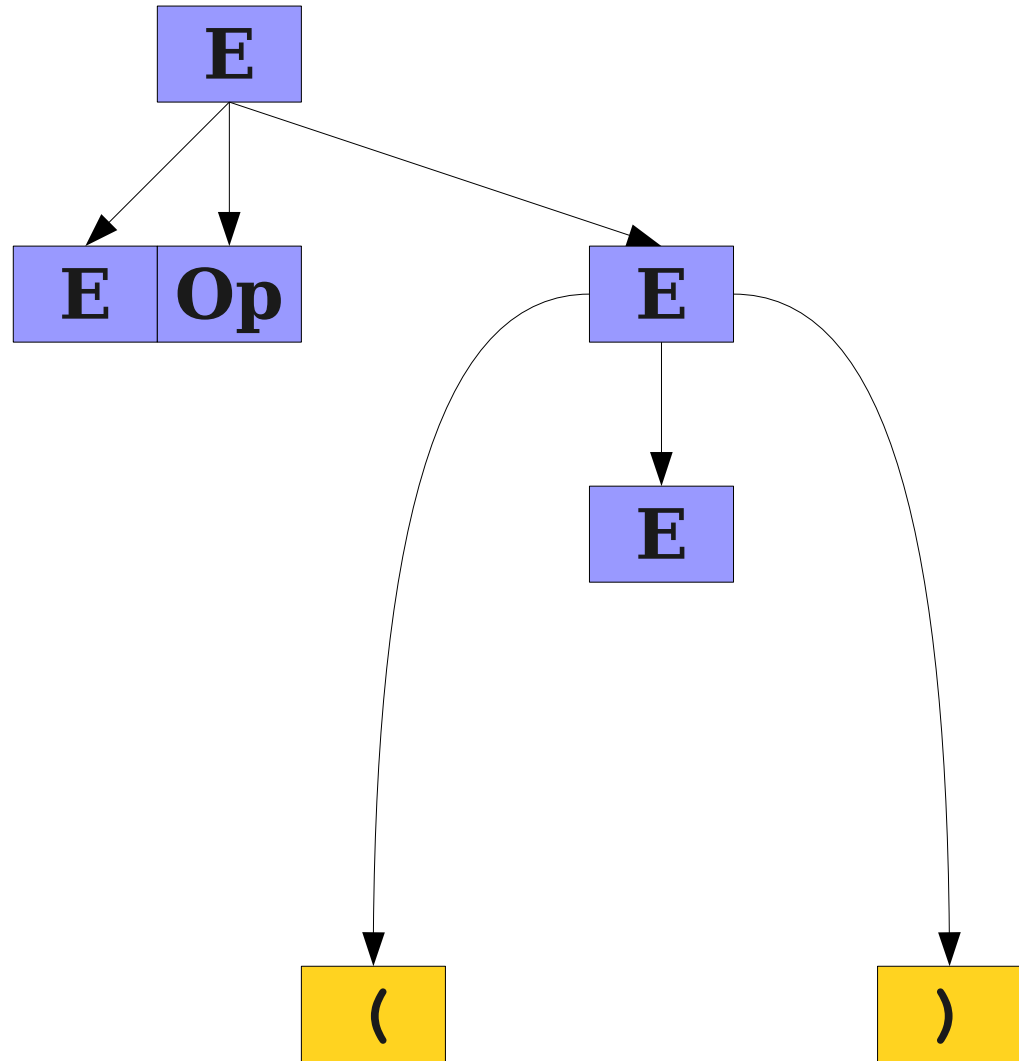
**E**  
 $\Rightarrow$  **E Op E**  
 $\Rightarrow$  **E Op (E)**



$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E\ Op\ E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

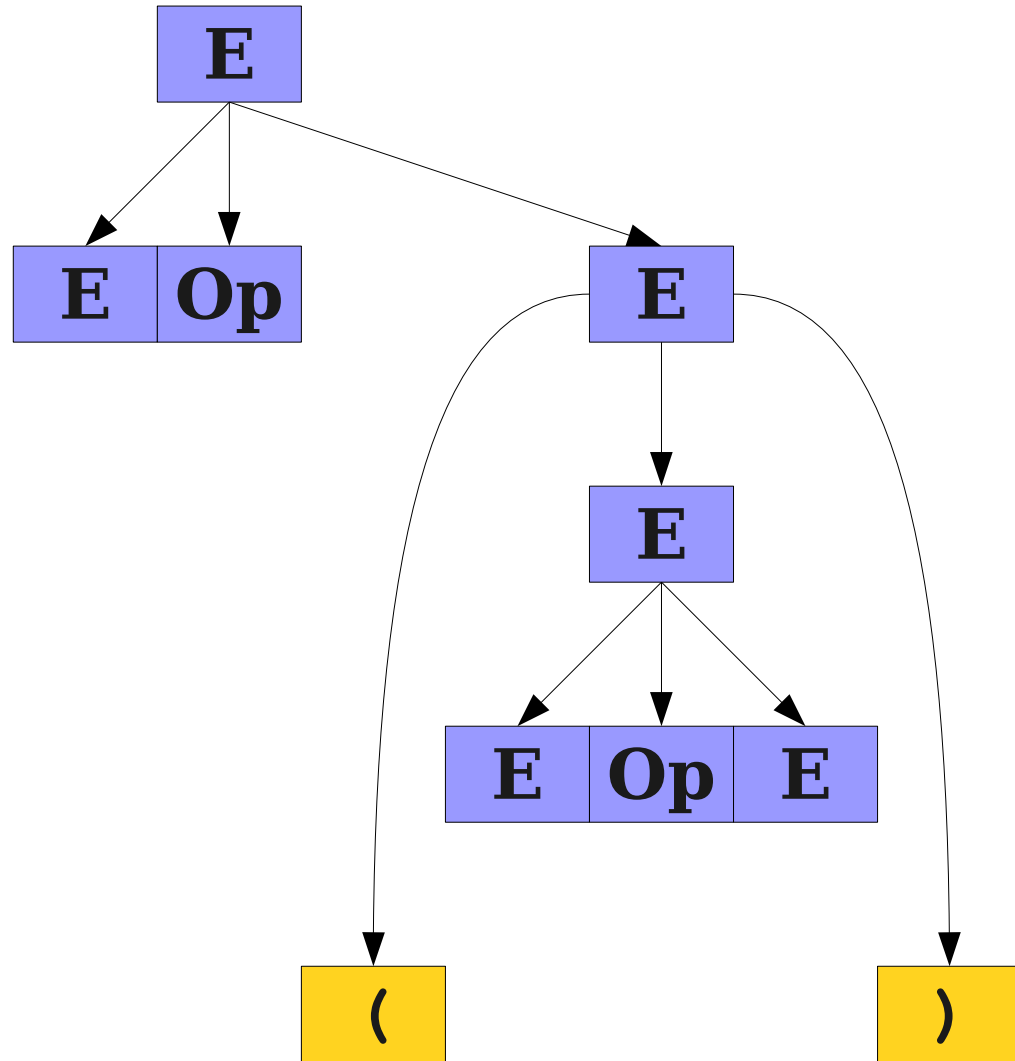
**E**  
⇒ **E Op E**  
⇒ **E Op (E)**  
⇒ **E Op (E Op E)**



$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E\ Op\ E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

