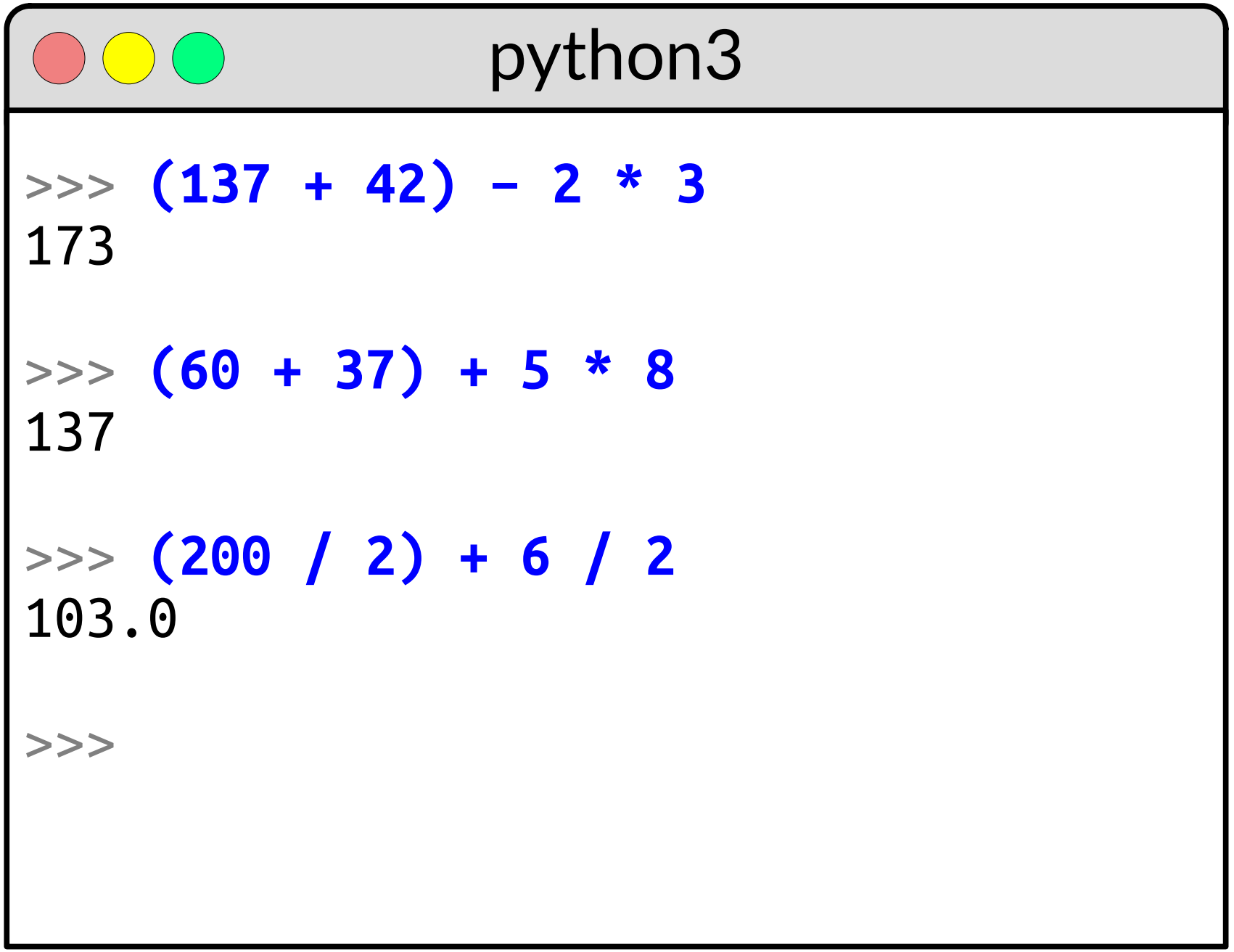
Context-Free Grammars

A Motivating Question


python3
>>> (137 + 42) - 2 * 3
173
>>> (60 + 37) + 5 * 8
137
>>> (200 / 2) + 6 / 2
103.0
>>>

Mad Libs for Arithmetic

(	26	+	42	)	*	2	+	1
	Int	Op	Int		Op	Int	Op	Int

Slide credit: Amy Liu

Mad Libs for Arithmetic

(	7	*	5	)	/	5	-	49
	Int	Op	Int		Op	Int	Op	Int

Slide credit: Amy Liu

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

What can an arithmetic expression be?

