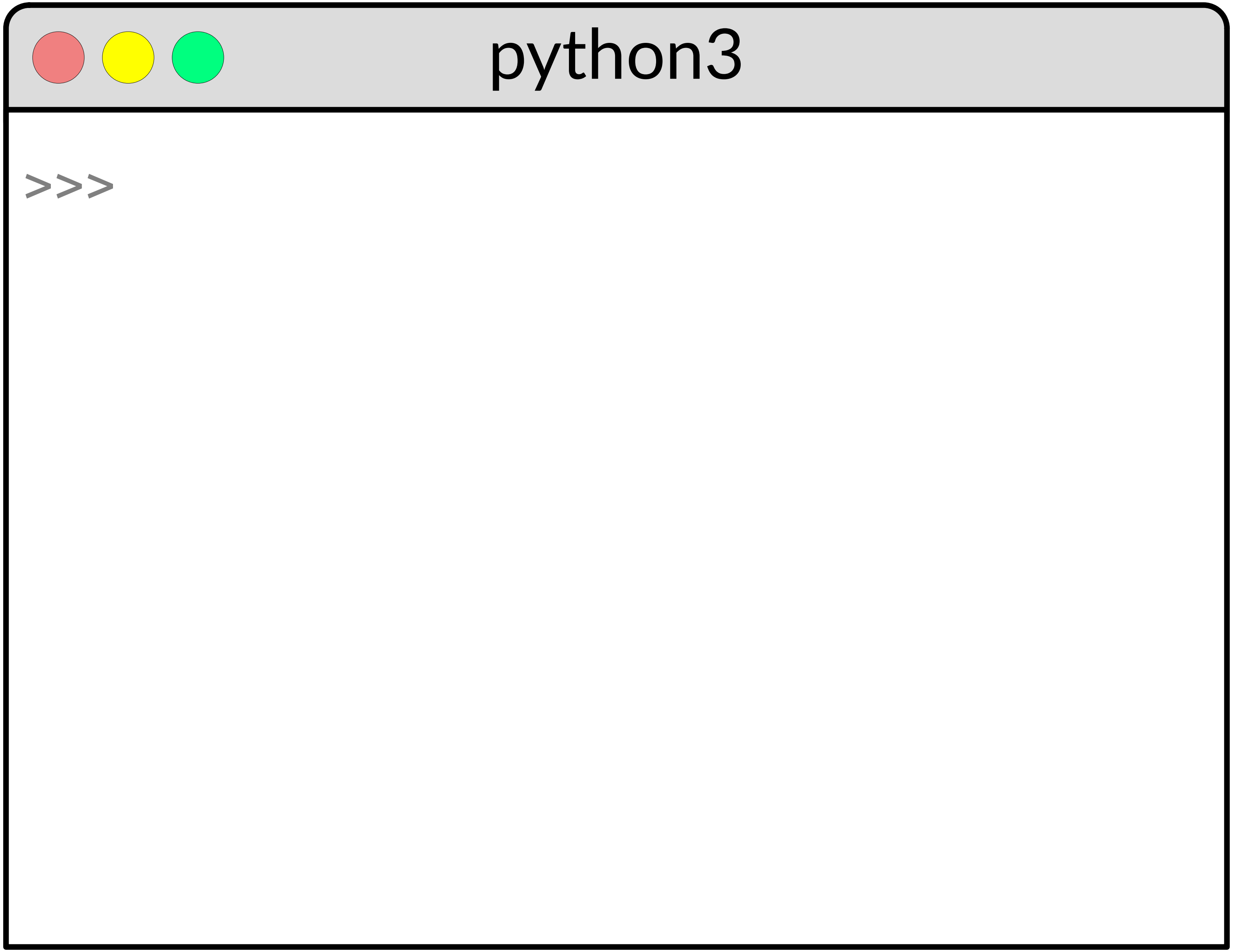
