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. ***α***, ***γ***, ***ω***

A Notational Shorthand

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

Derivations

E → **E Op E** | `int` | (**E**)
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)`

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

The Language of a Grammar

- If G is a CFG with alphabet Σ 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, $\mathscr{L}(G)$ is the set of strings derivable from the start symbol.
- Note: $\boldsymbol{\omega}$ must be in Σ^*, the set of strings made from terminals. Strings involving nonterminals aren't in the language.

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  → var
      | const
      | EXPR + EXPR
      | EXPR – EXPR
      | EXPR = EXPR
      | …
```

```
{
    var = var * var;
    if (var) var = const;
    while (var) {
        var = var + const;
    }
}
```

Context-Free Languages

- A language L is called a **context-free language** (or CFL) iff there is a CFG G such that $L = \mathscr{L}(G)$.
- 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 → **`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 → **A**`b`

A → **A**`a` | ε

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(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:

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

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:

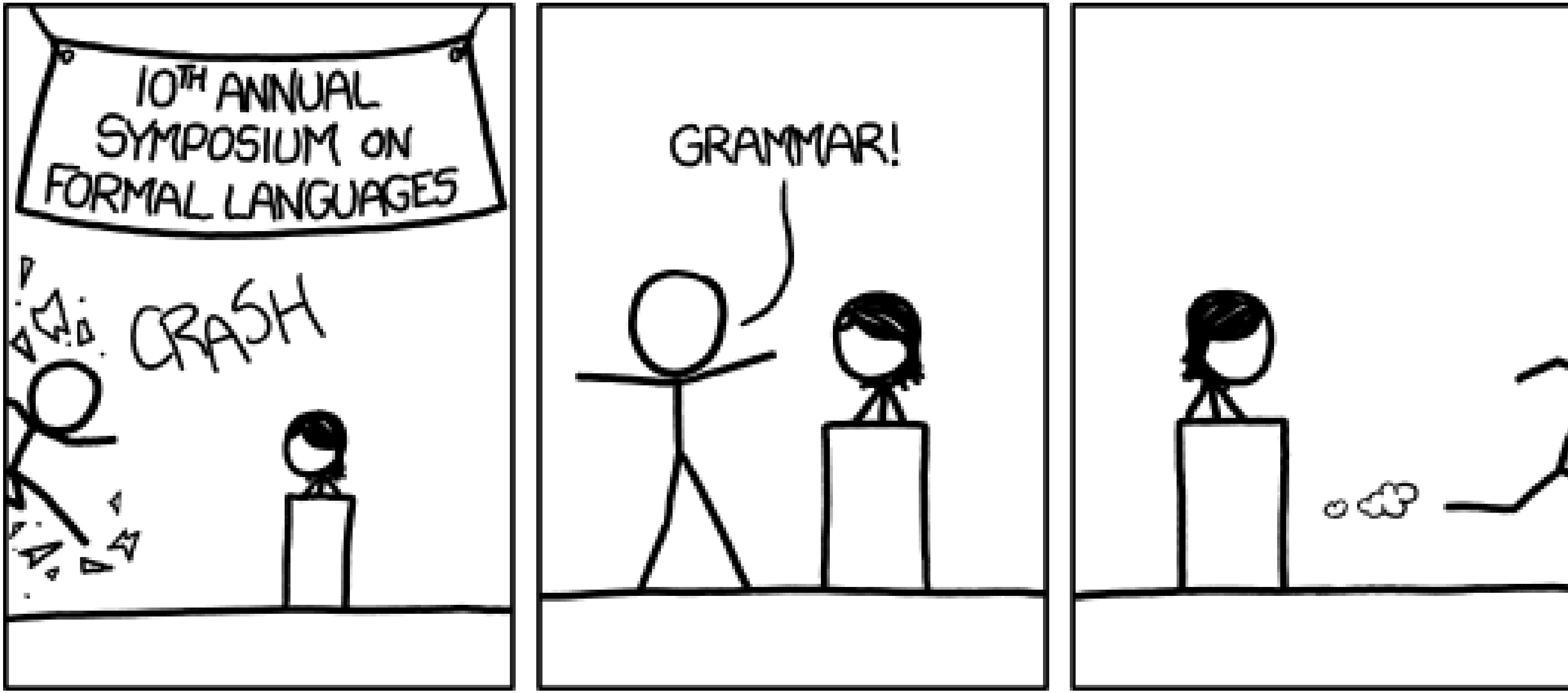
$$\mathbf{S} \to \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

- What strings can this generate?

a	a	a	a	a	a	b	b	b	b	b	b

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


Regular
Languages
CFLs
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 $L = \{ w \in \Sigma^* \mid w$ is a palindrome $\}$
- We can design a CFG for L by thinking inductively:
  - Base case: ε, **a**, and **b** are palindromes.
  - If ω is a palindrome, then **a**ω**a** and **b**ω**b** are palindromes.

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

Designing CFGs

- Let $\Sigma = \{$ **(**, **)** $\}$ and let $L = \{w \in \Sigma^* \mid w$ 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.

$$\mathbf{S} \rightarrow \mathbf{SS} \mid \mathtt{(}\mathbf{S}\mathtt{)} \mid \varepsilon$$

Designing CFGs: Watch Out!

