CS103
Winter 2026


Lecture 19:

# Context-Free Grammars

# A Motivating Question

python3

```
>>> (137 + 42) - 2 * 3
173

>>> (60 + 37) + 5 * 8
137

>>> (200 / 2) + 6 / 2
103.0

>>>
```

# Mad Libs for Arithmetic

( ______ ______ ______ ) ______ ______ ______ ______

**Int Op Int Op Int Op Int**

This only lets us make arithmetic expressions of the form **( Int Op Int ) Op Int Op Int**.

What about arithmetic expressions that don't follow this pattern?

*Slide credit: Amy Liu*

# Recursive Mad Libs

( int / ( int + int ) )

Expr Op Expr Op Expr

What can an arithmetic expression be?

| | |
|---|---|
| `int` | A single number. |
| **Expr Op Expr** | Two expressions joined by an operator. |
| **(Expr)** | A parenthesized expression. |

A ***context-free grammar*** (or ***CFG***) is a recursive set of rules that define a language.

***(There's a bunch of specific requirements about what those rules can be; more on that in a bit.)***

# Arithmetic Expressions

- Here's how we might express the recursive rules from earlier as a CFG.

**Expr** → **int**
**Expr** → **Expr Op Expr**
**Expr** → **(Expr)**
**Op** → **+**
**Op** → **-**
**Op** → **×**
**Op** → **/**

This is called a *production rule*. It says "if you see **Expr**, you can replace it with **Expr Op Expr**."

# Arithmetic Expressions

- Here's how we might express the recursive rules from earlier as a CFG.

**Expr** → `int`

**Expr** → **Expr Op Expr**

**Expr** → `(`**Expr**`)`

**Op** → `+`

**Op** → `-`

**Op** → `×`

**Op** → `/`

This one says "if you see **Op**, you can replace it with `-`."

# Arithmetic Expressions

- Here's how we might express the recursive rules from earlier as a CFG.

**Expr** → `int`
**Expr** → **Expr Op Expr**
**Expr** → `(`**Expr**`)`
**Op** → `+`
**Op** → `-`
**Op** → `×`
**Op** → `/`

**Expr**
⇒ **Expr Op Expr**
⇒ **Expr Op** `int`
⇒ `int` **Op** `int`
⇒ `int / int`

These red symbols are called ***nonterminals***. They're placeholders that get expanded later on.

# Arithmetic Expressions

- Here's how we might express the recursive rules from earlier as a CFG.

**Expr** → `int`
**Expr** → **Expr Op Expr**
**Expr** → `(`**Expr**`)`
**Op** → `+`
**Op** → `-`
**Op** → `×`
**Op** → `/`

**Expr**
⇒ **Expr Op Expr**
⇒ **Expr Op** `int`
⇒ `int` **Op** `int`
⇒ `int / int`

The symbols in blue monospace are **`terminals`**. They're the final characters used in the string and never get replaced.

# Arithmetic Expressions

- Here's how we might express the recursive rules from earlier as a CFG.

**Expr** → `int`

**Expr** → **Expr Op Expr**

**Expr** → `(`**Expr**`)`

**Op** → `+`

**Op** → `-`

**Op** → `×`

**Op** → `/`

**Expr**
⇒ **Expr Op Expr**
⇒ **Expr Op** `(`**Expr**`)`
⇒ **Expr Op** `(`**Expr Op Expr**`)`
⇒ **Expr** `×` `(`**Expr Op Expr**`)`
⇒ `int` `×` `(`**Expr Op Expr**`)`
⇒ `int` `×` `(int` **Op Expr**`)`
⇒ `int` `×` `(int` **Op** `int)`
⇒ `int` `×` `(int + int)`

# Context-Free Grammars

- Formally, a context-free grammar is a collection of four items:
  - 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 replaced by a string of terminals and nonterminals, and
  - a ***start symbol*** (which must be a nonterminal) that begins the derivation. By convention, the start symbol is the one on the left-hand side of the first production.