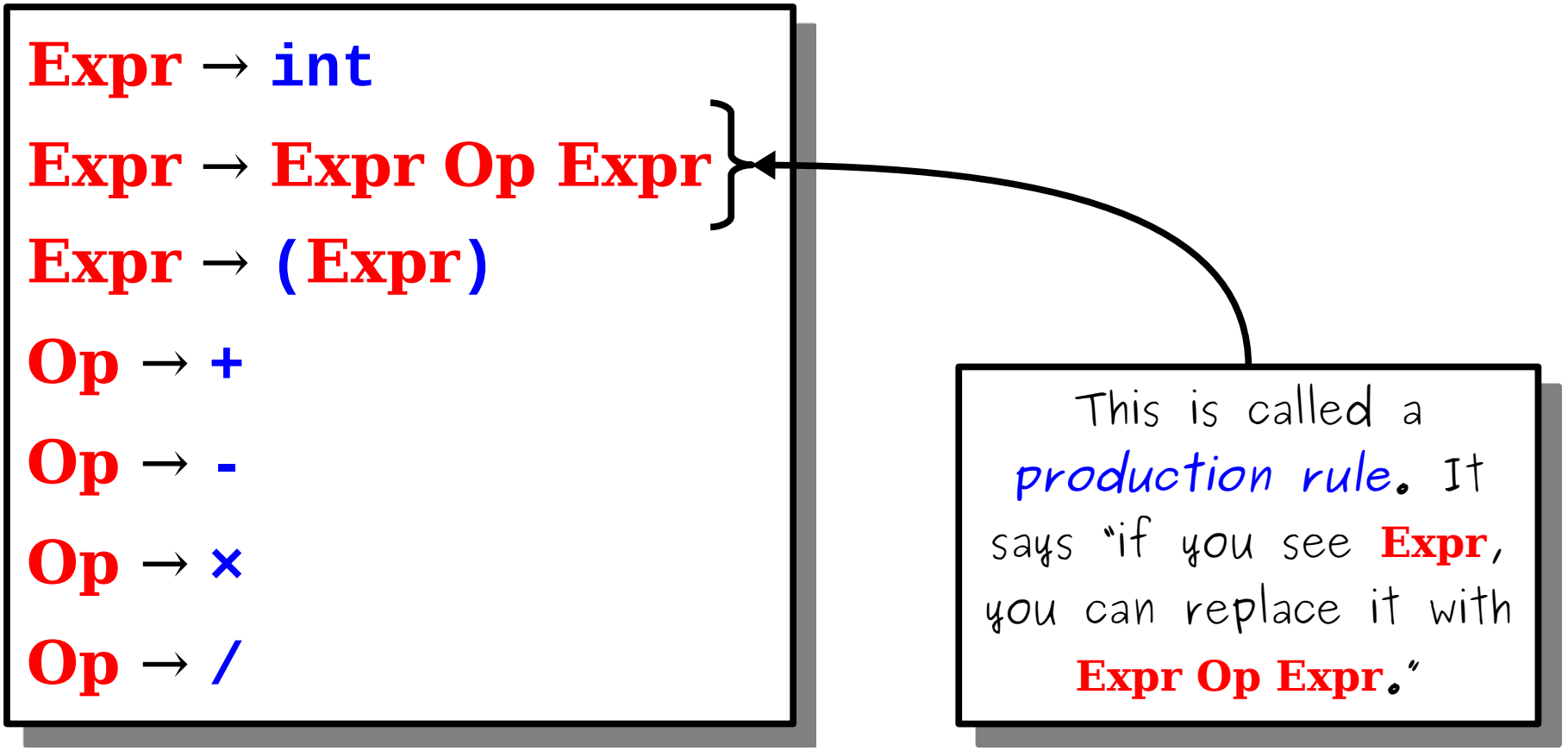
`int`	A single number.
Expr Op Expr	Two expressions joined by an operator.
(Expr)	A parenthesized expression.

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.