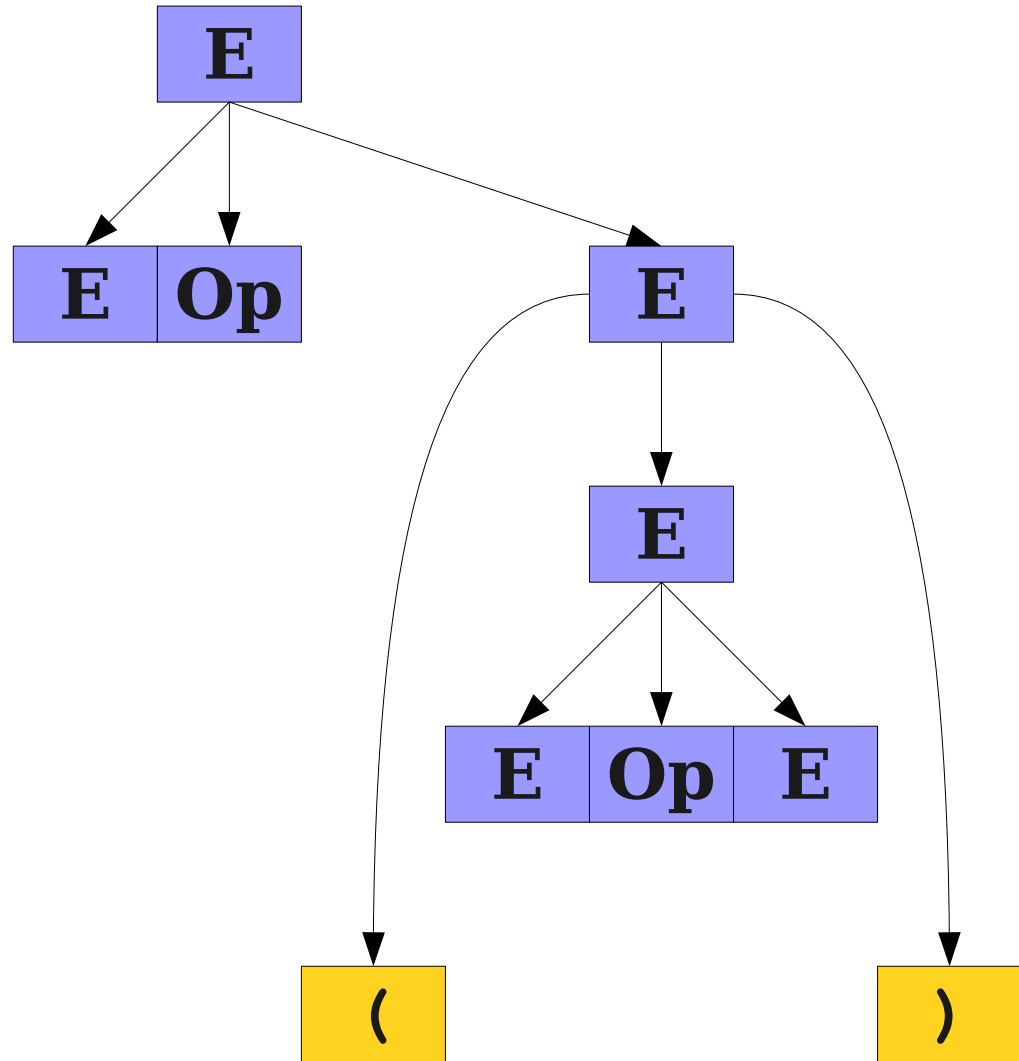
**E**  
 $\Rightarrow$  **E Op E**  
 $\Rightarrow$  **E Op (E)**  
 $\Rightarrow$  **E Op (E Op E)**



$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E Op E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
----------------------------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

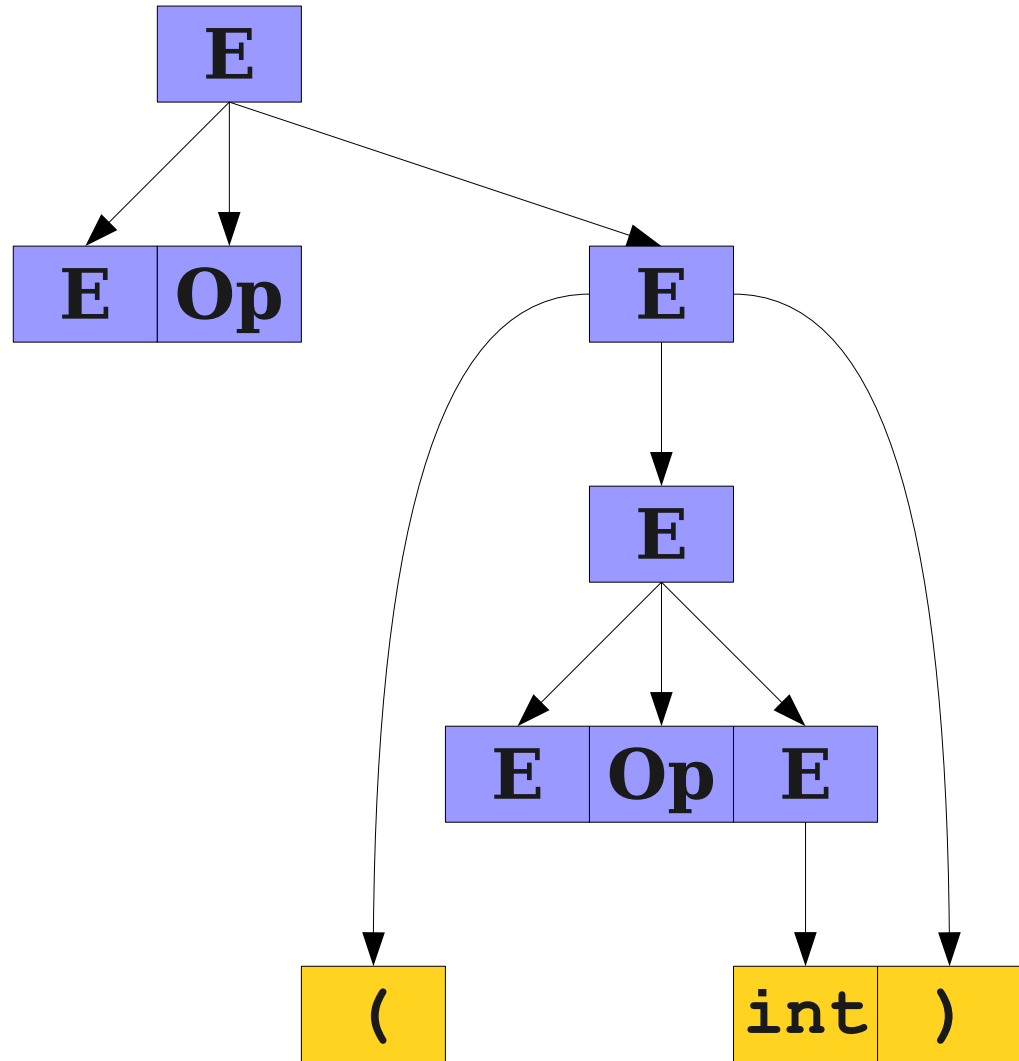
**E**  
⇒ **E Op E**  
⇒ **E Op (E)**  
⇒ **E Op (E Op E)**  
⇒ **E Op (E Op int)**



$\begin{aligned} E &\rightarrow E \text{ Op } E \mid \text{int} \mid (E) \\ \text{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