**Expr** → `int`

**Expr** → **Expr Op Expr**

**Expr** → `(`**Expr**`)`

**Op** → `+`

**Op** → `-`

**Op** → `×`

**Op** → `/`

# Some CFG Notation

- In today’s slides, capital letters in **Bold Red Uppercase** will represent nonterminals.
  - e.g. **A**, **B**, **C**, **D**
- Lowercase letters in `blue monospace` will represent terminals.
  - e.g. `t`, `u`, `v`, `w`
- Lowercase Greek letters in ***gray italics*** will represent arbitrary strings of terminals and nonterminals.
  - e.g. $\alpha$, $\gamma$, $\omega$
- You don't need to use these conventions on your own; just make sure whatever you do is readable.

# A Notational Shorthand

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

**Op** → `+` | `-` | `×` | `/`

# Derivations

Expr
⇒ Expr Op Expr
⇒ Expr Op (Expr)
⇒ Expr Op (Expr Op Expr)
⇒ Expr × (Expr Op Expr)
⇒ int × (Expr Op Expr)
⇒ int × (int Op Expr)
⇒ int × (int Op int)
⇒ int × (int + int)

- A sequence of zero or more 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 that

  **Expr** $\Rightarrow^*$ `int × (int + int)`.

| |
|---|
| **Expr** → `int` \| **Expr Op Expr** \| `(`**Expr**`)` |
| **Op** → `+` \| `-` \| `×` \| `/` |

# 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 of terminals derivable from the start symbol.

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

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

Consider the following CFG $G$ over Σ = {**a**, **b**, **c**, **d**}:

**Q** → **Qa** | **dH**
**H** → **bHb** | **c**

Which of the following strings are in $\mathscr{L}(G)$?

**dca**
**dc**
**cad**
**bcb**
**dHaa**

Answer at ***https://cs103.stanford.edu/pollev***

# Context-Free Languages

- A language $L$ is called a ***context-free language*** (or CFL) if there is a CFG $G$ such that $L = \mathscr{L}(G)$.
- Questions:
  - How are context-free and regular languages related?
  - How do we design context-free grammars for context-free languages?

# CFGs and Regular Expressions

- CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.
- You can use the symbols * and ∪ if you'd like in a CFG, but they just stand for themselves.
- Consider this CFG $G$:

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

- Here, $\mathscr{L}(G) = \{\texttt{a*b}\}$ and has cardinality one. That is, $\mathscr{L}(G) \neq \{\ \texttt{a}^n\texttt{b} \mid n \in \mathbb{N}\ \}$.

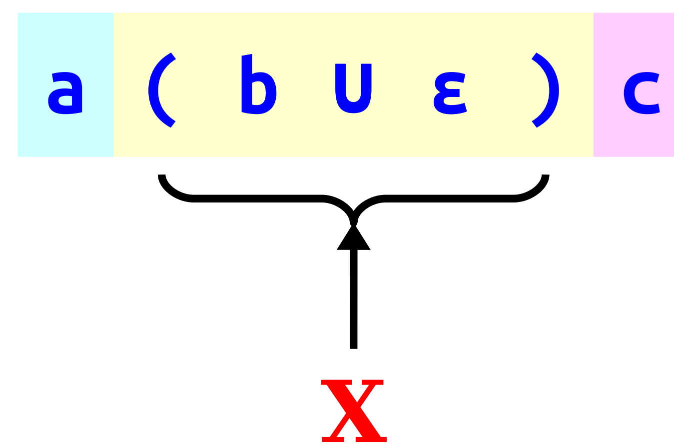
# CFGs and Regular Expressions

- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

**S** → a**X**c

**X** → b | ε

It’s totally fine for a production to replace a nonterminal with the empty string.



