# Context-Free Languages

# The Limits of Regular Languages

- The **pumping lemma for regular languages** can be used to establish limits on what languages are regular.
- If we want to describe more complex languages, we need a more powerful formalism.

# Context-Free Grammars

- A **context-free grammar** (or **CFG**) is an entirely different formalism for defining certain languages.
- 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**

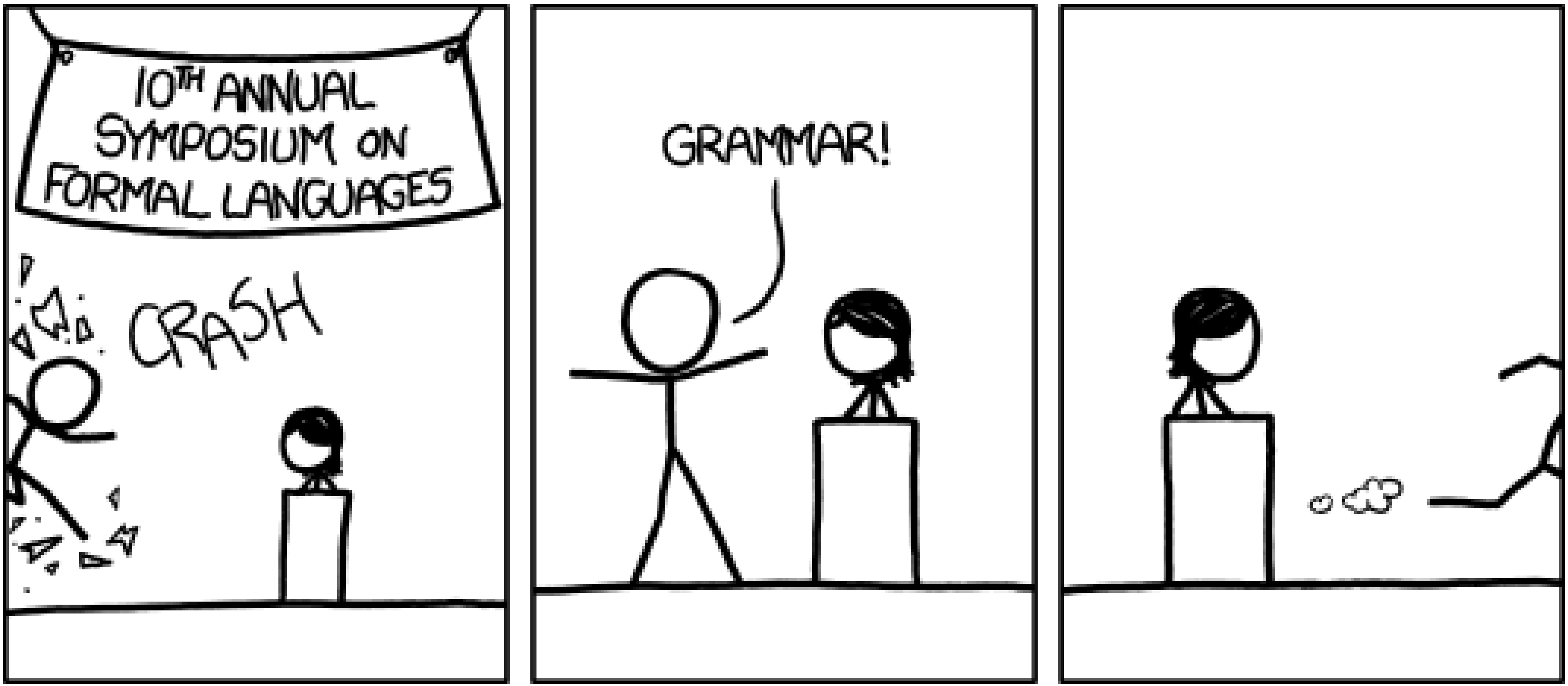
**E** → `(`**E**`)`

**Op** → `+`

**Op** → `-`

**Op** → `*`

**Op** → `/`




http://xkcd.com/1090/

# A Notational Shorthand

**E** → `int`

**E** → **E Op E**

**E** → `(`**E**`)`

**Op** → `+`

**Op** → `-`

**Op** → `*`

**Op** → `/`

# A Notational Shorthand

**E** → `int` | **E Op E** | `(`**E**`)`

**Op** → `+` | `-` | `*` | `/`

# Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \rightarrow \texttt{a*b}$$

# Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \rightarrow \mathbf{A}\mathtt{b}$$

$$\mathbf{A} \rightarrow \mathbf{A}\mathtt{a} \mid \varepsilon$$

# Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \rightarrow \texttt{a(b|c*)}$$

# Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}$$
$$\mathbf{X} \rightarrow \mathtt{b} \mid \mathbf{C}$$
$$\mathbf{C} \rightarrow \mathbf{C}\mathtt{c} \mid \varepsilon$$

# More Context-Free Grammars

- Chemicals!
  - $C_{19}H_{14}O_5S$
  - $Cu_3(CO_3)_2(OH)_2$
  - $MnO_4^-$
  - $S^{2-}$

**Form** → **Cmp** | **Cmp Ion**

**Cmp** → **Term** | **Term Num** | **Cmp Cmp**

**Term** → **Elem** | `(`**Cmp**`)`

**Elem** → `H` | `He` | `Li` | `Be` | `B` | `C` | …

**Ion** → `+` | `-` | **IonNum** `+` | **IonNum** `-`

**IonNum** → `2` | `3` | `4` | ...

**Num** → `1` | **IonNum**

# CFGs for Chemistry

**Form** → **Cmp** | **Cmp Ion**

**Cmp** → **Term** | **Term Num** | **Cmp Cmp**

**Term** → **Elem** | `(`**Cmp**`)`

**Elem** → `H` | `He` | `Li` | `Be` | `B` | `C` | …

**Ion** → `+` | `-` | **IonNum** `+` | **IonNum** `-`

**IonNum** → `2` | `3` | `4` | ...

**Num** → `1` | **IonNum**

**Form**

⇒ **Cmp Ion**

⇒ **Cmp Cmp Ion**

⇒ **Cmp Term Num Ion**

⇒ **Term Term Num Ion**

⇒ **Elem Term Num Ion**

⇒ `Mn` **Term Num Ion**

⇒ `Mn` **Elem Num Ion**

⇒ `MnO` **Num Ion**

⇒ `MnO` **IonNum Ion**

⇒ $MnO_4$ **Ion**

⇒ $MnO_4^-$

# CFGs for Programming Languages

| | | |
|---|---|---|
| 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 |
| | \| | ... |

# 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. $\boldsymbol{\alpha}$, $\boldsymbol{\gamma}$, $\boldsymbol{\omega}$

# Examples

- We might write an arbitrary production as

$$\mathbf{A} \rightarrow \boldsymbol{\omega}$$

- We might write a string of a nonterminal followed by a terminal as

$$\mathbf{A}\mathtt{t}$$

- We might write an arbitrary production containing a nonterminal followed by a terminal as

$$\mathbf{B} \rightarrow \boldsymbol{\alpha}\mathbf{A}\mathtt{t}\boldsymbol{\omega}$$

# Derivations

```
  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)
```

- This sequence of steps is called a **derivation**.
- A string $\alpha A \omega$ **yields** string $\alpha \gamma \omega$ iff $A \to \gamma$ is a production.
- If $\alpha$ yields $\beta$, we write $\alpha \Rightarrow \beta$.
- We say that $\alpha$ **derives** $\beta$ iff there is a sequence of strings where

$$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \beta$$

- If $\alpha$ derives $\beta$, we write $\alpha \Rightarrow^* \beta$.