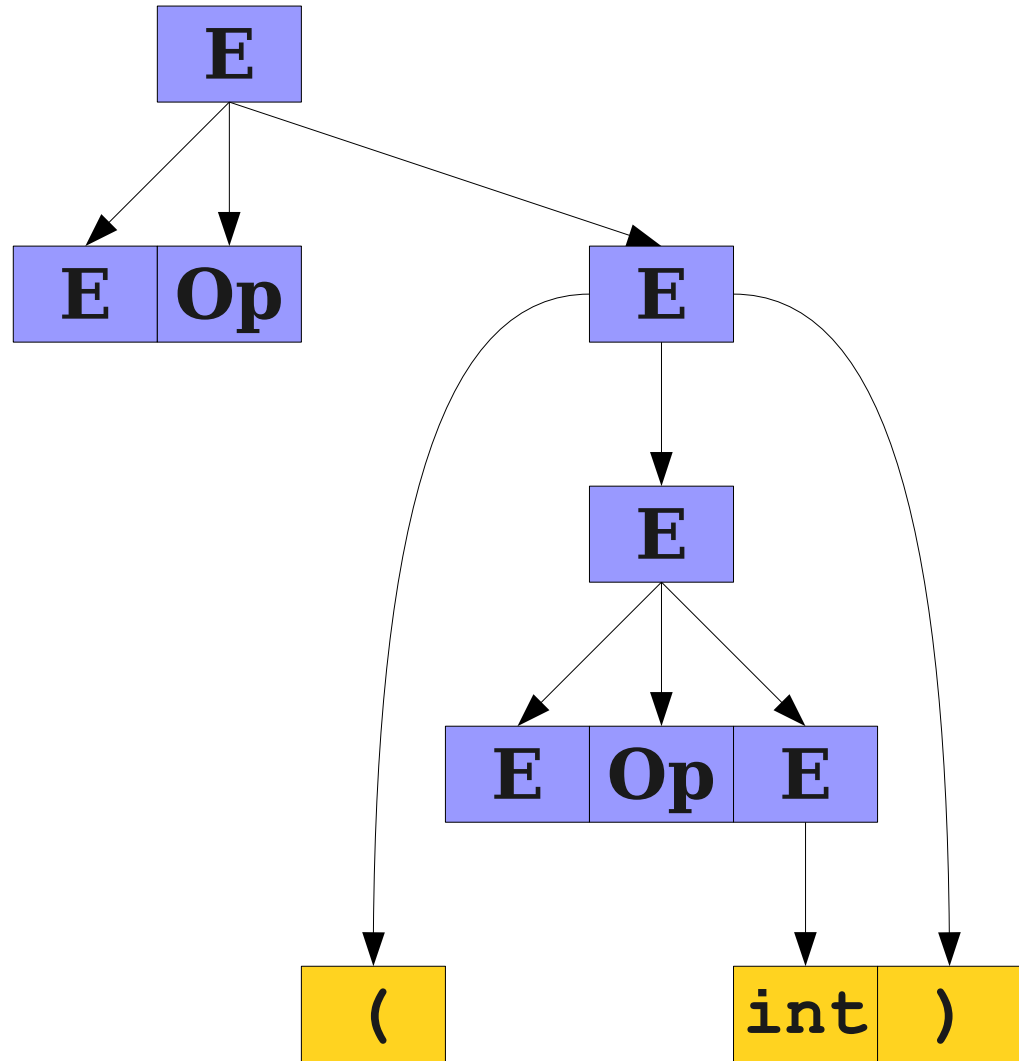
**E**  
⇒ **E Op E**  
⇒ **E Op (E)**  
⇒ **E Op (E Op E)**  
⇒ **E Op (E Op int)**



$\begin{aligned} E &\rightarrow E \text{ Op } E \mid \text{int} \mid (E) \\ \text{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

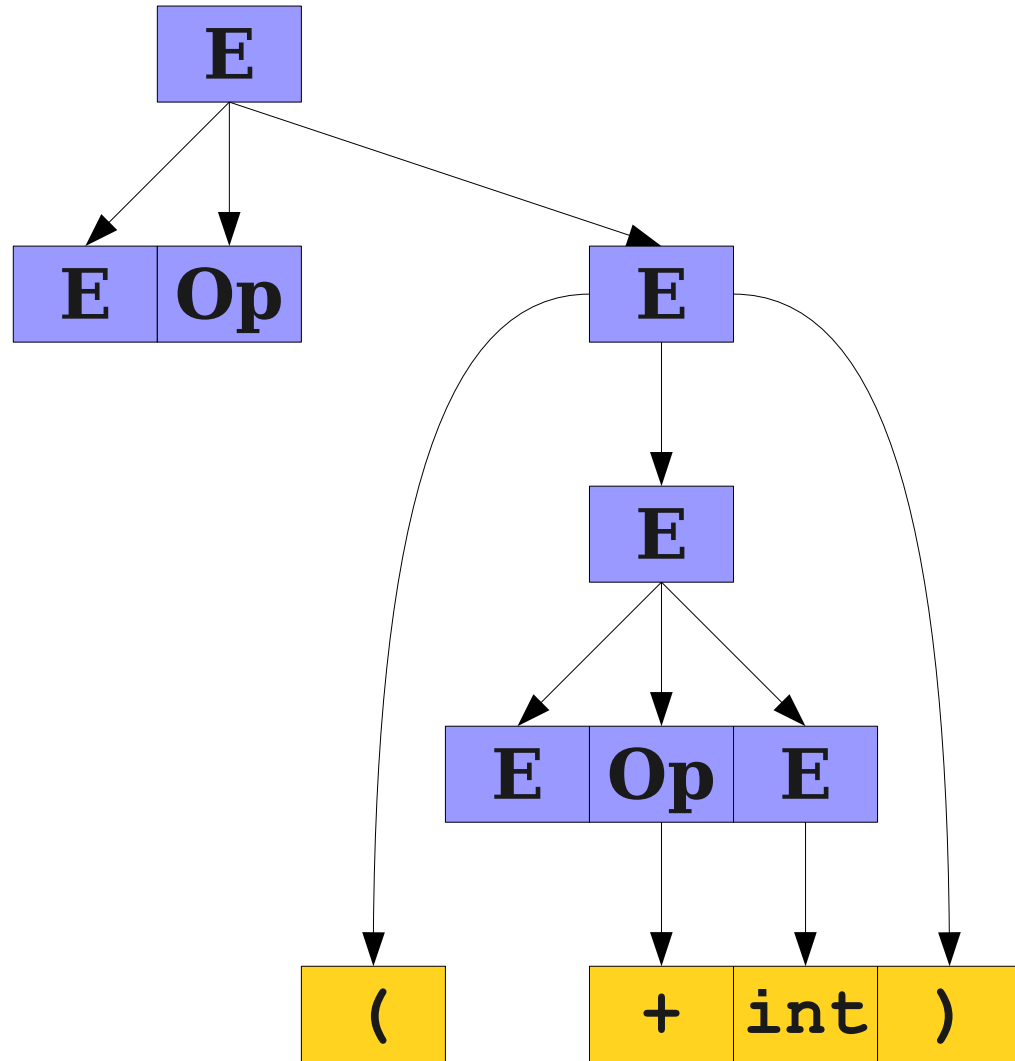
**E**  
⇒ **E Op E**  
⇒ **E Op (E)**  
⇒ **E Op (E Op E)**  
⇒ **E Op (E Op int)**  
⇒ **E Op (E + int)**



$\begin{aligned} E &\rightarrow E \text{ Op } E \mid \text{int} \mid (E) \\ \text{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