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

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

Finding a Build Order

- Let $\Sigma = \{\texttt{a}, \stackrel{?}{=}\}$ and let $L = \{\texttt{a}^n \stackrel{?}{=} \texttt{a}^n \mid n \in \mathbb{N}\ \}$.
- 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: $\mathbf{S} \rightarrow \texttt{a}\mathbf{S}\texttt{a} \mid \stackrel{?}{=}$

$$
\begin{aligned}
& \mathbf{S} \\
\Rightarrow\ & \texttt{a}\mathbf{S}\texttt{a} \\
\Rightarrow\ & \texttt{aa}\mathbf{S}\texttt{aa} \\
\Rightarrow\ & \texttt{aaa}\mathbf{S}\texttt{aaa} \\
\Rightarrow\ & \texttt{aaa}\stackrel{?}{=}\texttt{aaa}
\end{aligned}
$$

Designing CFGs: A Caveat

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

$$\mathbf{S} \rightarrow \mathtt{l}\mathbf{S}\mathtt{r} \mid \mathtt{r}\mathbf{S}\mathtt{l} \mid \varepsilon$$

- Can you derive the string **`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: **Ø**
- When designing CFGs, make sure your recursion actually terminates!

Parse Trees

Parse Trees

E

⇒ E Op E

⇒ int Op E

⇒ int * E

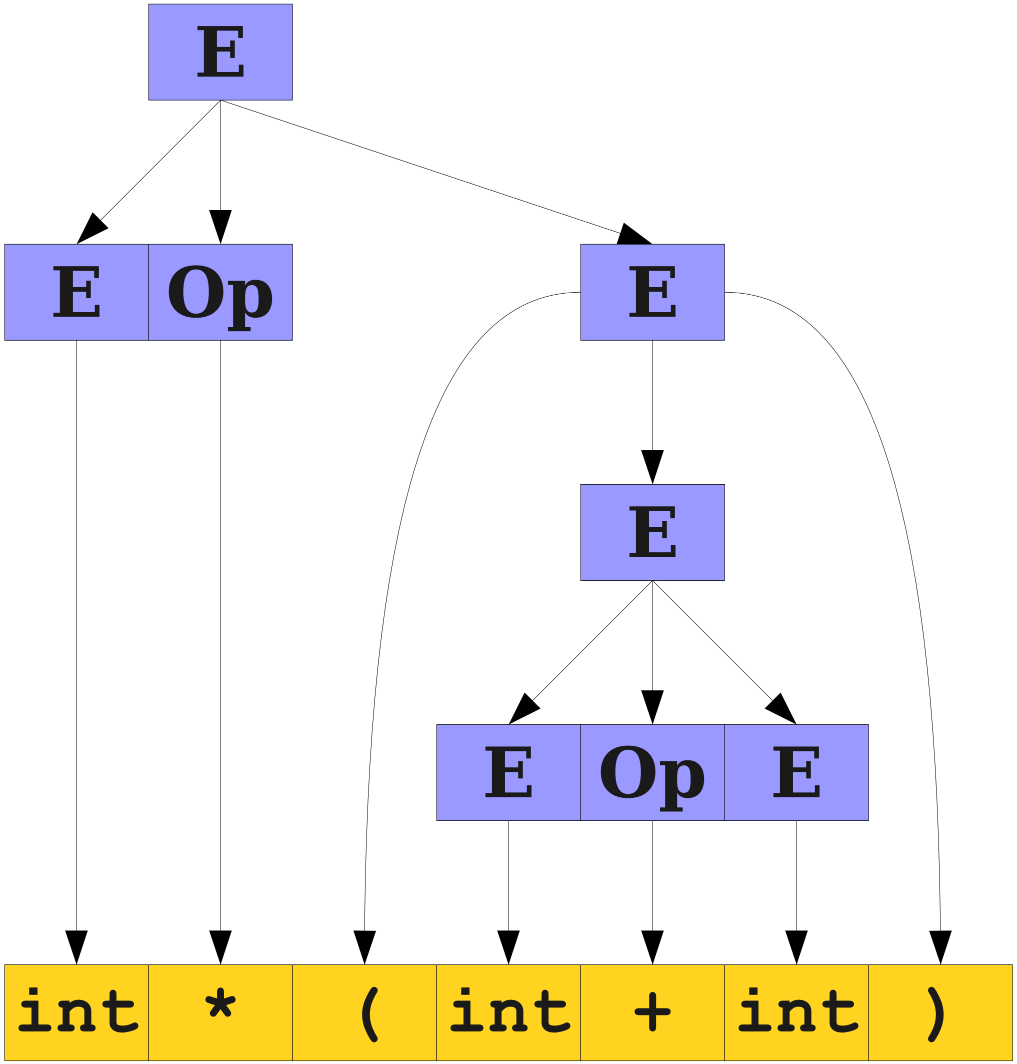
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)