# 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

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

- That is, the set of strings derivable from the start symbol.
- If $L$ is a language and there is some CFG $G$ such that $L = \mathscr{L}(\text{G})$, then we say that $L$ is a **context-free language** (or **CFL**).

# Regular and Context-Free Languages

- Consider the following CFG $G$:

$$\mathbf{S} \rightarrow \mathtt{0}\mathbf{S}\mathtt{1} \mid \varepsilon$$

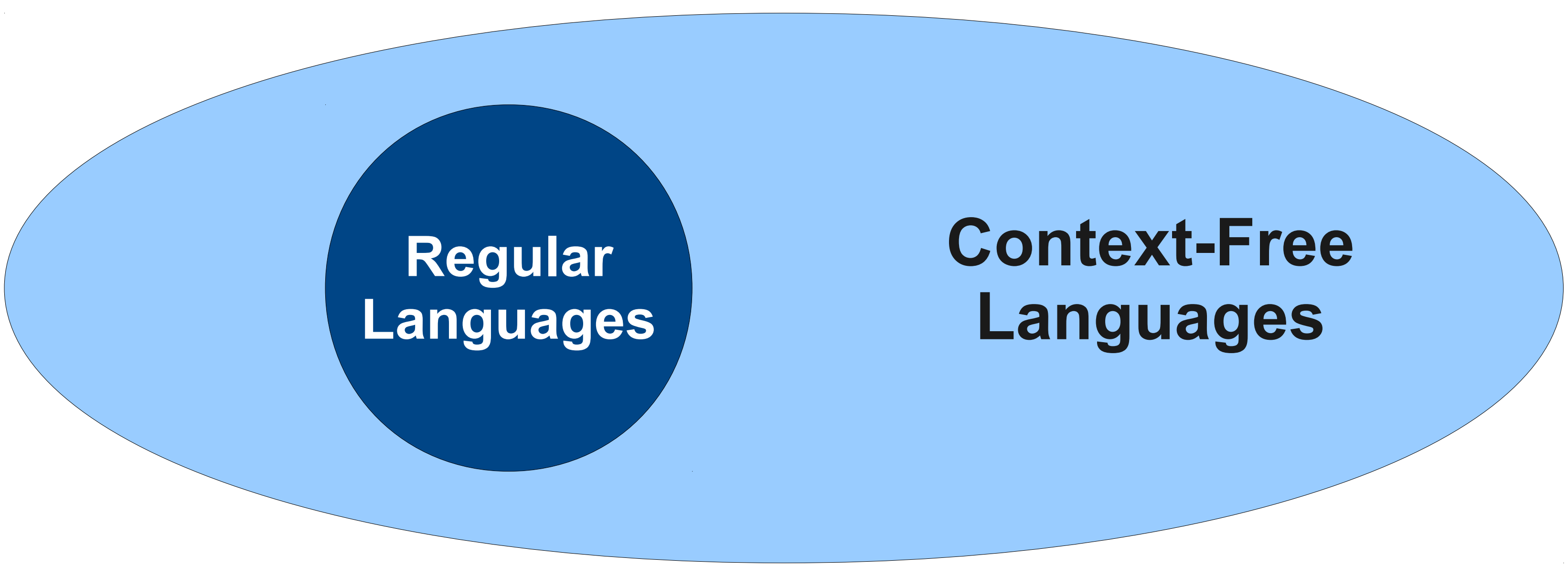
- What strings can this generate?

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

$$\mathscr{L}(G) = \{\ \mathtt{0}^n\mathtt{1}^n \mid n \in \mathbb{N}\ \}$$

# Regular and Context-Free Languages

- Context-free languages are a **strict superset** of the regular languages.
- Every regular language is context-free, but not necessarily the other way around.
- We'll see a proof of this next time.