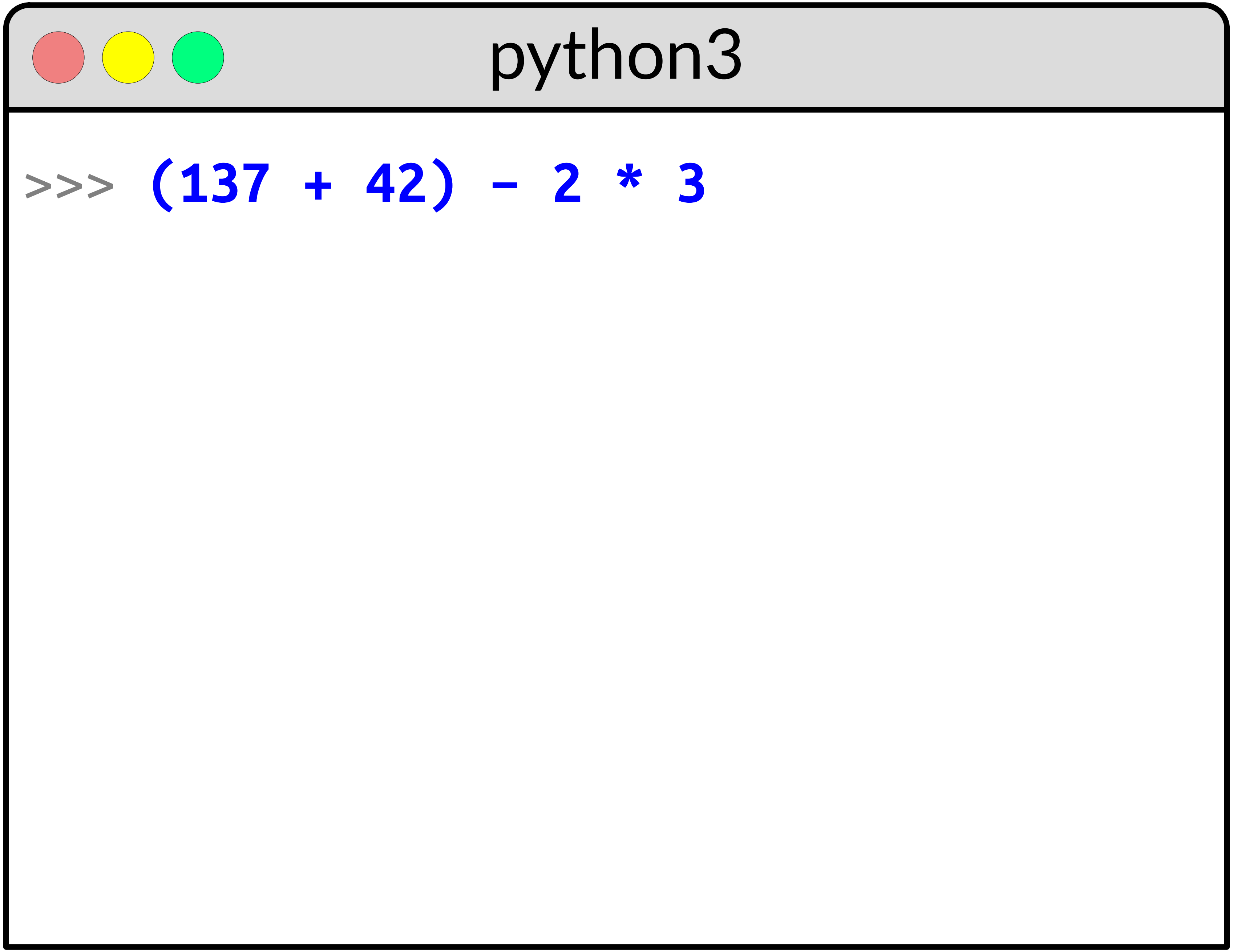
Context-Free Grammars