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

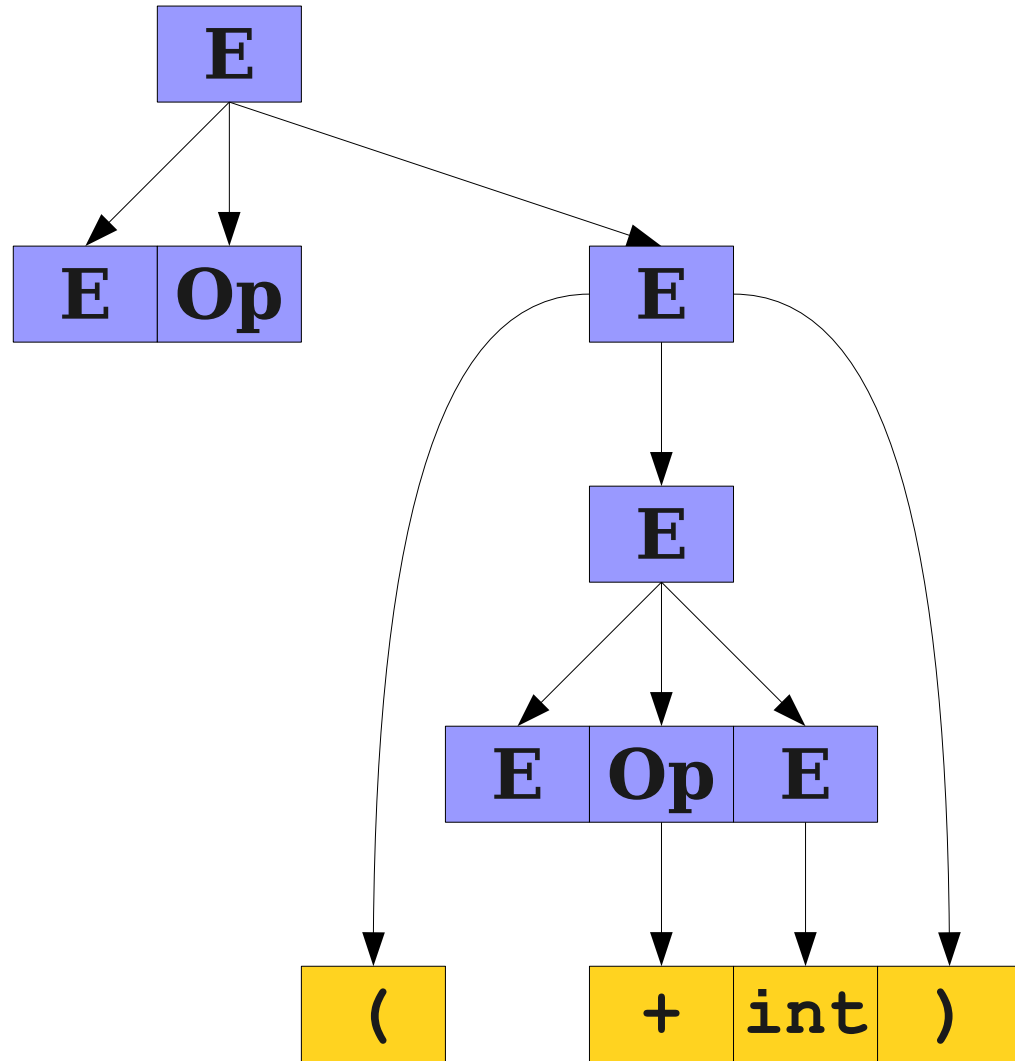


$\begin{aligned} E &\rightarrow E \text{ Op } E \mid \text{int} \mid (E) \\ \text{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------



# Parse Trees

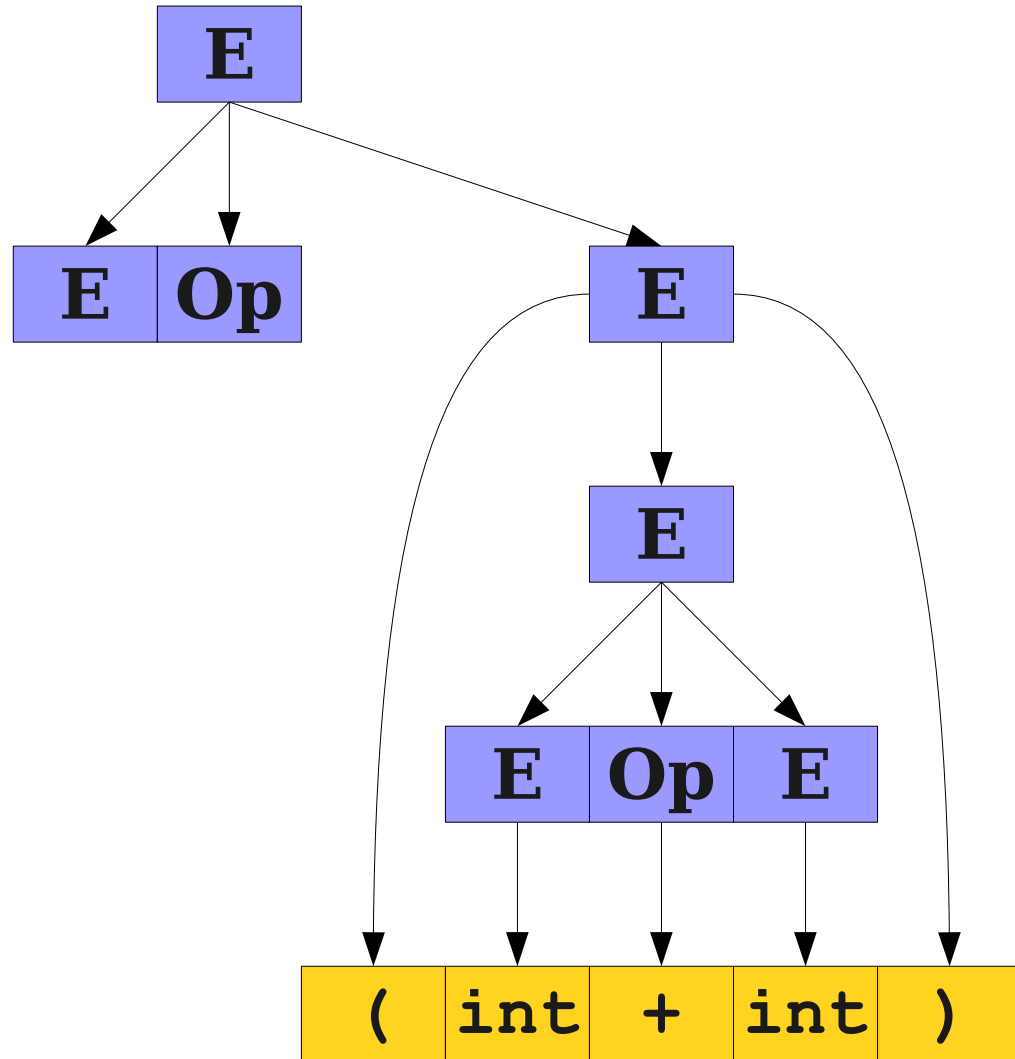
**E**  
⇒ **E Op E**  
⇒ **E Op (E)**  
⇒ **E Op (E Op E)**  
⇒ **E Op (E Op int)**  
⇒ **E Op (E + int)**  
⇒ **E Op (int + int)**



$\begin{aligned} E &\rightarrow E \text{ Op } E \mid \text{int} \mid (E) \\ \text{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------