# 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
       |  …
```

```
   STMTS
⇒ STMT STMTS
⇒ EXPR; STMTS
⇒ EXPR = EXPR; STMTS
⇒ id = EXPR; STMTS
⇒ id = EXPR + EXPR; STMTS
⇒ id = id + EXPR; STMTS
⇒ id = id + constant; STMTS
⇒ id = id + constant;
```

# Leftmost 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.
- These will be of great importance next lecture when we discuss *pushdown automata*.

# Related Derivations

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

# Derivations Revisited

- 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.

# Parse Trees

E

⇒ E Op E

⇒ int Op E

⇒ int * E

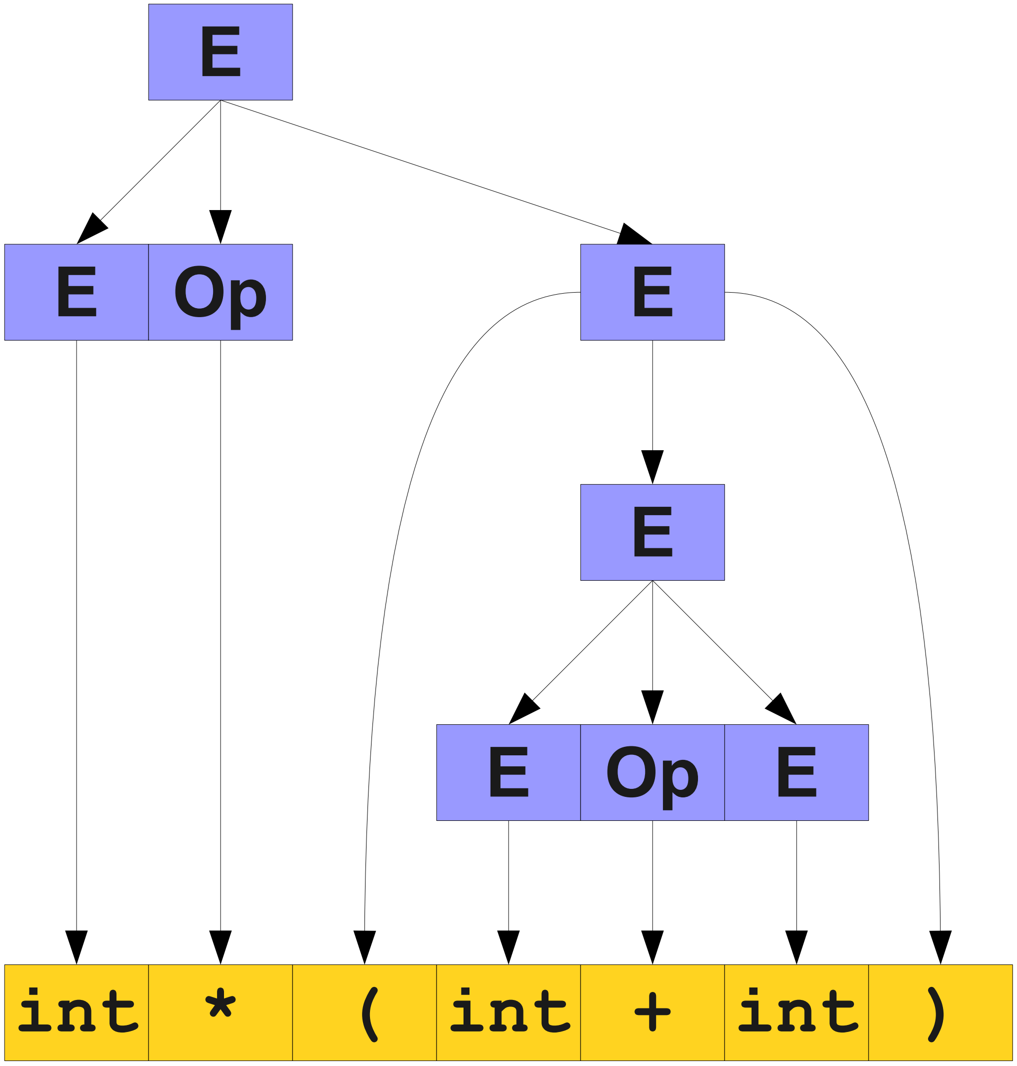
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)



# Parse Trees

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

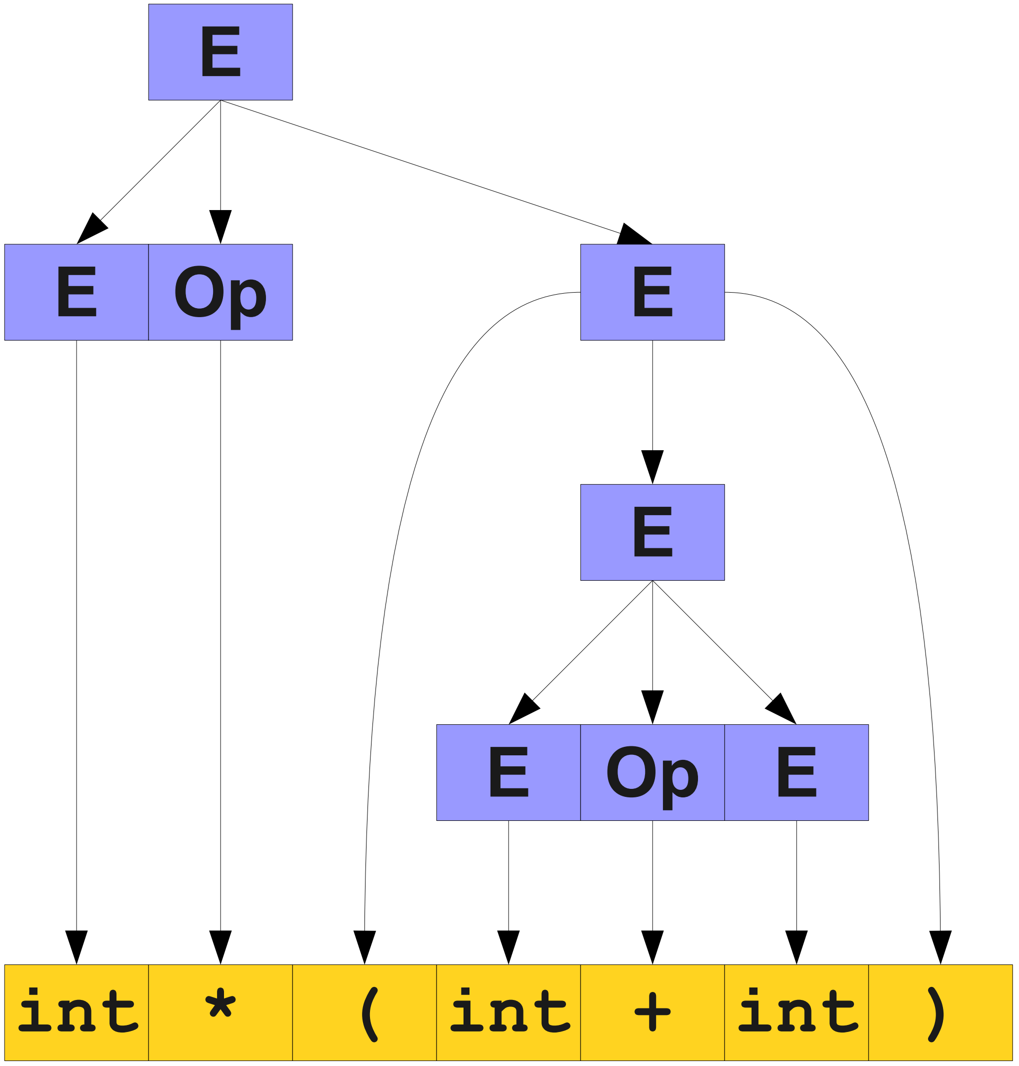
⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)

⇒ E * (int + int)

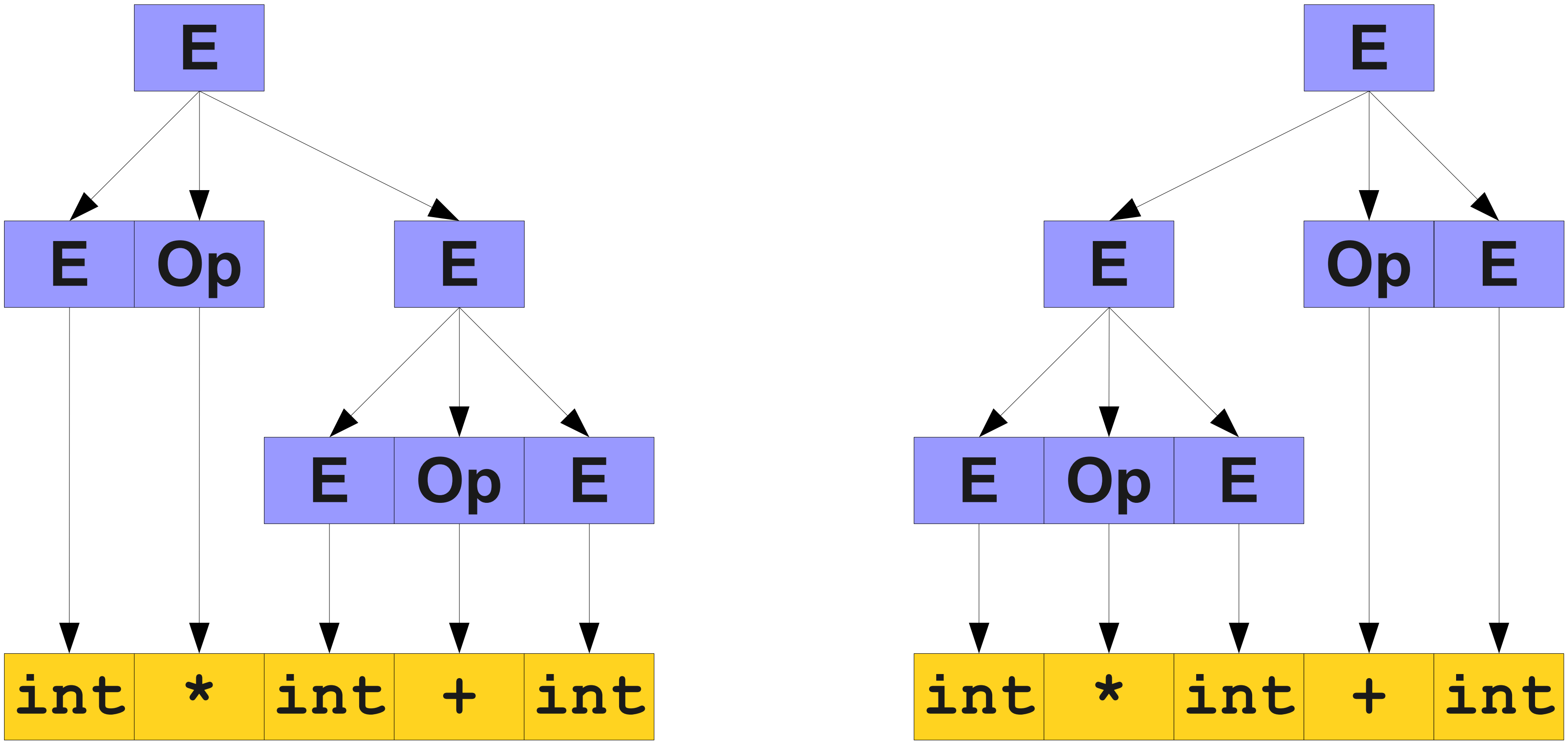
⇒ int * (int + int)



# Parse Trees

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

# A Serious Problem




int * (int + int)

(int * int) + int

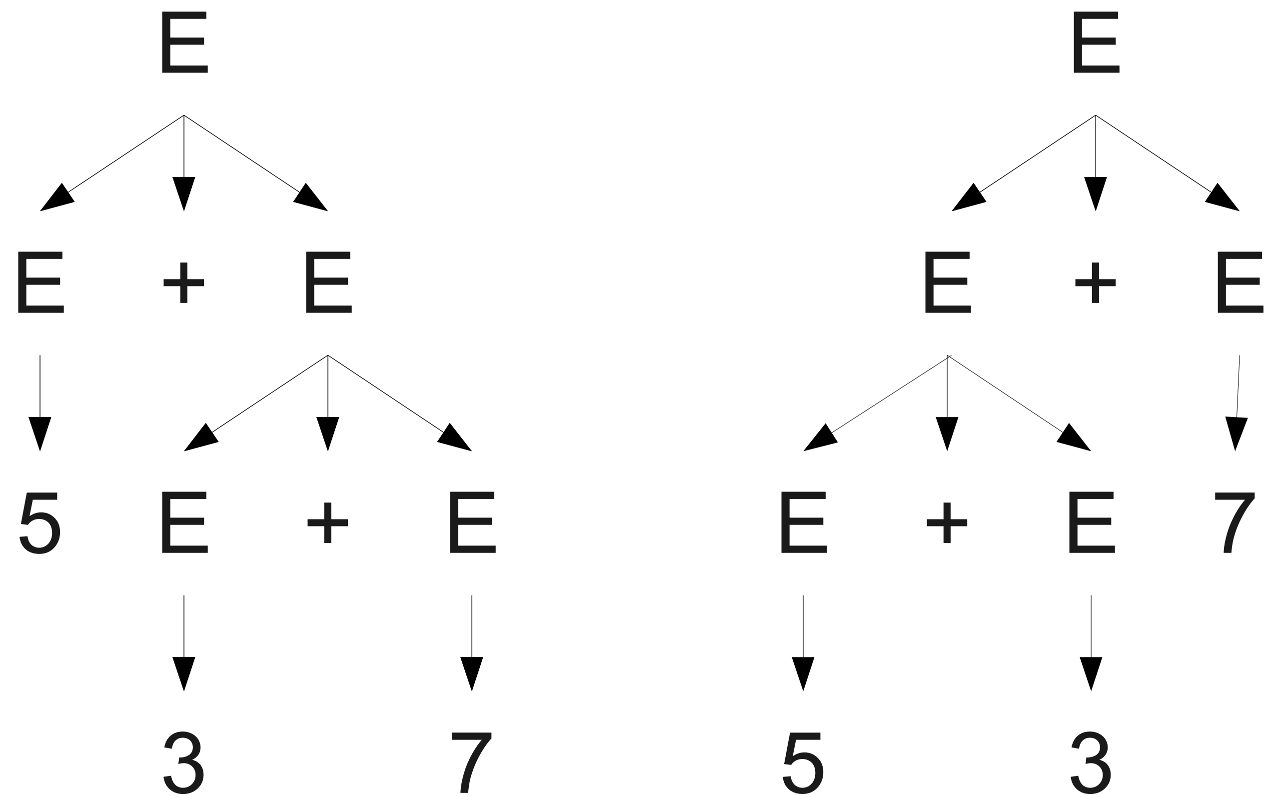
# Ambiguity

- A CFG is said to be **ambiguous** if there is at least one string with two or more parse trees.