A Quick Update

Context-Free Grammars

A Motivating Question


python3
>>>


python3
>>> (137 + 42) - 2 * 3

python3

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

>>>
```

python3

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

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

python3

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

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

>>>
```

python3

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

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

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

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

Slide credit: Amy Liu

Mad Libs for Arithmetic

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

Slide credit: Amy Liu

Mad Libs for Arithmetic

(______ ______ ______) ______ ______ ______ ______

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

Expr

Recursive Mad Libs

______

Expr

What can an arithmetic expression be?

Recursive Mad Libs

$$\frac{\texttt{int}}{\textbf{Expr}}$$

What can an arithmetic expression be?

`int`	A single number.

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number.

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number.
Expr Op Expr	Two expressions joined by an operator.

Recursive Mad Libs

______ ______ ______

Expr **Op** **Expr**

What can an arithmetic expression be?

`int`	A single number.
Expr Op Expr	Two expressions joined by an operator.

Recursive Mad Libs

`int`

Expr **Op** **Expr**

What can an arithmetic expression be?

`int`	A single number.
Expr Op Expr	Two expressions joined by an operator.

Recursive Mad Libs

`int`	`+`	
Expr	**Op**	**Expr**

What can an arithmetic expression be?

`int`	A single number.
Expr Op Expr	Two expressions joined by an operator.

Recursive Mad Libs

int	+	
Expr	Op	Expr

What can an arithmetic expression be?

`int`	A single number.
Expr Op Expr	Two expressions joined by an operator.

Recursive Mad Libs

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.

Recursive Mad Libs

`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.

Recursive Mad Libs

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.

Recursive Mad Libs

`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.

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.

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number.
Expr Op Expr	Two expressions joined by an operator.

Recursive Mad Libs

Expr

What can an arithmetic expression be?

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

Recursive Mad Libs

(______ **)**

Expr

What can an arithmetic expression be?

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

Recursive Mad Libs

(______)

Expr

What can an arithmetic expression be?

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

Recursive Mad Libs

(_____ _____ _____)

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.

Recursive Mad Libs

(______ ______ ______)

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.

Recursive Mad Libs

(`int` ______ ______)

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.

Recursive Mad Libs

(int /)

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.

Recursive Mad Libs

`( int / )`

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.

Recursive Mad Libs

(int / ( ))

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.

Recursive Mad Libs

(int / (____))

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.

Recursive Mad Libs

(int / ( ))

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.

Recursive Mad Libs

`( 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.

Recursive Mad Libs

`( 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.

Recursive Mad Libs

(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.

Recursive Mad Libs

(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.

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`

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 $\boldsymbol{\alpha}$ derives string $\boldsymbol{\omega}$, we write $\boldsymbol{\alpha} \Rightarrow^* \boldsymbol{\omega}$.
- 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

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

Consider the following CFG G over $\Sigma = \{$**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?