Arithmetic Expressions

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

Expr → `int`

Expr → **Expr Op Expr**

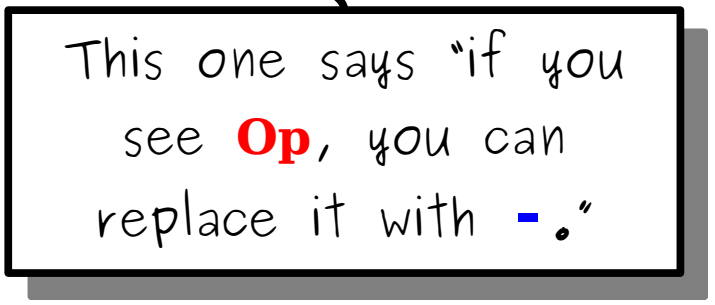
Expr → `(` **Expr** `)`

Op → `+`

Op → `-`

Op → `×`

Op → `/`



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`

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. α, γ, ω
- 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 → **Expr Op Expr**

Expr → `(`**Expr**`)`

Op → `+`

Op → `-`

Op → `×`

Op → `/`

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 α derives string ω, 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

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

- That is, $\mathcal{L}(G)$ 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

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

Consider the following CFG G over $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$:

$$\mathbf{S} \to \mathbf{S}\mathtt{a} \mid \mathtt{d}\mathbf{T}$$
$$\mathbf{T} \to \mathtt{b}\mathbf{T}\mathtt{b} \mid \mathtt{c}$$

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

dca
dc
cad
bcb
dTaa

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?

Time-Out for Announcements!

Problem Set Seven

- Problem Set Six was due today at 2:30PM.
- Problem Set Seven goes out today. It's due next Friday at 2:30PM.
  - It's all about regular expressions, properties of regular languages, and gives a first glimpse at nonregular languages.
  - We realistically don't expect you to look at this until Tuesday, after you've finished the midterm.
  - We've made this problem set smaller than usual to account for the exam.

Second Midterm Logistics

- Our second midterm exam is next ***Monday, November 14th*** from ***7PM – 10PM***. It'll be held here, ***Hewlett 200***.
- Topic coverage is primarily lectures 06 – 13 (functions through induction) and PS3 – PS5. Finite automata and onward won't be tested here.
  - Because the material is cumulative, topics from PS1 – PS2 and Lectures 00 – 05 are also fair game.
- The exam is closed-book and closed-computer. You can bring one double-sided 8.5" × 11" sheet of notes with you.
- Extra Practice Problems 2 is available on the course website if you want to get more practice with these topics.
- ***We want you to do well on this exam***. Keep in touch and let us know what we can do to help make that happen!
- And always, keep the TAs in the loop! Let us know what we can do to help out.

Our Advice

- ***Stay fed and rested.*** You are not a brain in a jar. You are a rich, complex, beautiful human being. Please take care of yourself.
- ***Read all questions before diving into them.*** You don’t have to go sequentially. Read over each problem so you know what to expect, then pick whichever one looks easiest and start there.
- ***Reflect on how far you’ve come.*** How many of these questions would you have been able to *understand* two months ago? That’s the mark that you’re learning something!

Three Questions

- What’s something you know now that, at the start of the quarter, you knew you didn’t know?
- What’s something you know now that, at the start of the quarter, you *didn’t* know you didn’t know?
- What’s something you *don’t* know now that, at the start of the quarter, you *didn’t* know you didn’t know?