- There is no algorithm for converting an arbitrary ambiguous grammar into an unambiguous one.
  - Some languages are **inherently ambiguous**, meaning that no unambiguous grammar exists for them.
- There is no algorithm for detecting whether an arbitrary grammar is ambiguous.

# Is Ambiguity a Problem?

- Depends on **semantics**.

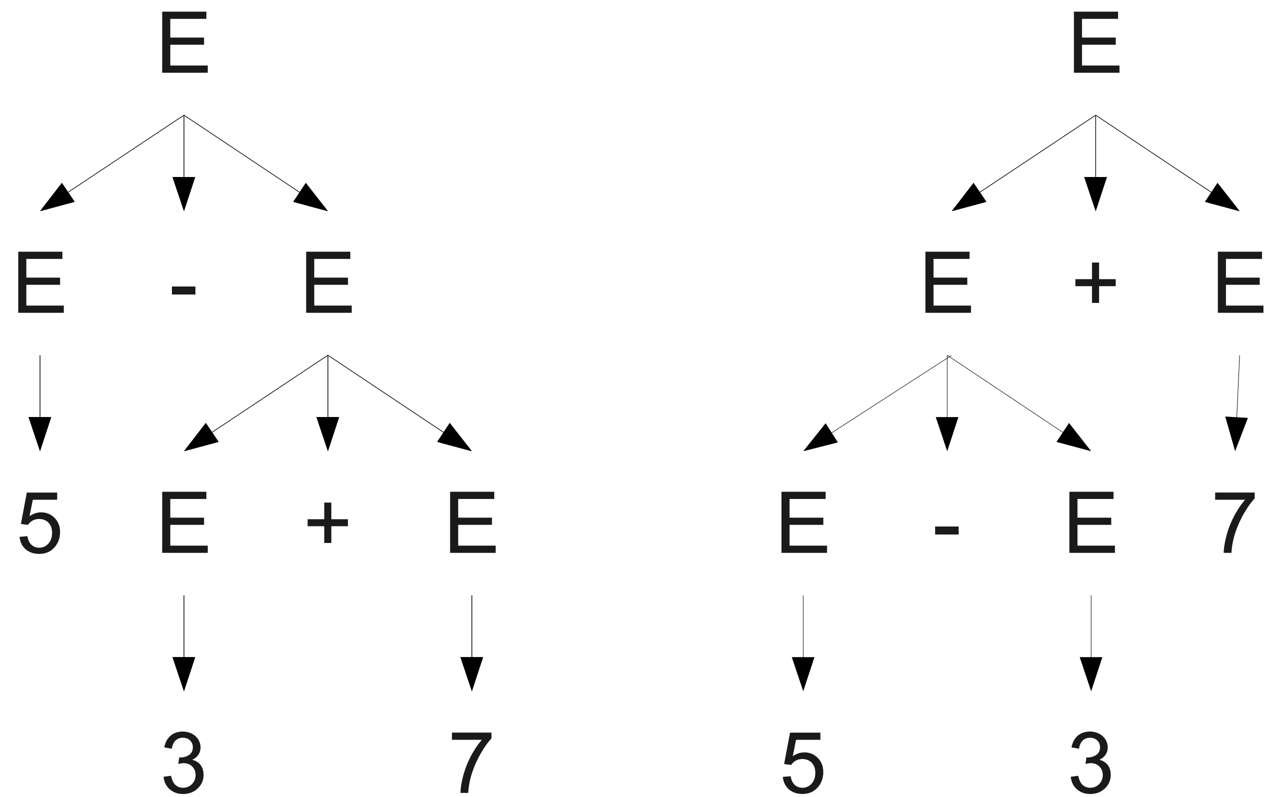
$$E \rightarrow \texttt{int} \mid E + E$$



# Is Ambiguity a Problem?

- Depends on **semantics**.

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



# Resolving Ambiguity

- If a grammar can be made unambiguous at all, it is usually made unambiguous through **layering**.
- Have exactly one way to build each piece of the string.
- Have exactly one way of combining those pieces back together.

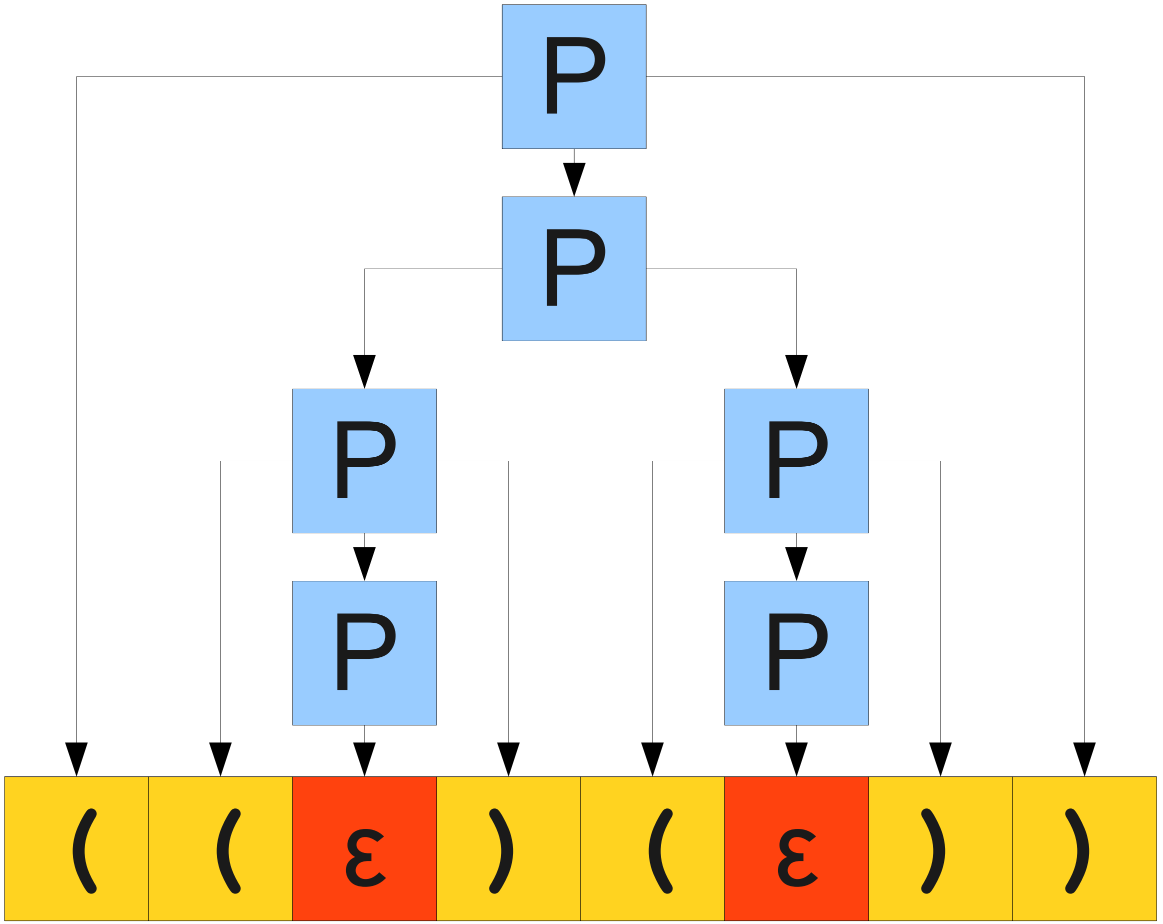
# Example: Balanced Parentheses

- Consider the language of all strings of balanced parentheses.
- Examples:
  - ε
  - `()`
  - `(()())`
  - `((()))(())()`
- Here is one possible grammar for balanced parentheses:

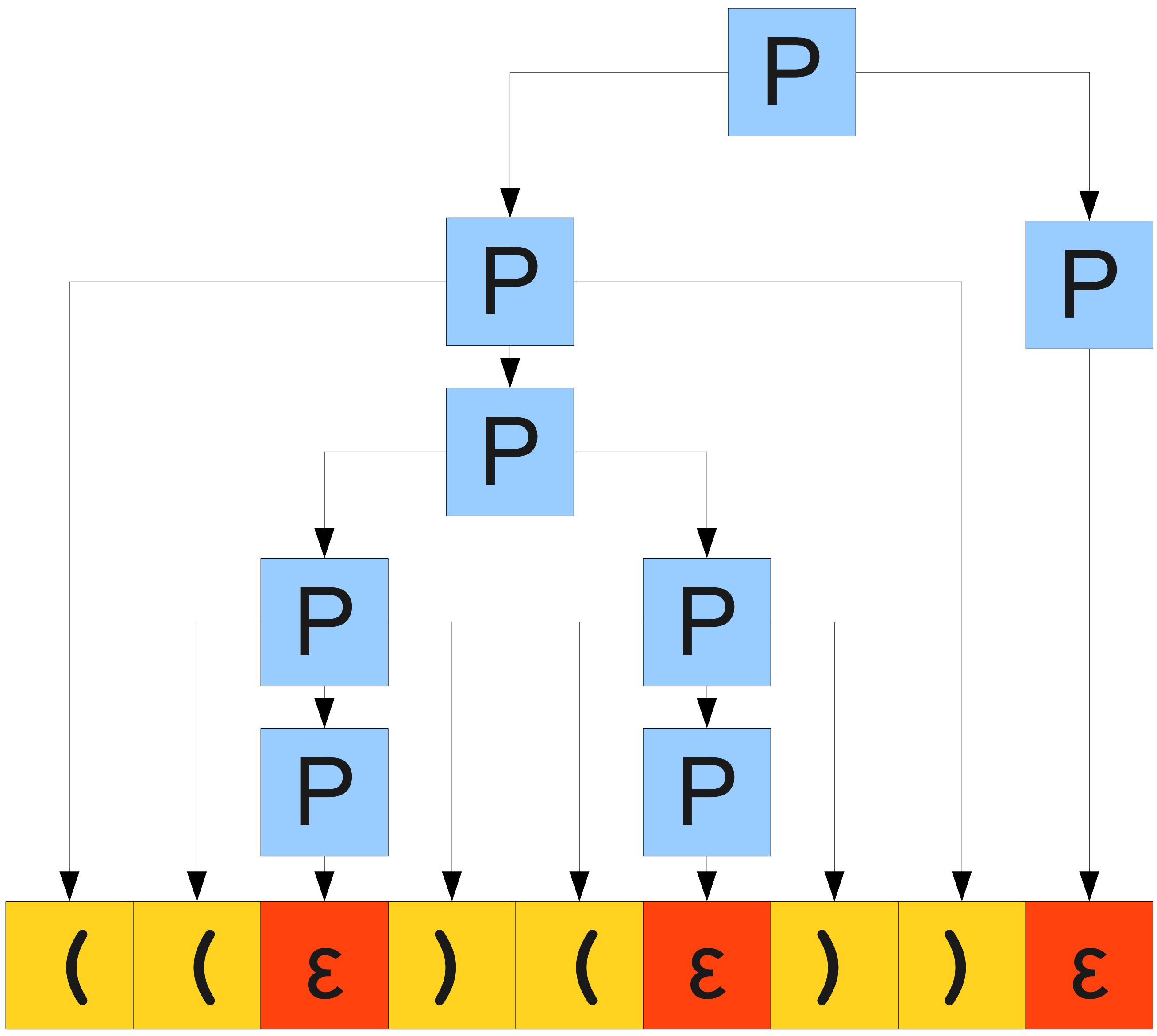
$$\mathbf{P} \rightarrow \boldsymbol{\varepsilon} \mid \mathbf{PP} \mid \mathtt{(}\mathbf{P}\mathtt{)}$$

# Balanced Parentheses

- Given the grammar **P** → **ε** | **PP** | **(P)**