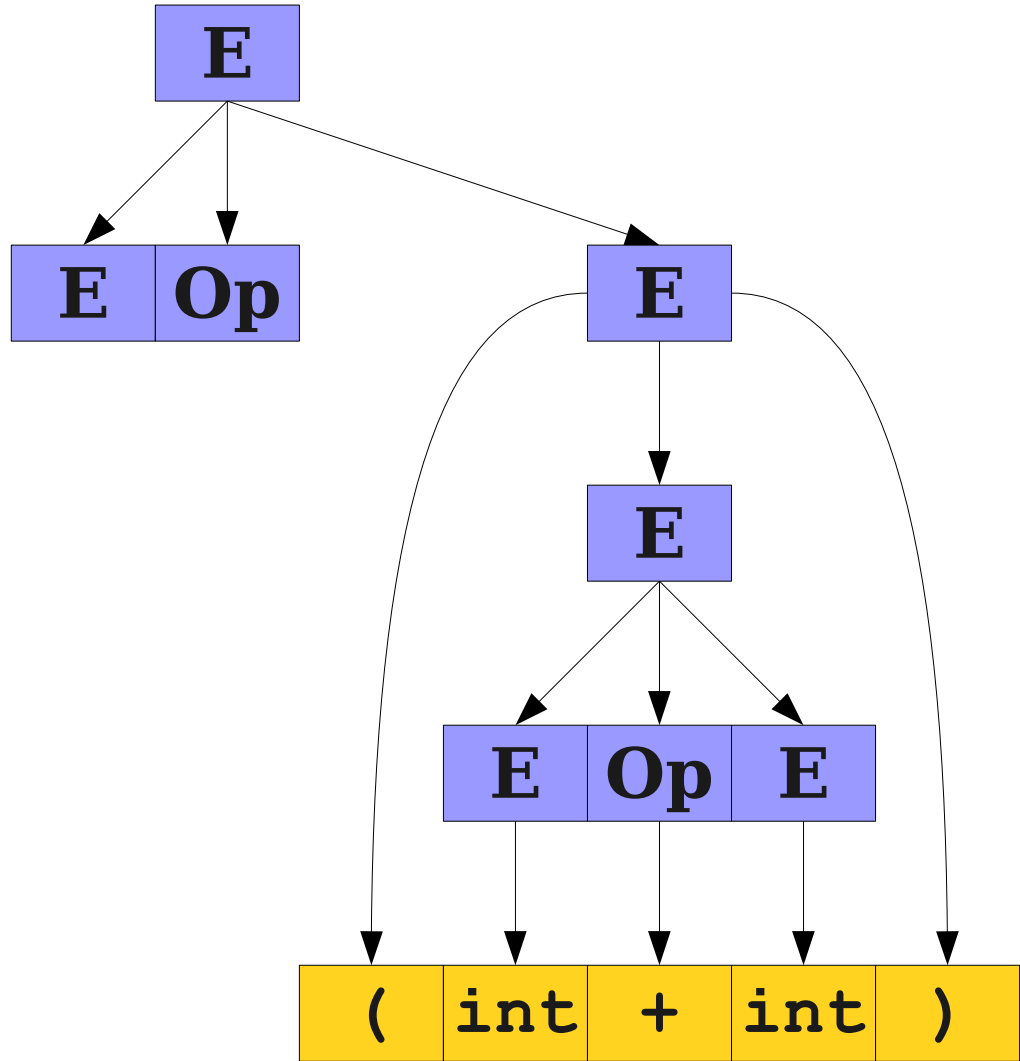
# Parse Trees

**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)**



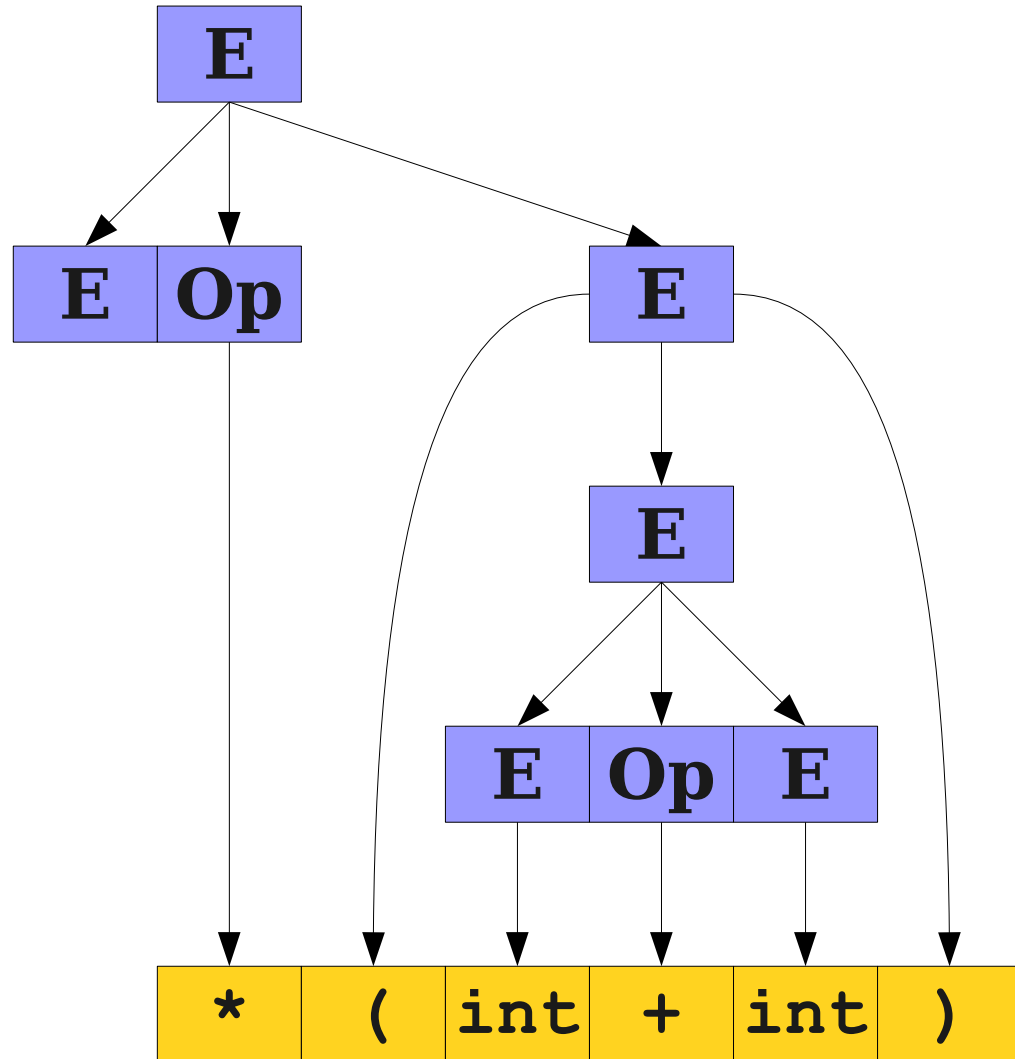
$\begin{aligned} E &\rightarrow E \text{ Op } E \mid \text{int} \mid (E) \\ \text{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

$$\begin{aligned} & \mathbf{E} \\ \Rightarrow & \mathbf{E} \mathbf{Op} \mathbf{E} \\ \Rightarrow & \mathbf{E} \mathbf{Op} (\mathbf{E}) \\ \Rightarrow & \mathbf{E} \mathbf{Op} (\mathbf{E} \mathbf{Op} \mathbf{E}) \\ \Rightarrow & \mathbf{E} \mathbf{Op} (\mathbf{E} \mathbf{Op} \text{int}) \\ \Rightarrow & \mathbf{E} \mathbf{Op} (\mathbf{E} + \text{int}) \\ \Rightarrow & \mathbf{E} \mathbf{Op} (\text{int} + \text{int}) \\ \Rightarrow & \mathbf{E} * (\text{int} + \text{int}) \end{aligned}$$

$$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E} \mathbf{Op} \mathbf{E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$$

# Parse Trees

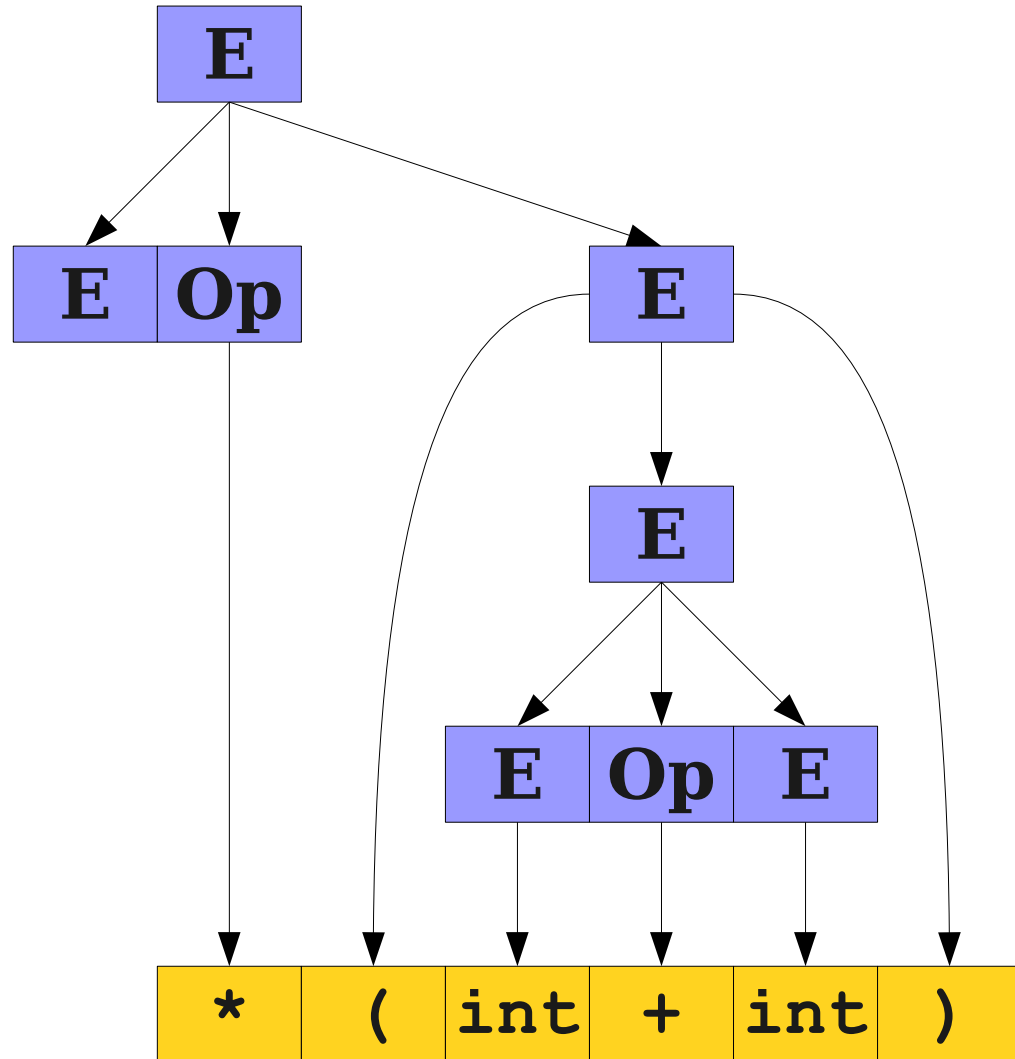
**E**  
⇒ **E Op E**  
⇒ **E Op (E)**  
⇒ **E Op (E Op E)**  
⇒ **E Op (E Op int)**  
⇒ **E Op (E + int)**  
⇒ **E Op (int + int)**  
⇒ **E \* (int + int)**