# CFGs and Regular Expressions

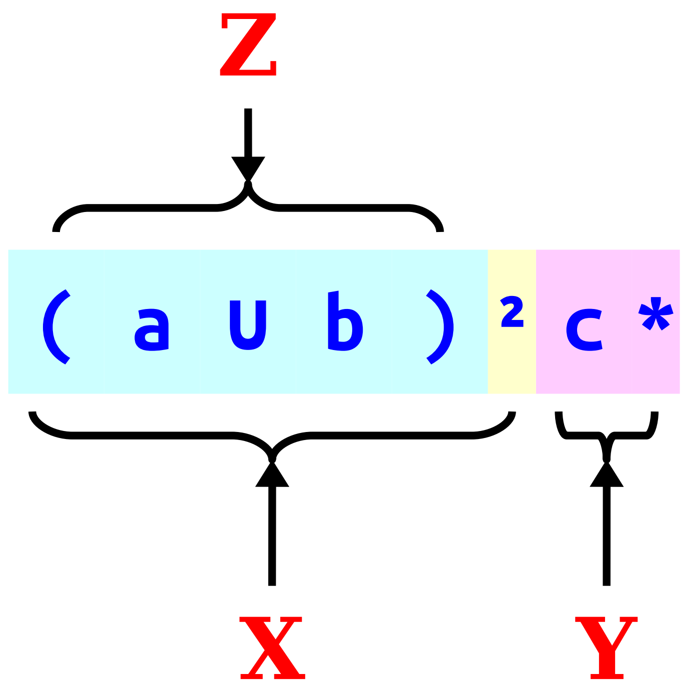
- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

**S** → **XY**

**X** → **ZZ**

**Z** → a | b

**Y** → c**Y** | ε



# The Language of a Grammar

- Consider the following CFG $G$:

$$\mathbf{S} \rightarrow \mathbf{aSb} \mid \boldsymbol{\varepsilon}$$

- What strings can this generate?

| a | a | a | a | b | b | b | b |
|---|---|---|---|---|---|---|---|

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


Regular
Languages
CFLs
All Languages

# 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.
  - ***Store information in nonterminals:*** Have each nonterminal correspond to some useful piece of information.
- Check our online “Guide to CFGs” for more information about CFG design.
- We’ll hit the highlights in the rest of this lecture.

# 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 $\omega$ is a palindrome, then **a**$\omega$**a** and **b**$\omega$**b** are palindromes.
  - No other strings 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 braces $\}$
- Some sample strings in $L$:

**{{{}}}**

**{{}}{}**

**{{}{}}{{}{}}**

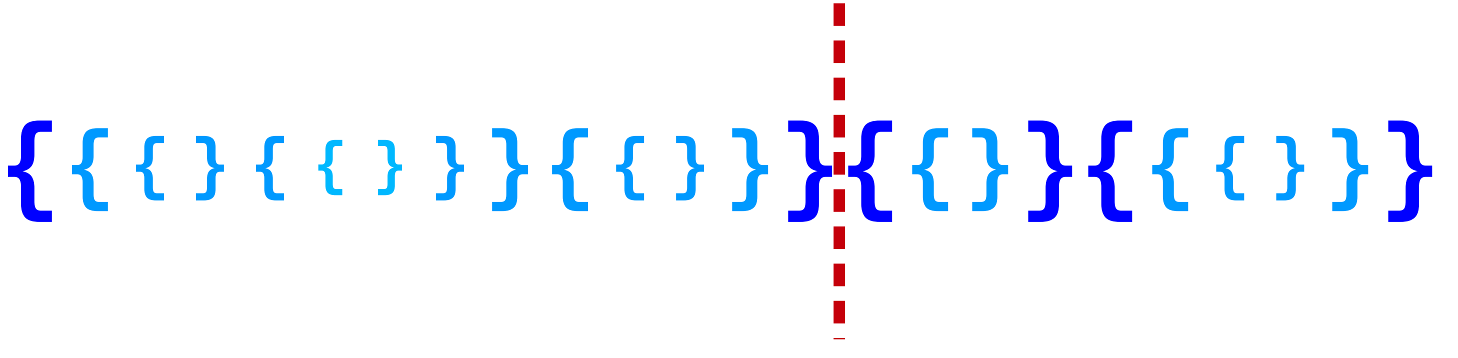
**{{{{{}}}{{}}}}**

**ε**

**{}{}**

# Designing CFGs

- Let $\Sigma = \{$**{**, **}**$\}$ and let $L = \{ w \in \Sigma^* \mid w$ is a string of balanced braces $\}$
- Let's think about this recursively.
  - Base case: the empty string is a string of balanced braces.
  - Recursive step: Look at the closing brace that matches the first open brace.



# Designing CFGs

- Let $\Sigma = \{$**{**, **}**$\}$ and let $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces $\}$
- Let's think about this recursively.
  - Base case: the empty string is a string of balanced braces.
  - Recursive step: Look at the closing brace that matches the first open brace. Removing the first brace and the matching brace forms two new strings of balanced braces.

**S → {S}S | ε**

# Designing CFGs

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid w$ has the same number of **a**'s and **b**'s $\}$

Which of these CFGs have language $L$?

| | |
|---|---|
| **S** → **aSb** \| **bSa** \| **ε** | **S** → **abS** \| **baS** \| **ε** |
| **S** → **abSba** \| **baSab** \| **ε** | **S** → **SbaS** \| **SabS** \| **ε** |

Answer at
*https://cs103.stanford.edu/pollev*

# 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 (most likely) design your own CFG for this language on Problem Set 8.

# CFG Caveats II

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

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

- What strings in {**a**, **b**}* can you derive?
  - Answer: ***None!***
- What is the language of the grammar?
  - Answer: **Ø**
- When designing CFGs, make sure your recursion actually terminates!

# Designing CFGs

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

$$\mathbf{S} \rightarrow \mathbf{X} \stackrel{?}{=} \mathbf{X}$$

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

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

# Finding a Build Order

- Let $\Sigma = \{$a, $\overset{?}{=}\}$ and let $L = \{$a$^n\overset{?}{=}$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.
  - If we build the strings of a's independently of one another, then we can't enforce that they have the same length.
  - ***Idea:*** Build both strings of a's at the same time.
- Here's one possible grammar based on that idea:

$$S \to \overset{?}{=} \mid aSa$$

$$\begin{aligned} & S \\ \Rightarrow\ & aSa \\ \Rightarrow\ & aaSaa \\ \Rightarrow\ & aaaSaaa \\ \Rightarrow\ & aaa\overset{?}{=}aaa \end{aligned}$$

# Summary of CFG Design Tips

- Look for recursive structures where they exist: they can help guide you toward a solution.
- Keep the build order in mind – often, you'll build two totally different parts of the string concurrently.
  - Usually, those parts are built in opposite directions: one's built left-to-right, the other right-to-left.
- Use different nonterminals to represent different structures.

# Applications of Context-Free Grammars

# 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
         |  …
```

# Grammars in Compilers

- One of the key steps in a compiler is figuring out what a program “means.”
- This is usually done by defining a grammar showing the high-level structure of a programming language.
- There are certain classes of grammars (LL(1) grammars, LR(1) grammars, LALR(1) grammars, etc.) for which it's easy to figure out how a particular string was derived.
- Tools like `yacc` or `bison` automatically generate parsers from these grammars.
- Curious to learn more? ***Take CS143!***

# Natural Language Processing

- By building context-free grammars for actual languages and applying statistical inference, it's possible for a computer to recover the likely meaning of a sentence.
  - In fact, CFGs were first called ***phrase-structure grammars*** and were introduced by Noam Chomsky in his seminal work *Syntactic Structures*.
  - They were then adapted for use in the context of programming languages, where they were called ***Backus-Naur forms***.
- The ***Stanford Parser*** project is one place to look for an example of this.
- Want to learn more? Take CS124 or CS224N!

# Next Time

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