Your Questions

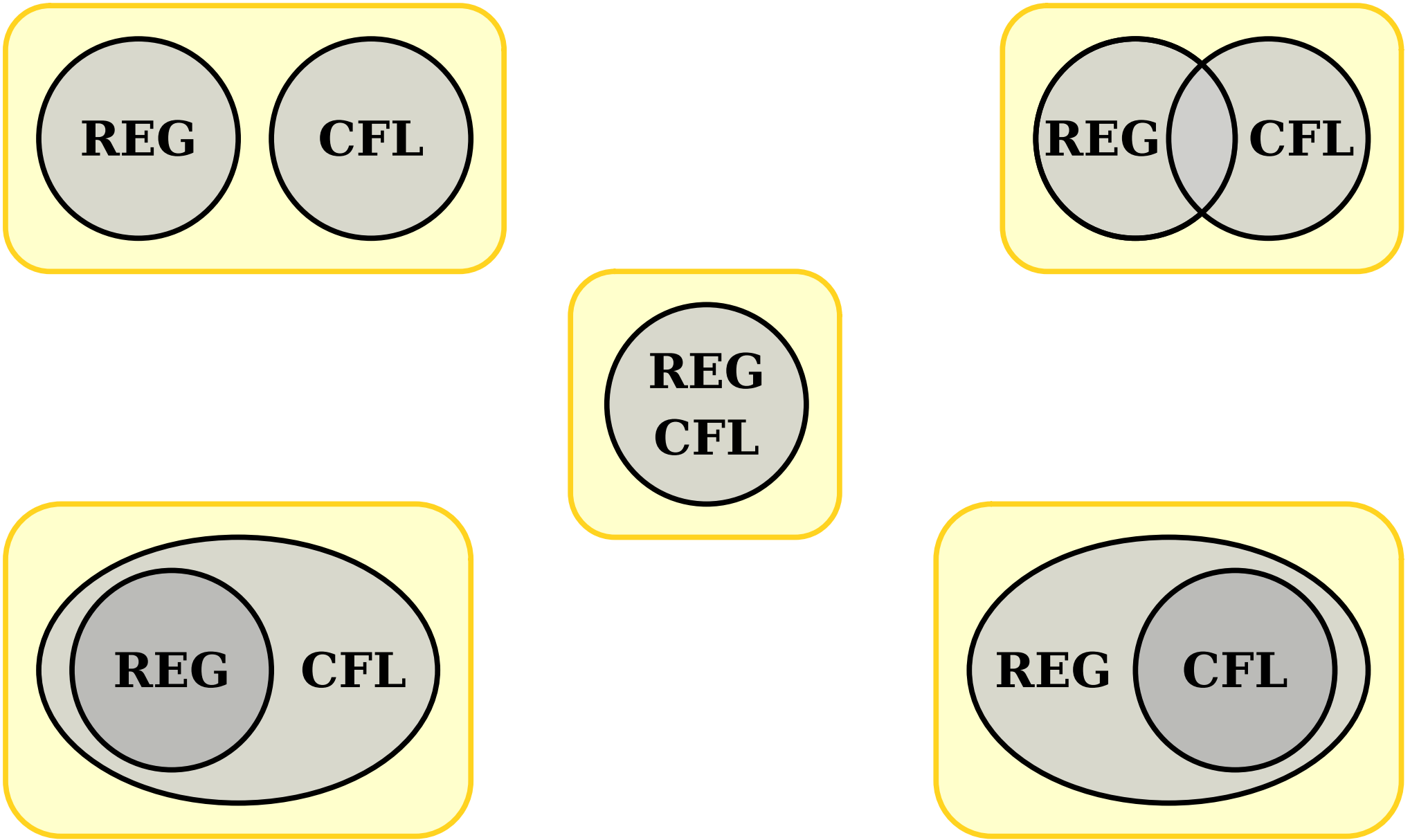
“What have been some of the most important realizations you've had that altered the way you view life and your place in the universe?”

The biggest one for me has been learning more about history and seeing just how constant the human condition is. Reading Seneca's "On the Brevity of Life," from 2,000 years ago gives an impression of how universal so many intensely personal moments can be!

Back to CS103!

Regular and Context-Free Languages

Five Possibilities



CFGs and Regular Expressions

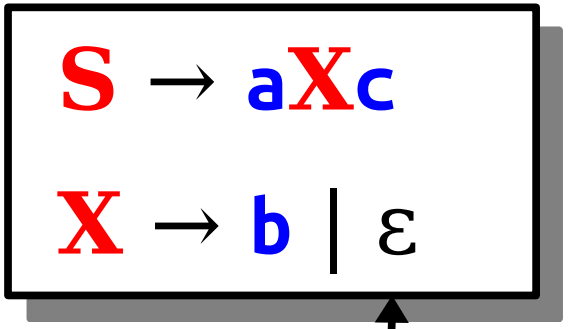
- CFGs consist purely of production rules of the form **A** → ω. They do not have the regular expression operators * or ∪.
- You can use the symbols * and ∪ if you'd like in a CFG, but they just stand for themselves.
- Consider this CFG G:

S → **a*b**

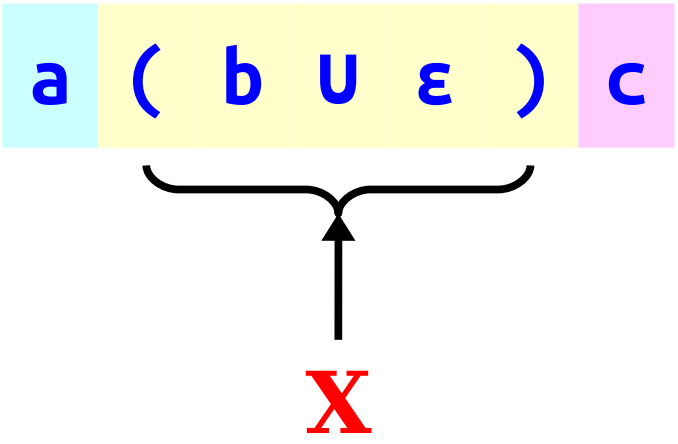
- Here, $\mathcal{L}(G) = \{$**a*b**$\}$ and has cardinality one. That is, $\mathcal{L}(G) \neq \{$ **a**^{n}**b** $\mid n \in \mathbb{N}\}$.

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.



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



CFGs and Regular Expressions

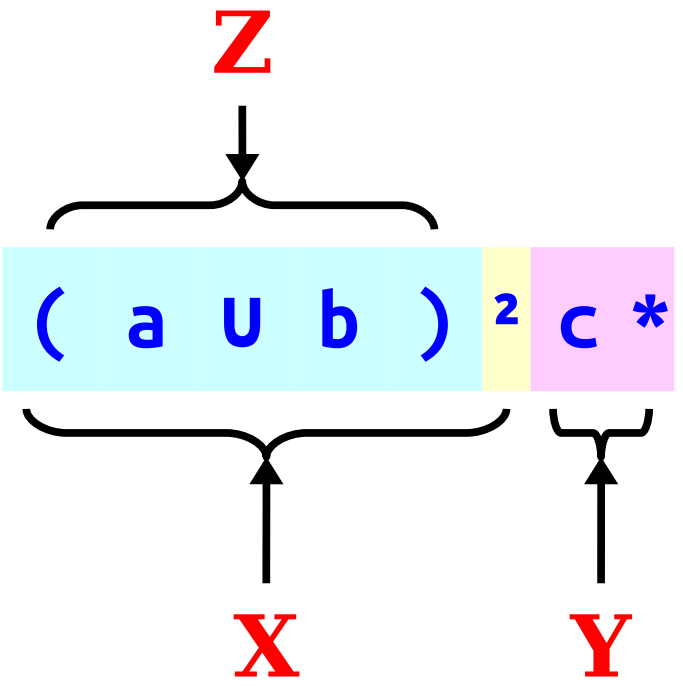
- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