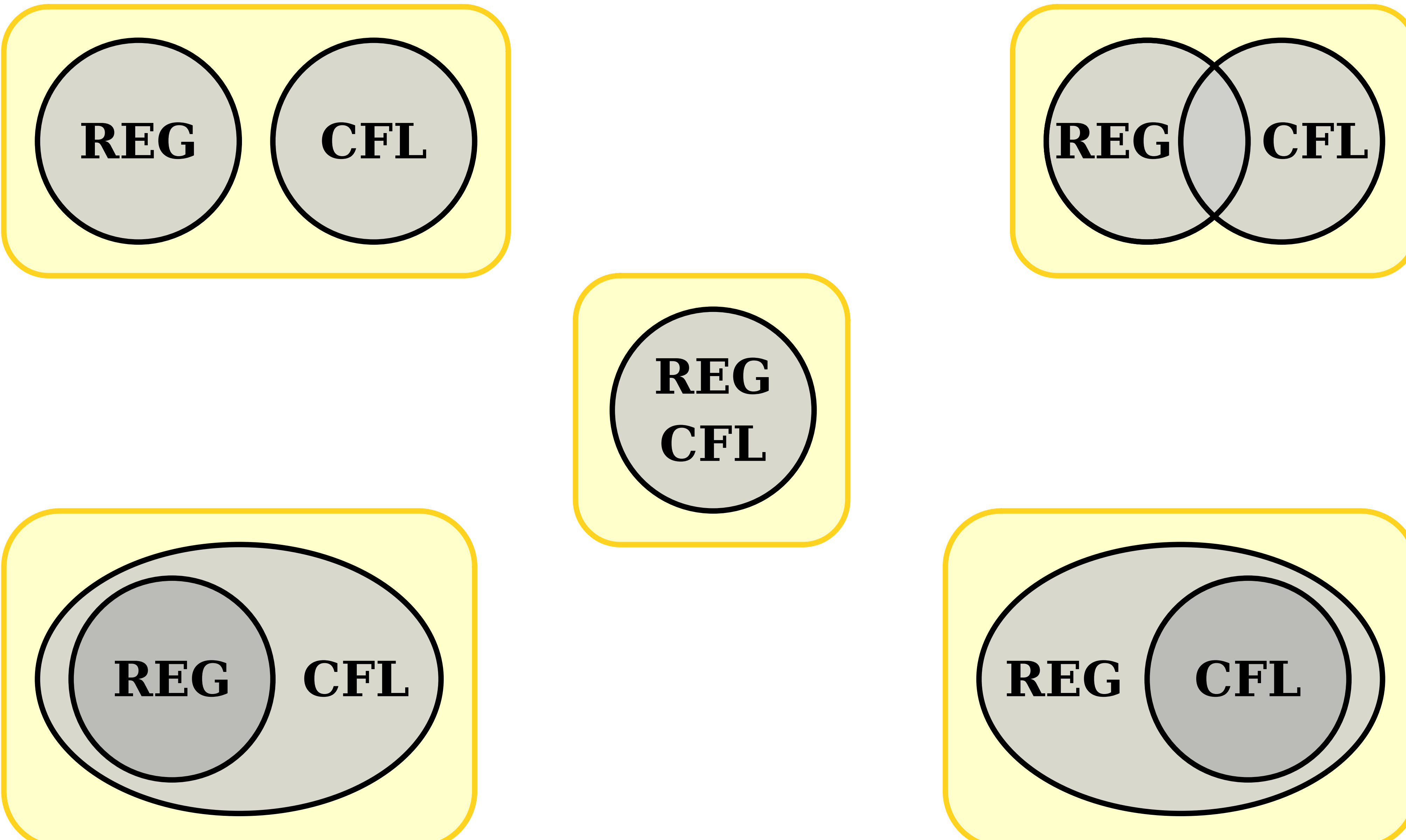
Context-Free Languages

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?

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:

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

- 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.

a (b ∪ ε) c

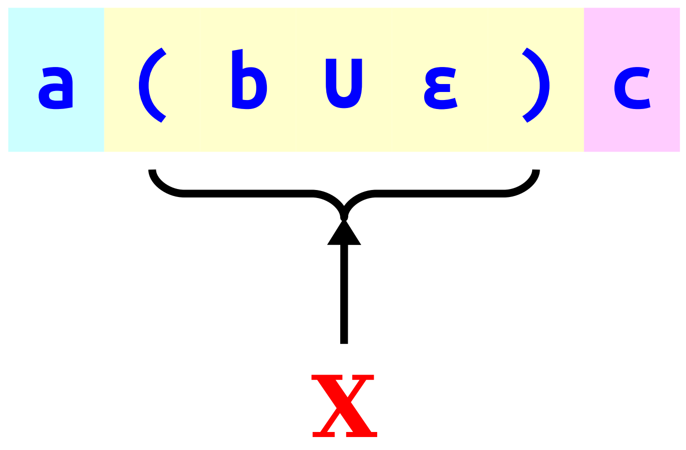
CFGs and Regular Expressions

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

$a(b \cup \varepsilon)c$

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.



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.

$$S \to aXc$$

$$a\underbrace{(b \cup \varepsilon)}_{X}c$$

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.

$$S \rightarrow aXc$$
$$X \rightarrow b \mid \varepsilon$$

$$a\underbrace{(\ b\ \cup\ \varepsilon\ )}_{X}c$$

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.

$\mathtt{a}\underbrace{(\ \mathtt{b} \cup \varepsilon\ )}_{X}\mathtt{c}$

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.

$$(a \cup b)^2 c^*$$

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.

$(a \cup b)^2 c^*$

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.

$$\underbrace{(a \cup b)^2}_{X}\,\underbrace{c^*}_{Y}$$

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**

$$\underbrace{(a \cup b)^2}_{X}\underbrace{c^*}_{Y}$$

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.