$\begin{aligned} E &\rightarrow E \text{ Op } E \mid \text{int} \mid (E) \\ \text{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------

# Parse Trees

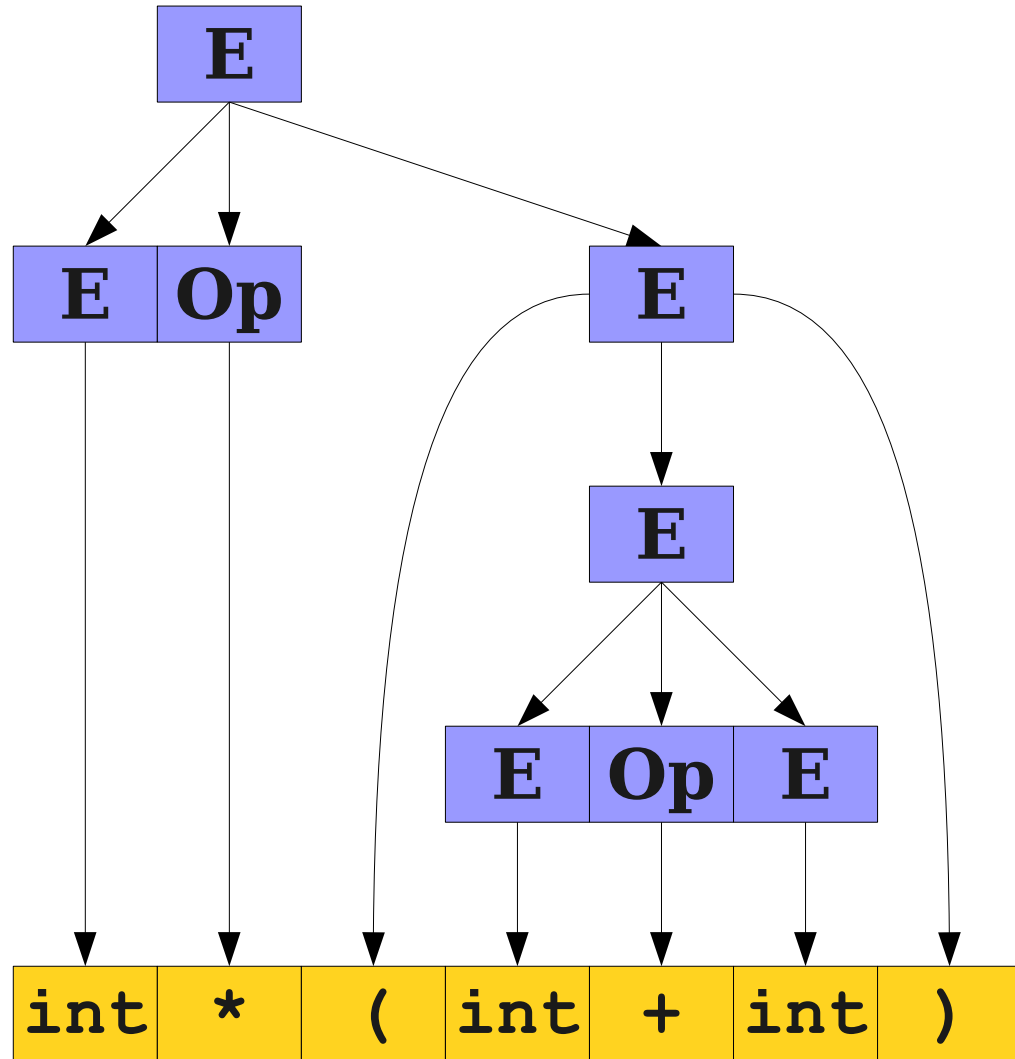
**E**  
⇒ **E Op E**  
⇒ **E Op (E)**  
⇒ **E Op (E Op E)**  
⇒ **E Op (E Op int)**  
⇒ **E Op (E + int)**  
⇒ **E Op (int + int)**  
⇒ **E \* (int + int)**  
⇒ **int \* (int + int)**



$\begin{aligned} E &\rightarrow E \text{ Op } E \mid \text{int} \mid (E) \\ \text{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
------------------------------------------------------------------------------------------------------------------------------------------