$$S \to XY$$

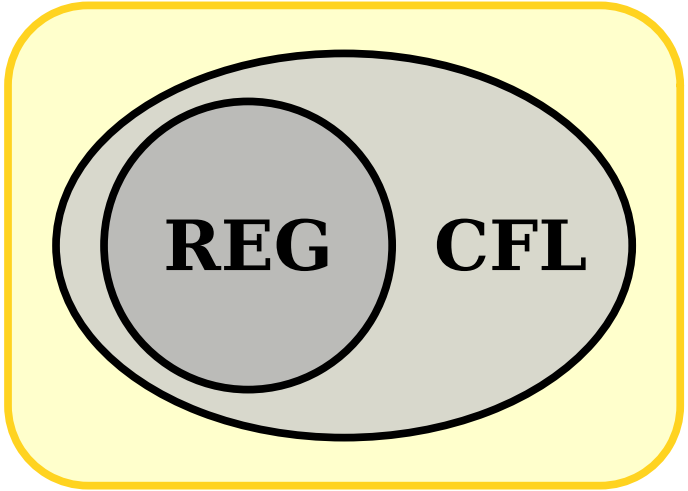
$$X \to ZZ$$

$$Z \to a \mid b$$

$$Y \to cY \mid \varepsilon$$



Two ~~Five~~ Possibilities



The Language of a Grammar

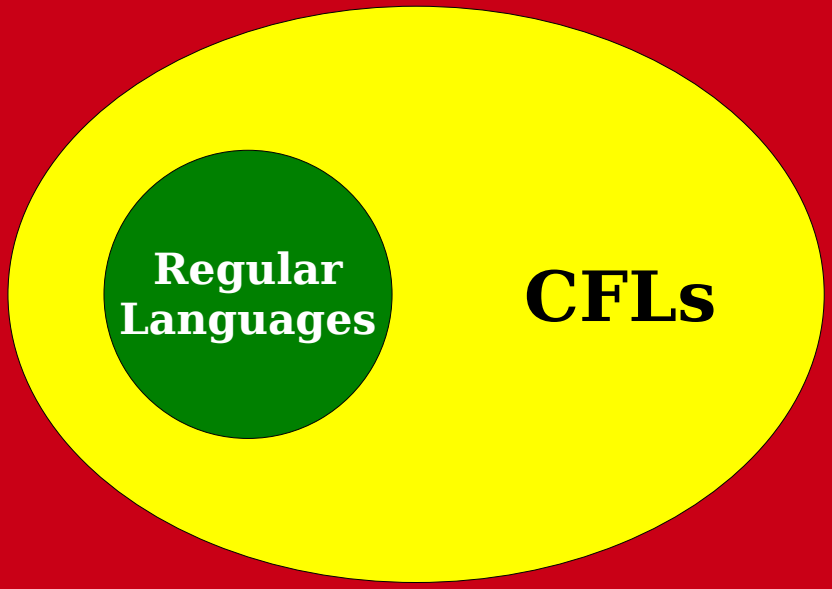
- Consider the following CFG G:

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

- What strings can this generate?

a	**a**	**a**	**a**	**b**	**b**	**b**	**b**

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



All Languages

Why the Extra Power?

- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded "memory."

$$S \rightarrow aSb \mid \varepsilon$$

a	a	a	a	S	b	b	b	b

Designing CFGs

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 ω is a palindrome, then **a**ω**a** and **b**ω**b** are palindromes.
  - No other strings are palindromes.

S → **ε** | **a** | **b** | **a****S****a** | **b****S****b**

Designing CFGs

- Let $\Sigma = \{$**{**, **}**$\}$ and let $L = \{w \in \Sigma^* \mid w$ is a string of balanced braces $\}$
- Some sample strings in L:

{{{}}}

{{}}{}

{{}{}}{{}{}}

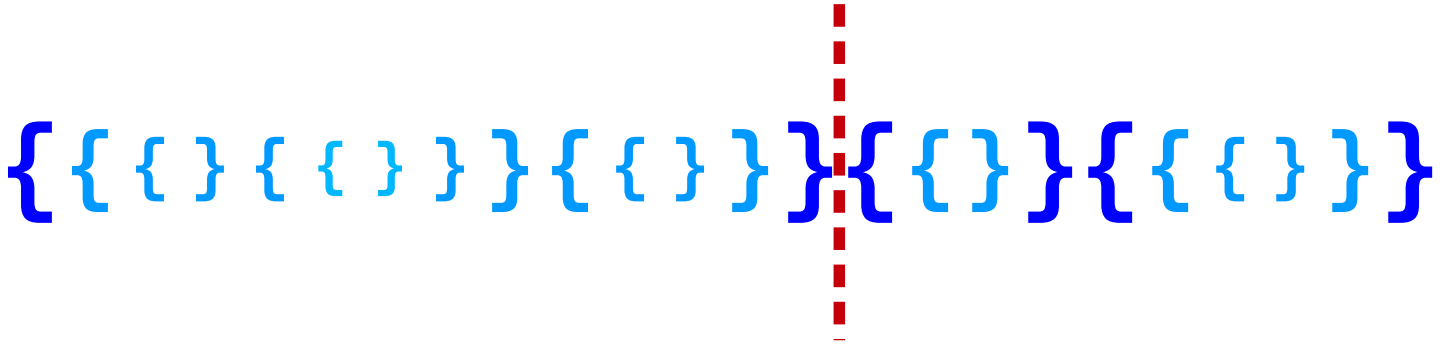
{{{{}}}{{}}}

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 $\}$

How many of the following CFGs have language L?

S → **a****S****b** | **b****S****a** | **ε**

S → **ab****S** | **ba****S** | **ε**

S → **ab****S****ba** | **ba****S****ab** | **ε**

S → **S****ba****S** | **S****ab****S** | **ε**

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 Problem Set 8.

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

- ***No Class on Monday***
  - There's a midterm, so we're giving you the day off.
- ***Then, On Wednesday...***
  - ***Turing Machines***
    - What does a computer with unbounded memory look like?
    - How would you program it?