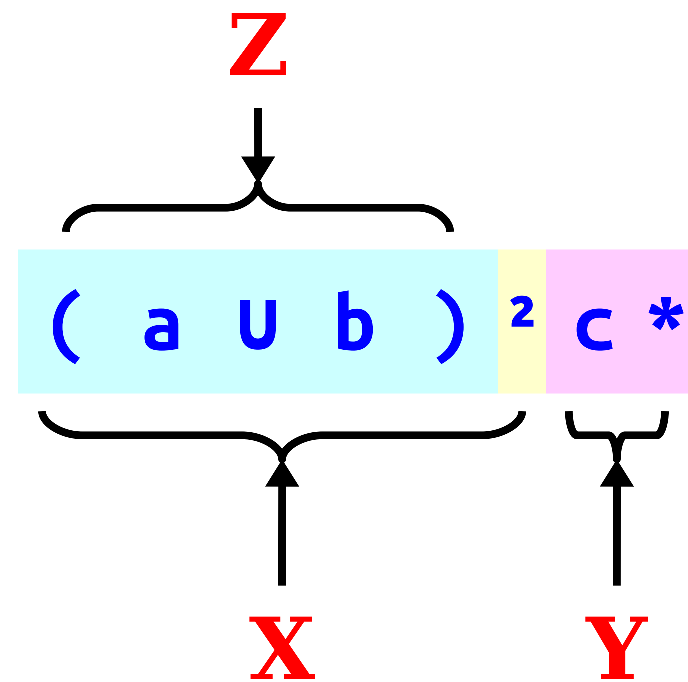
S → **XY**

$$\underbrace{(a \cup b)^2}_{\mathbf{X}}\ \underbrace{c^*}_{\mathbf{Y}}$$

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**

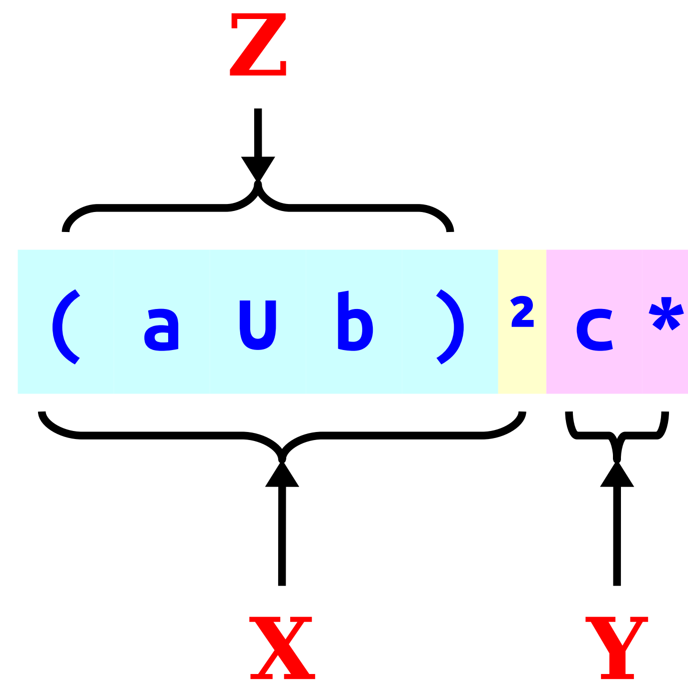


CFGs and Regular Expressions

- ***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**



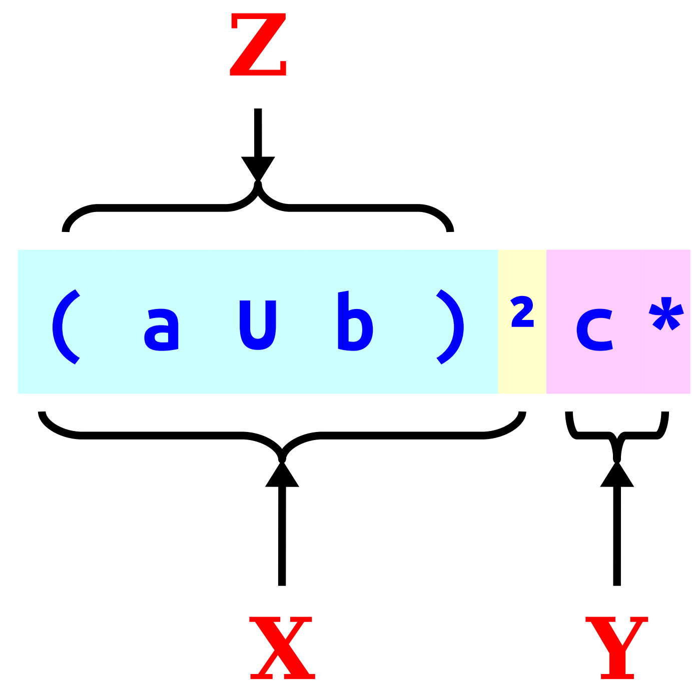
CFGs and Regular Expressions

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

$$\mathbf{S} \rightarrow \mathbf{XY}$$

$$\mathbf{X} \rightarrow \mathbf{ZZ}$$

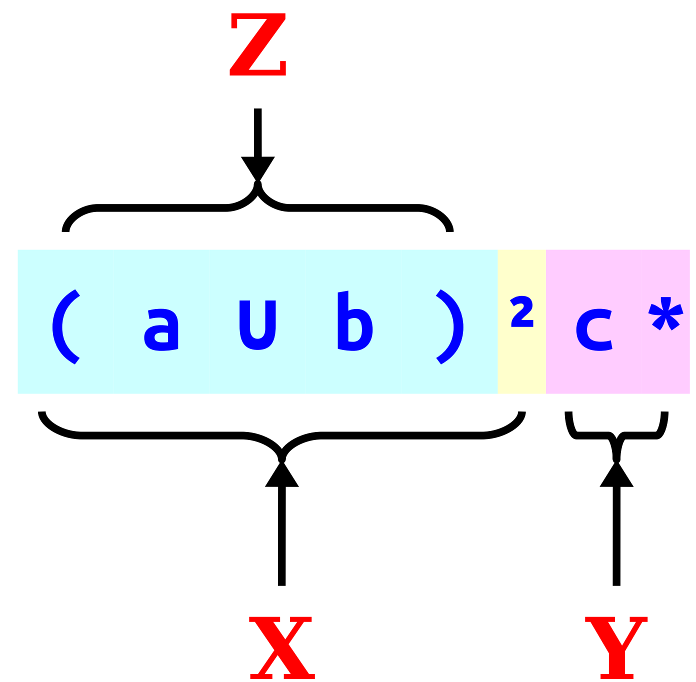
$$\mathbf{Z} \rightarrow \mathtt{a} \mid \mathtt{b}$$