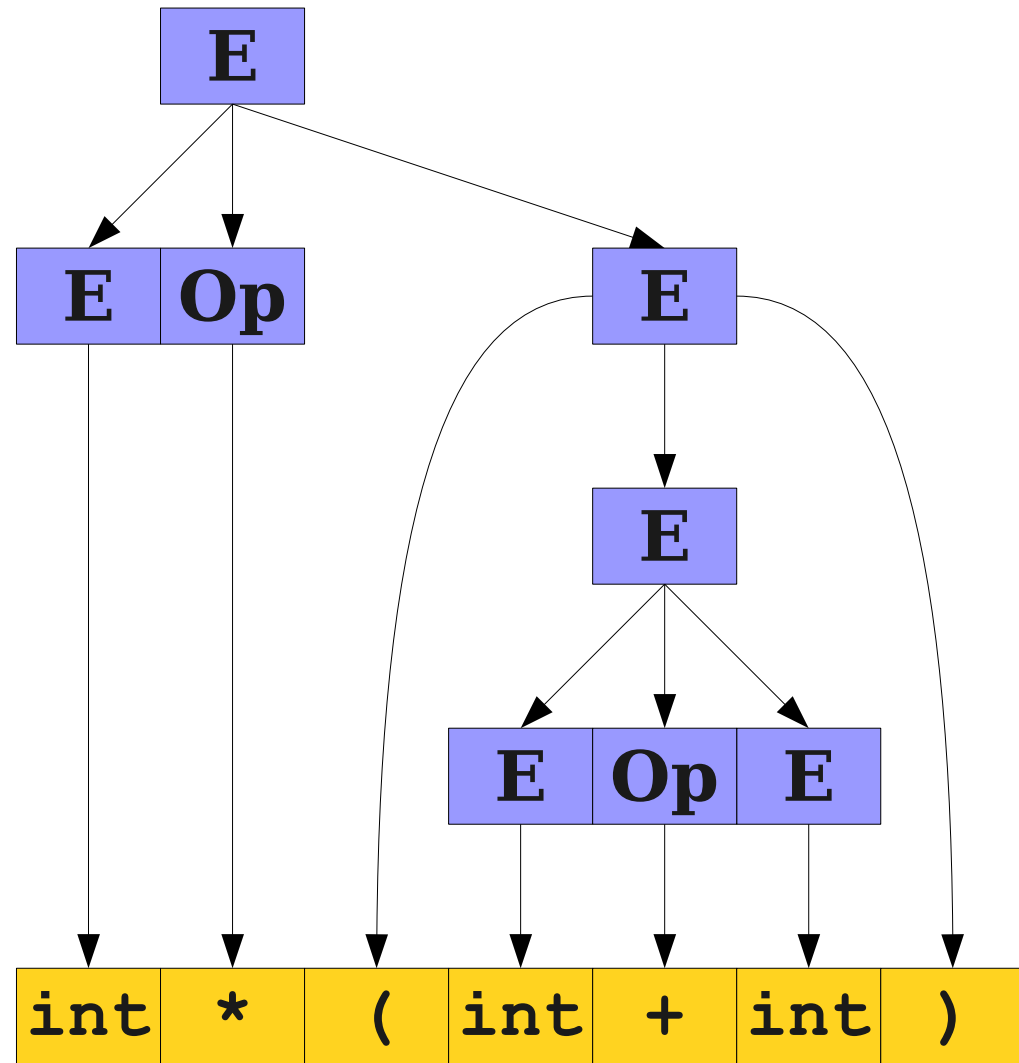
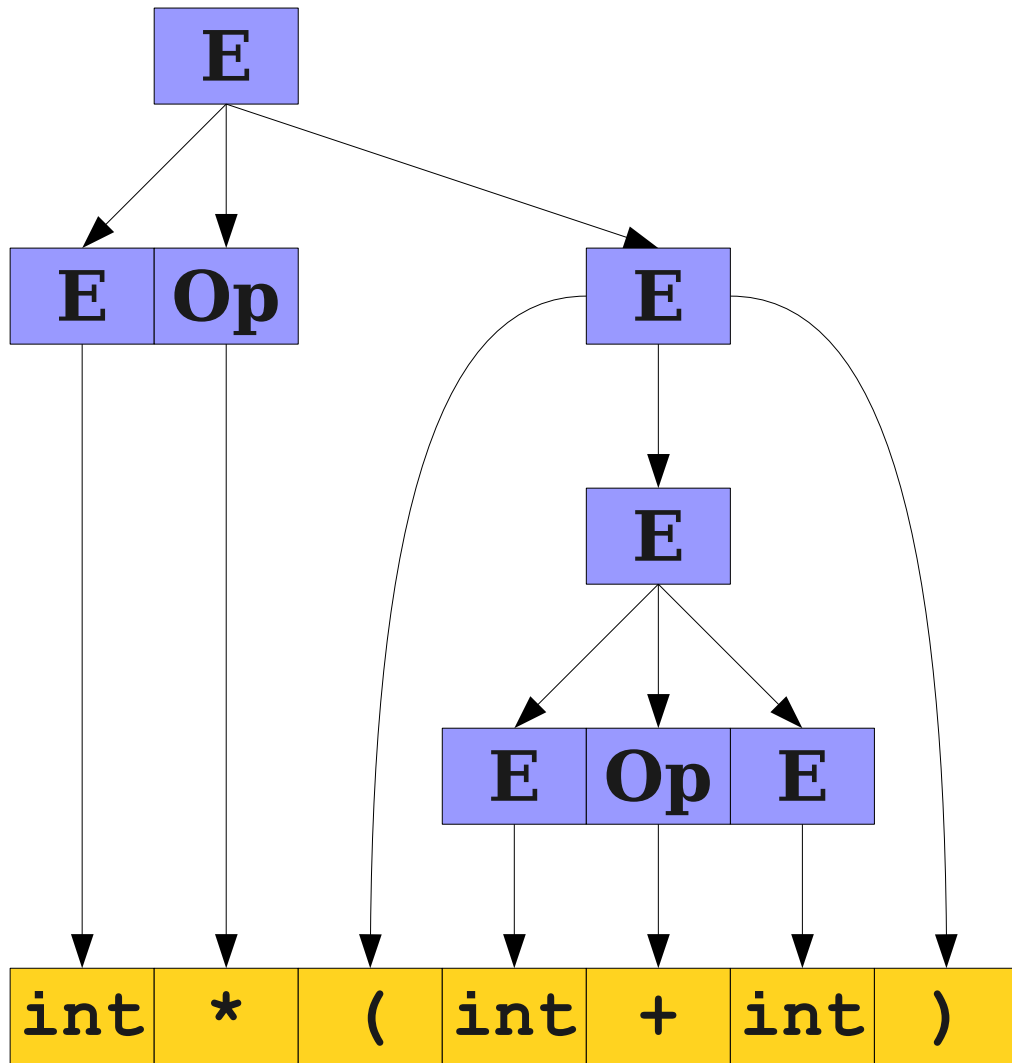
# Parse Trees

**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)**



**E** → **E Op E** | **int** | **(E)**  
**Op** → **+** | **\*** | **-** | **/**

# For Comparison



# Parse Trees

- A **parse tree** is a tree encoding the steps in a derivation.
- Each internal node is labeled with a nonterminal.
- Each leaf node is labeled with a terminal.
- Reading the leaves from left to right gives the string that was produced.

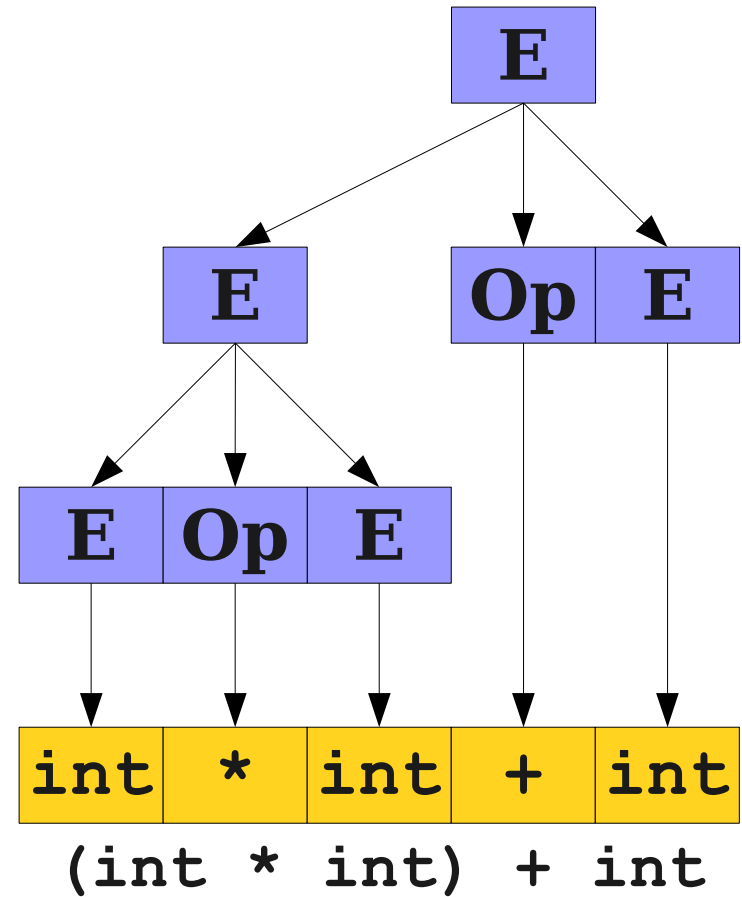
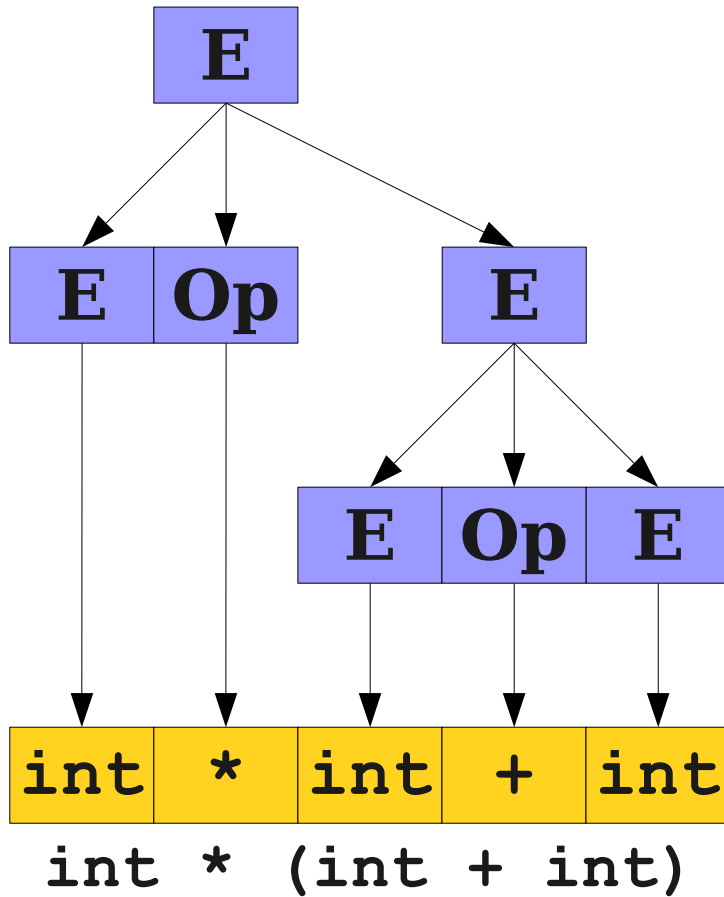