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

Parse Trees

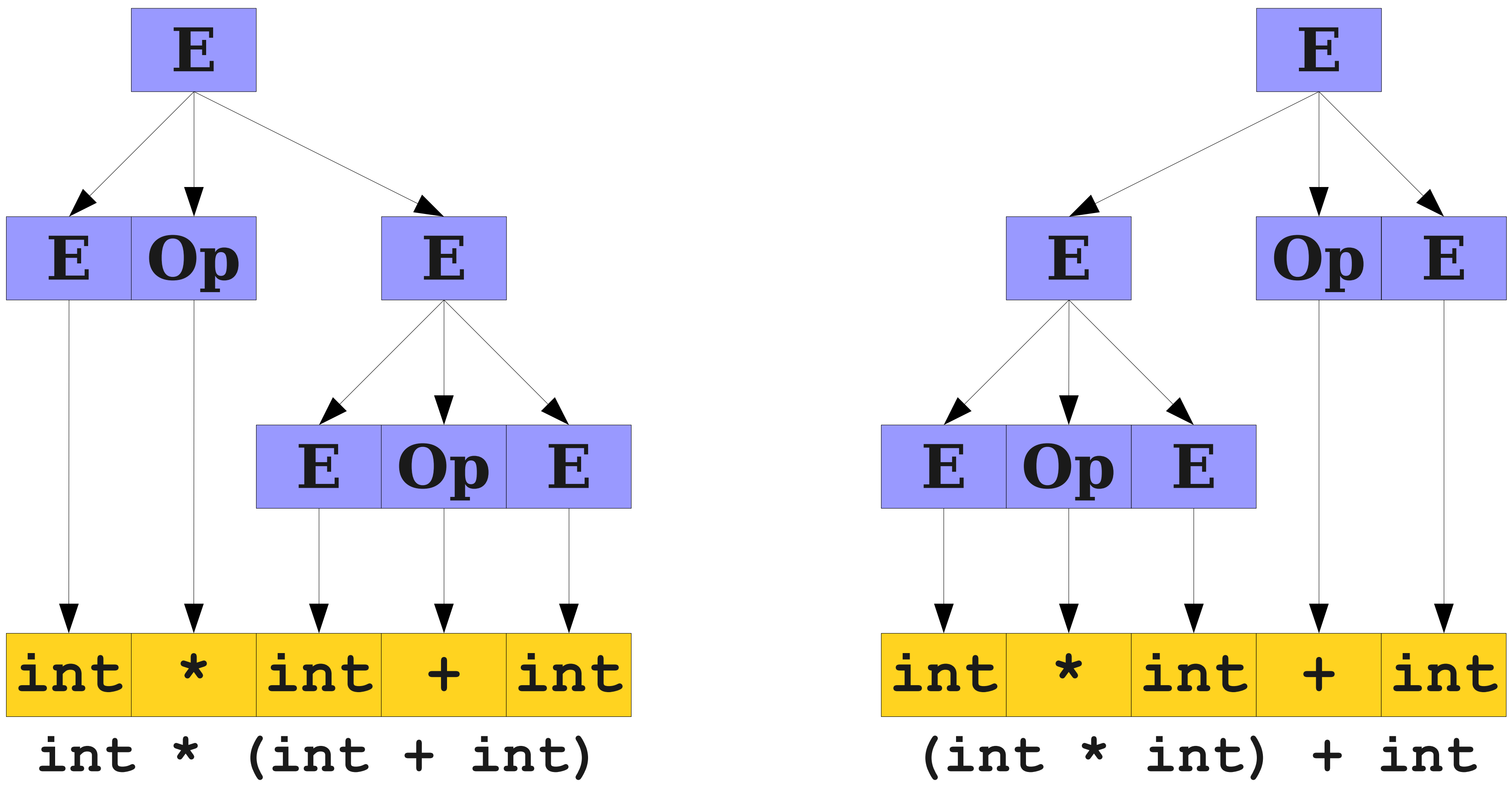
- A **parse tree** is a tree encoding the steps in a derivation.
- Each internal node is labeled with a nonterminal.
- Each leaf node is labeled with a terminal.
- Reading the leaves from left to right gives the string that was produced.

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:

  E → **E Op E** | **`int`**

  Op → **`+`** | **`-`** | **`*`** | **`/`**

- 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`

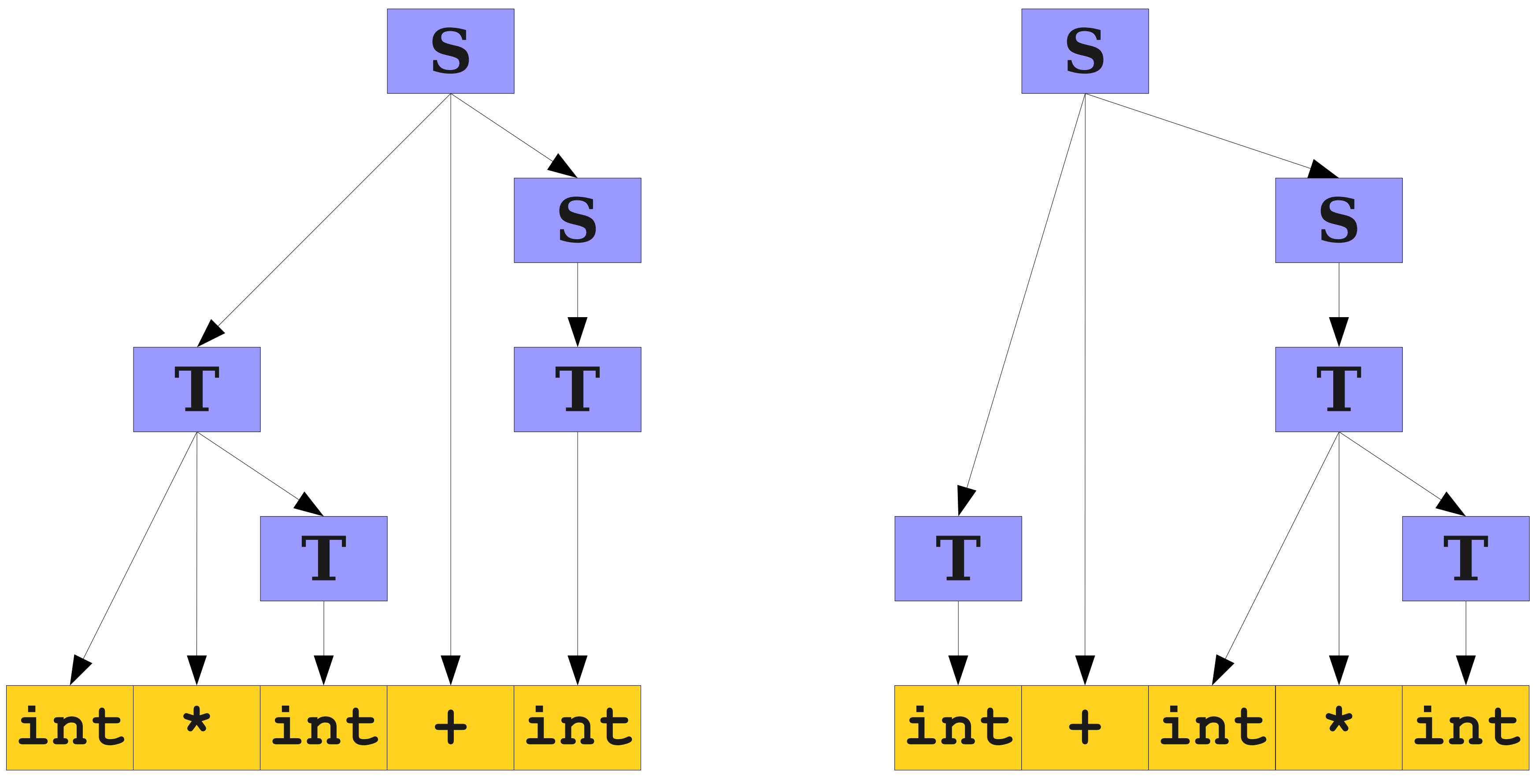
Rebuilding the Grammar

- 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:

$$\mathbf{S} \rightarrow \mathbf{T} \mid \mathbf{T} \; \texttt{+} \; \mathbf{S} \mid \mathbf{T} \; \texttt{-} \; \mathbf{S}$$

$$\mathbf{T} \rightarrow \texttt{int} \mid \texttt{int} \; \texttt{*} \; \mathbf{T} \mid \texttt{int} \; \texttt{/} \; \mathbf{T}$$

An Unambiguous Grammar



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

$T \rightarrow$ `int` $\mid$ `int` $*\ T \mid$ `int` $/\ T$

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?