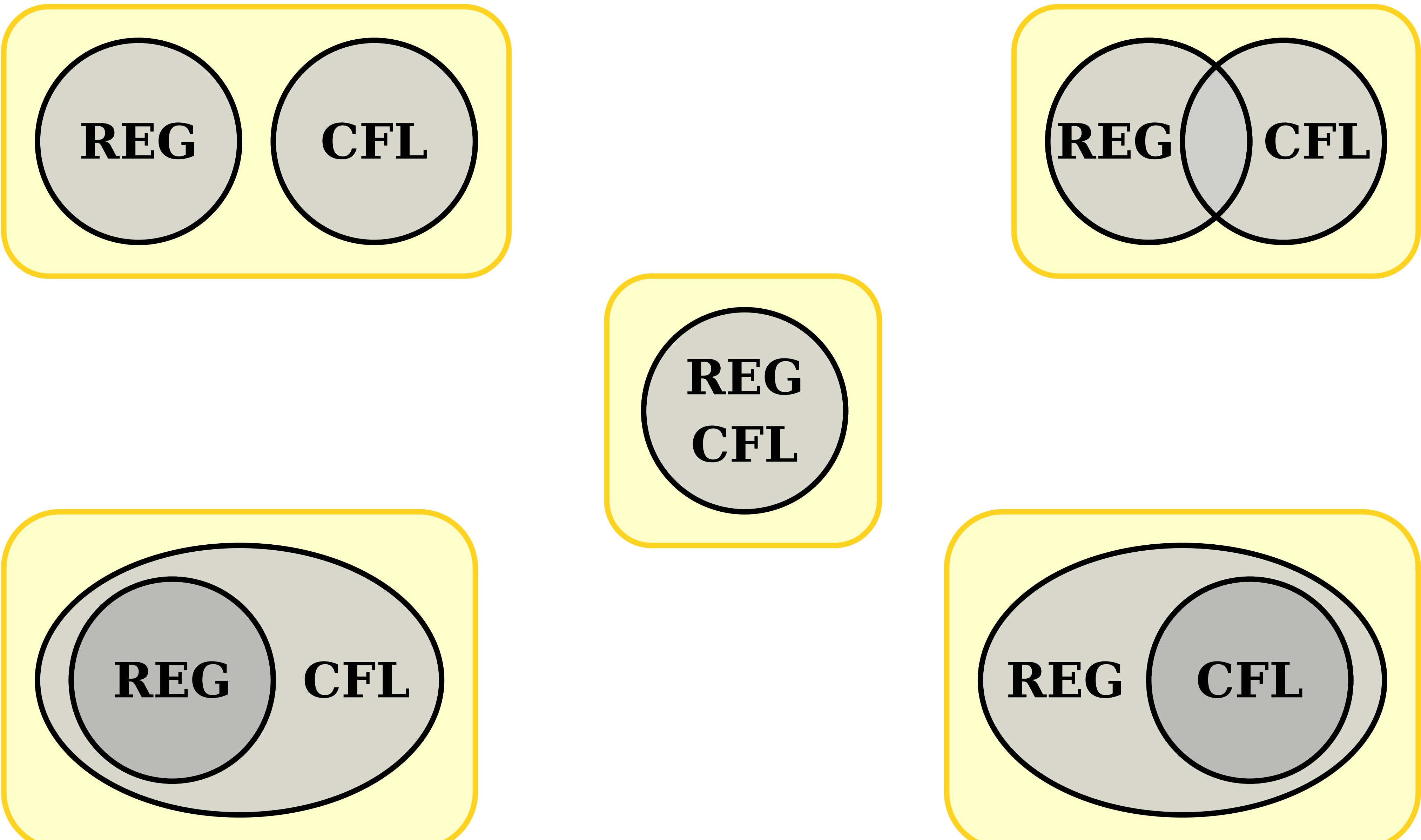
CFGs and Regular Expressions

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

$$S \rightarrow XY$$
$$X \rightarrow ZZ$$
$$Z \rightarrow a \mid b$$
$$Y \rightarrow cY \mid \varepsilon$$



Two ~~Five~~ Possibilities



The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

The Language of a Grammar

- Consider the following CFG G:

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

- What strings can this generate?

$$\boxed{\mathbf{S}}$$

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

a	**S**	**b**

The Language of a Grammar

- Consider the following CFG G:

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

- What strings can this generate?

$$\boxed{\mathtt{a}} \quad \boxed{\mathbf{S}} \quad \boxed{\mathtt{b}}$$

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

a	a	S	b	b

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

a	**a**	**S**	**b**	**b**

The Language of a Grammar

- Consider the following CFG G:

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

- What strings can this generate?

a	a	a	S	b	b	b

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

a	a	a	S	b	b	b

The Language of a Grammar

- Consider the following CFG G:

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

- What strings can this generate?

a	a	a	a	S	b	b	b	b

The Language of a Grammar

- Consider the following CFG G:

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

- What strings can this generate?

a	a	a	a		b	b	b	b

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

a	a	a	a	b	b	b	b

The Language of a Grammar

- Consider the following CFG G:

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

- What strings can this generate?

a	a	a	a	b	b	b	b

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


Regular
Languages
CFLs
All Languages