- How might we generate the string (()())?



# Balanced Parentheses



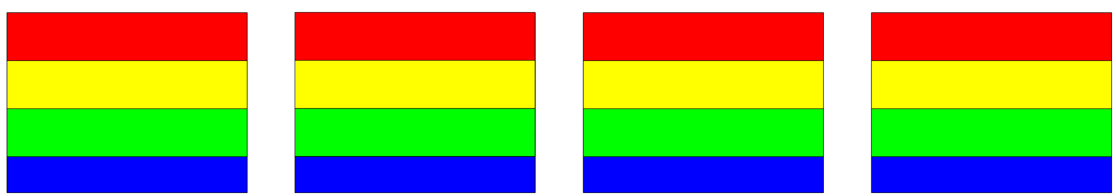
How to resolve this ambiguity?

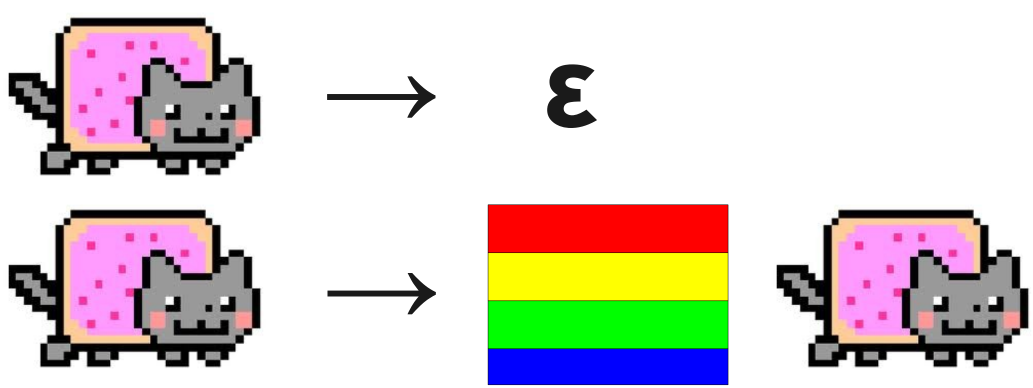
# Rethinking Parentheses

- A string of balanced parentheses is a sequence of strings that are themselves balanced parentheses.
- To avoid ambiguity, we can build the string in two steps:
  - Decide how many different substrings we will glue together.
  - Build each substring independently.

# Um... what?

- The way the nyancat flies across the sky is similar to how we can build up strings of balanced parentheses.





# Building Parentheses

- Spread a string of parentheses across the string.  There is exactly one way to do this for any number of parentheses.
- Expand out each substring by adding in parentheses and repeating.

$$\mathbf{S} \rightarrow \mathbf{P}\ \mathbf{S} \mid \varepsilon$$

$$\mathbf{P} \rightarrow (\ \mathbf{S}\ )$$

# Building Parentheses

$$S \rightarrow P\ S \mid \varepsilon$$

$$P \rightarrow (\ S\ )$$

$$
\begin{aligned}
& S \\
\Rightarrow\ & PS \\
\Rightarrow\ & PPS \\
\Rightarrow\ & PP \\
\Rightarrow\ & (S)P \\
\Rightarrow\ & (S)(S) \\
\Rightarrow\ & (PS)(S) \\
\Rightarrow\ & (P)(S) \\
\Rightarrow\ & ((S))(S) \\