# Parsing

- Given a context-free grammar, the problem of **parsing** a string is to find a parse tree for that string.
- Applications to compilers:
  - Given a CFG describing the structure of a programming language and an input program (string), recover the parse tree.
  - The parse tree represents the structure of the program – what's declared where, how expressions nest, etc.

# Challenges in Parsing

# A Serious Problem



**E** → **E Op E** | **int**  
**Op** → **+** | **\*** | **-** | **/**

# Ambiguity

- A CFG is said to be **ambiguous** if there is at least one string with two or more parse trees.
- Note that ambiguity is a property of *grammars*, not *languages*: there can be multiple grammars for the same language, where some are ambiguous and some aren't.
- Some languages are *inherently ambiguous*: there are no unambiguous grammars for those languages.

# Resolving Ambiguity

- Designing unambiguous grammars is tricky and requires planning from the start.
- It's hard to start with an ambiguous grammar and to manually massage it into an unambiguous one.
- Often, have to throw the whole thing out and start over.

# Resolving Ambiguity

- We have just seen that this grammar is ambiguous:

$$\mathbf{E} \rightarrow \mathbf{E} \mathbf{Op} \mathbf{E} \mid \mathbf{int}$$
$$\mathbf{Op} \rightarrow + \mid - \mid * \mid /$$

- Goals:
  - Eliminate the ambiguity from the grammar.
  - Make the only parse trees for the grammar the ones corresponding to operator precedence.

# Operator Precedence

- Can often eliminate ambiguity from grammars with operator precedence issues by building precedences into the grammar.
- 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.

int	*	int	*	int	+	int	*	int	-	int
-----	---	-----	---	-----	---	-----	---	-----	---	-----

# Operator Precedence

- Can often eliminate ambiguity from grammars with operator precedence issues by building precedences into the grammar.
- 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.

int	*	int	*	int	+	int	*	int	-	int
-----	---	-----	---	-----	---	-----	---	-----	---	-----



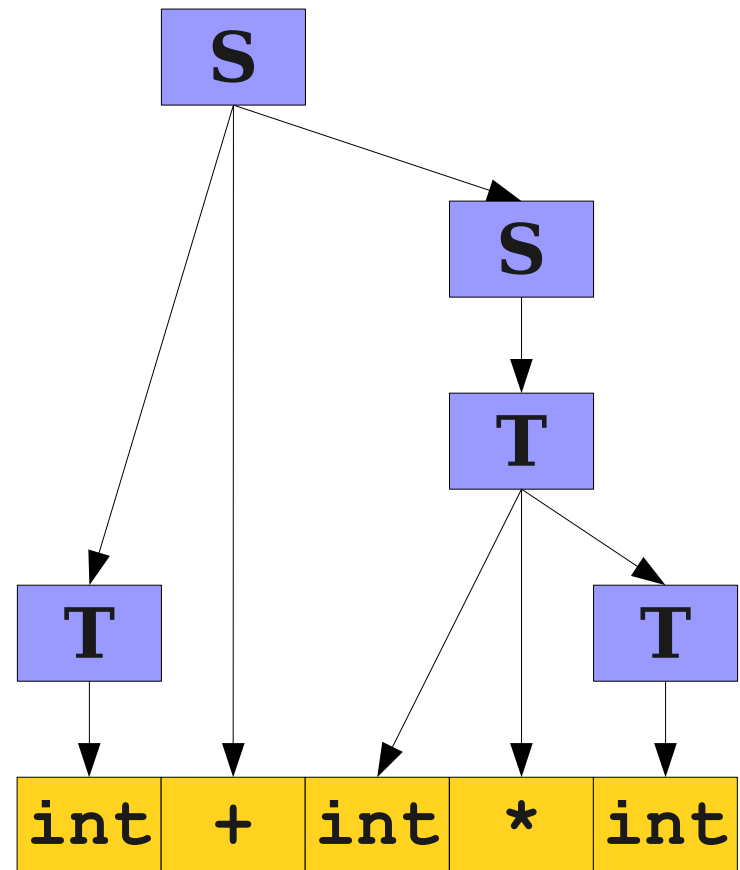
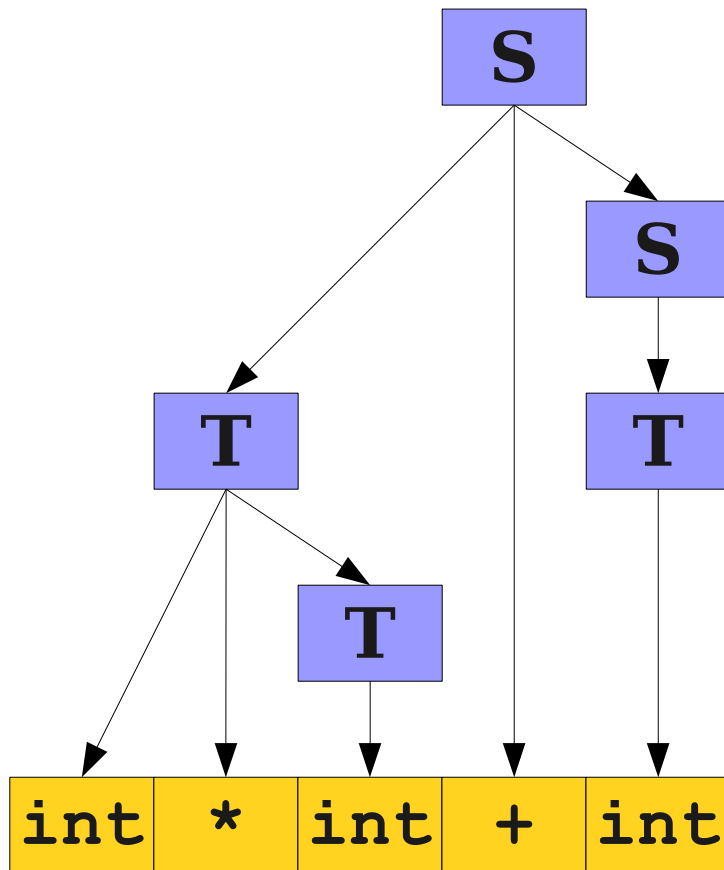
# Rebuilding the Grammar

- Idea: Force a construction order where
  - First decide how many “blocks” there will be of terms joined by  $+$  and  $-$ .
  - Then, expand those blocks by filling in the integers multiplied and divided together.
- One possible grammar:

$$S \rightarrow T \mid T + S \mid T - S$$

$$T \rightarrow \text{int} \mid \text{int} * T \mid \text{int} / T$$

# An Unambiguous Grammar



$$\begin{aligned} S &\rightarrow T \mid T + S \mid T - S \\ T &\rightarrow \text{int} \mid \text{int} * T \mid \text{int} / T \end{aligned}$$

# Summary

- Context-free grammars give a formalism for describing languages by generating all the strings in the language.
- Context-free languages are a strict superset of the regular languages.
- CFGs can be designed by finding a “build order” for a given string.
- Ambiguous grammars generate some strings with two different parse trees.

# Next Time

- **Turing Machines**
  - What does a computer with unbounded memory look like?
  - How do you program them?