\Rightarrow\ & (())(S) \\
\Rightarrow\ & (())()
\end{aligned}
$$

# Context-Free Grammars

- A regular expression can be
  - Any letter
  - ε
  - Ø
  - The concatenation of regular expressions.
  - The union of regular expressions.
  - The Kleene closure of a regular expression.
  - A parenthesized regular expression.

# Context-Free Grammars

- This gives us the following CFG:

  **R** → `a` | `b` | `c` | …

  **R** → "`ε`"

  **R** → `Ø`

  **R** → **RR**

  **R** → **R** "`|`" **R**

  **R** → **R**`*`

  **R** → `(`**R**`)`

# An Ambiguous Grammar

R → a | b | c | …

R → "ε"

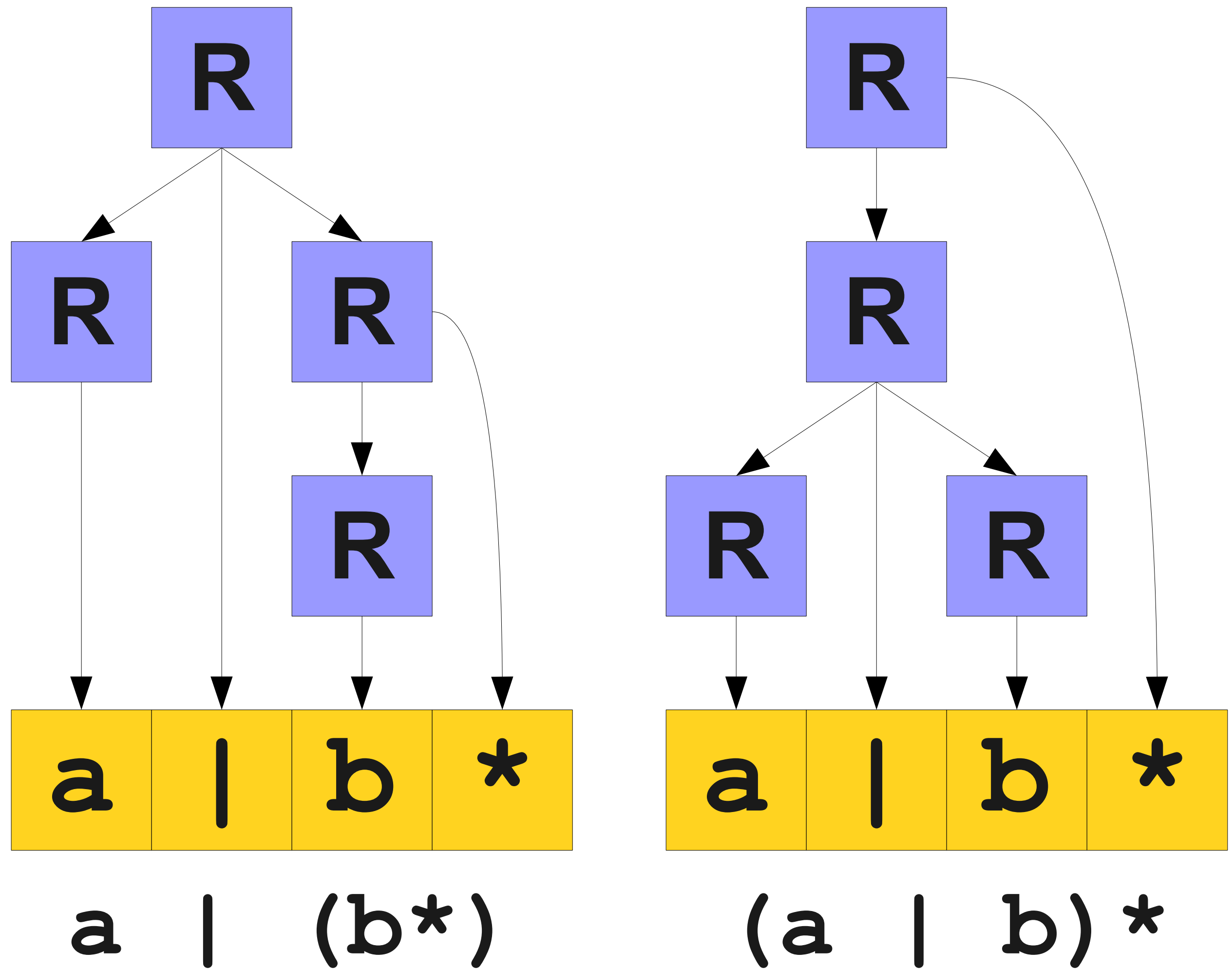
R → Ø

R → RR

R → R "|" R

R → R*

R → (R)



# Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

  **R** → **a** | **b** | **c** | …

  **R** → "**ε**"

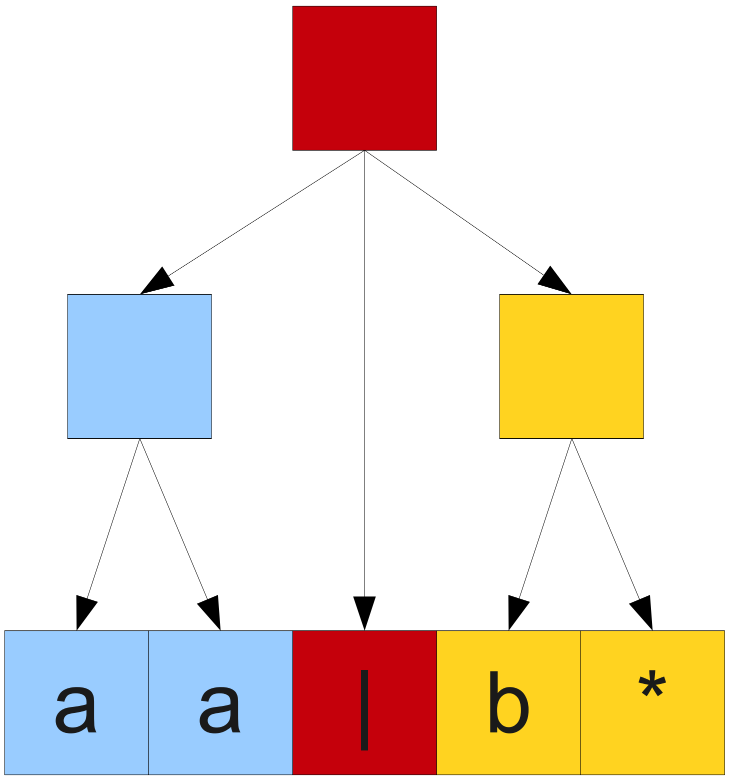
  **R** → **Ø**

  **R** → **RR**

  **R** → **R** "**|**" **R**

  **R** → **R***

  **R** → **(R)**



# Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

**R** → `a` | `b` | `c` | ...

**R** → "`ε`"

**R** → `Ø`

**R** → **RR**

**R** → **R** "`|`" **R**

**R** → **R**`*`

**R** → `(`**R**`)`

**R** → **S** | **R** "`|`" **S**

**S** → **T** | **ST**

**T** → **U** | **T**`*`

**U** → `a` | `b` | `c` | ...

**U** → "`ε`"

**U** → `Ø`

**U** → `(`**R**`)`

# Why is this unambiguous?

**R** → **S** | **R** "|" **S**

**S** → **T** | **ST**

**T** → **U** | **T***

**U** → a | b | …

**U** → "ε"

**U** → Ø

**U** → (**R**)

Unions concatenated expressions

Concatenates starred expressions

Puts stars onto atomic expressions

Only generates "atomic" expressions

**R** → **S** | **R** "|" **S**

**S** → **T** | **ST**

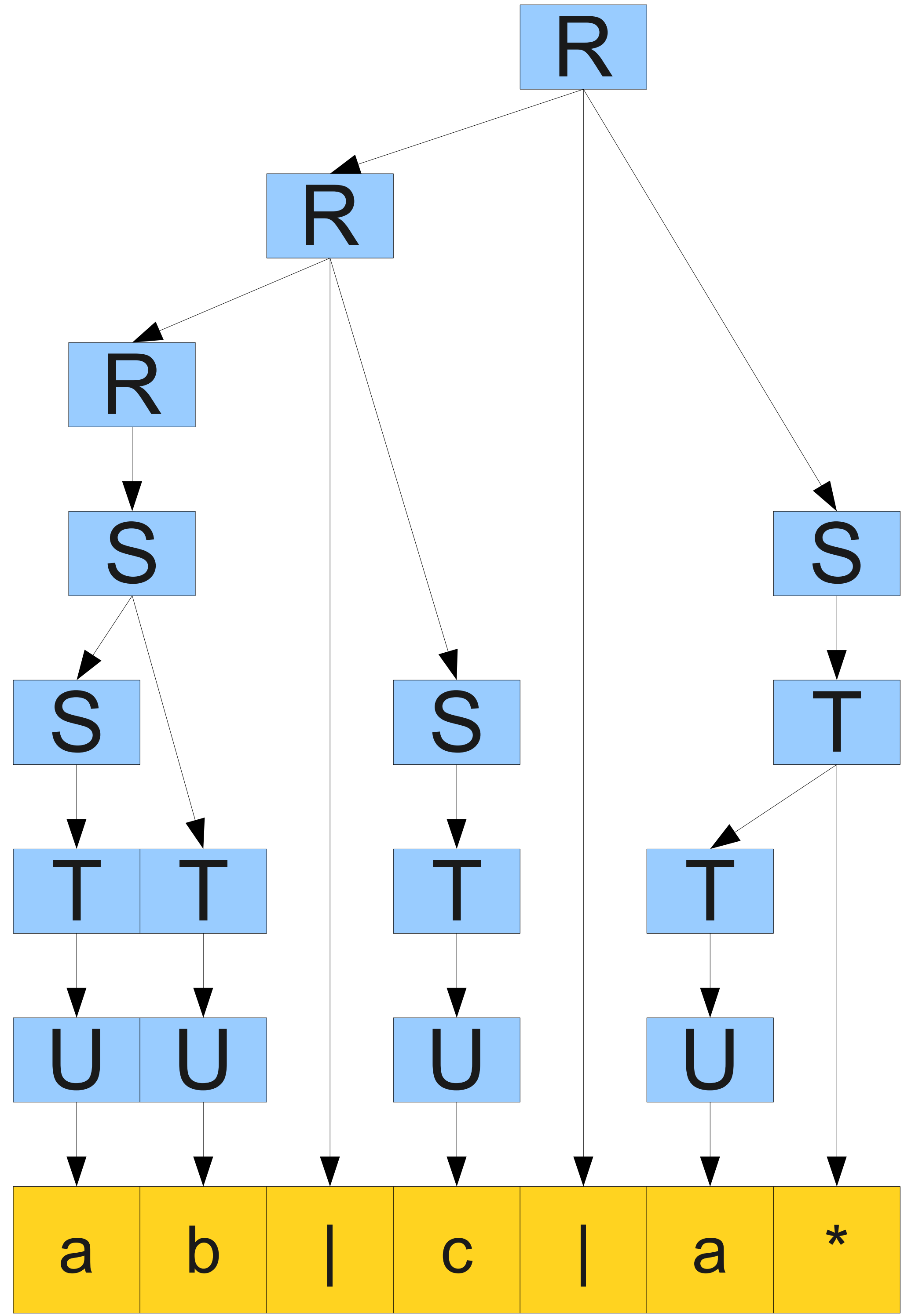
**T** → **U** | **T***

**U** → a | b | c | ...

**U** → "ε"

**U** → ø

**U** → (**R**)



# Summary

- **Context-free grammars** give a way to describe a class of formal languages (the **context-free languages**) that are strictly larger than the regular languages.
- 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.
- There is no algorithm for eliminating ambiguity; it must be done by hand.

# Closure Properties of Context-Free Languages

# Closure Properties

- If $L_1$ and $L_2$ are regular, then
  - $\overline{L}_1$ is regular.
  - $L_1 \cup L_2$ is regular.
  - $L_1 \cap L_2$ is regular.
  - $L_1L_2$ is regular.
  - $L_1$* is regular.
  - $h^*(L_1)$ is regular.
- How many of these properties still hold for context-free languages?

# The Union of CFLs

- Suppose that $L_1$ and $L_2$ are ***context-free*** languages.
- Is $L_1 \cup L_2$ a context-free language?

**X** → **S** | **E**

**S** → **PS** | **ε**
**P** → **(S)**

**E** → **E + E** | **int**

- ***Yes!*** Use the above construction.
  - Rename nonterminals in the two grammars if necessary.

# The Concatenation of CFLs

- Suppose that $L_1$ and $L_2$ are ***context-free*** languages.
- Is $L_1 L_2$ a context-free language?

**X → SE**

| | |
|---|---|
| **S → PS \| ε**<br>**P → (S)** | **E → E + E \| int** |

- ***Yes!*** Use the above construction.
  - Rename nonterminals in the two grammars if necessary.

# The Kleene Closure of CFLs

- Suppose that $L$ is a ***context-free*** language.
- Is $L^*$ a context-free language?

**X** → **EX** | **ε**

**E** → **E + E** | **`int`**

- ***Yes!*** Use the above construction.

# Closure Properties of CFLs

- If $L_1$ and $L_2$ are context-free languages, then
  - $L_1 \cup L_2$ is context-free.
  - $L_1 L_2$ is context-free.
  - $L_1$* is context-free.
  - $h$*($L_1$) is context-free.
- Do the other properties still hold?
- We'll see early next week…

# Next Time

- **Pushdown Automata**
  - Automata for recognizing CFLs.
  - A beautiful generalization of DFAs and NFAs.
  - An easy proof that any regular language is context-free.