Why the Extra Power?

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

Why the Extra Power?

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

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

Why the Extra Power?

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

S → **aSb** | **ε**

$\boxed{\mathbf{S}}$

Why the Extra Power?

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

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

a	S	b

Why the Extra Power?

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

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

$\boxed{\mathtt{a}}$ $\boxed{\mathbf{S}}$ $\boxed{\mathtt{b}}$

Why the Extra Power?

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

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

a	a	S	b	b

Why the Extra Power?

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

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

a	a		S		b	b

Why the Extra Power?

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

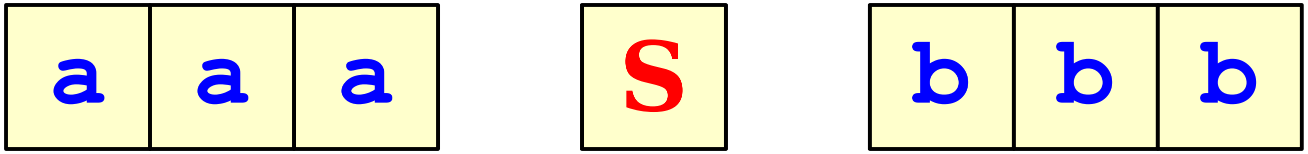
$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

a	a	a	S	b	b	b

Why the Extra Power?

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

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



Why the Extra Power?

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

$$S \rightarrow \mathtt{a}S\mathtt{b} \mid \varepsilon$$

a	**a**	**a**	**a**	**S**	**b**	**b**	**b**	**b**

Why the Extra Power?

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

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

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

Why the Extra Power?

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

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

a	a	a	a	b	b	b	b

Time-Out for Announcements!

Problem Set Seven

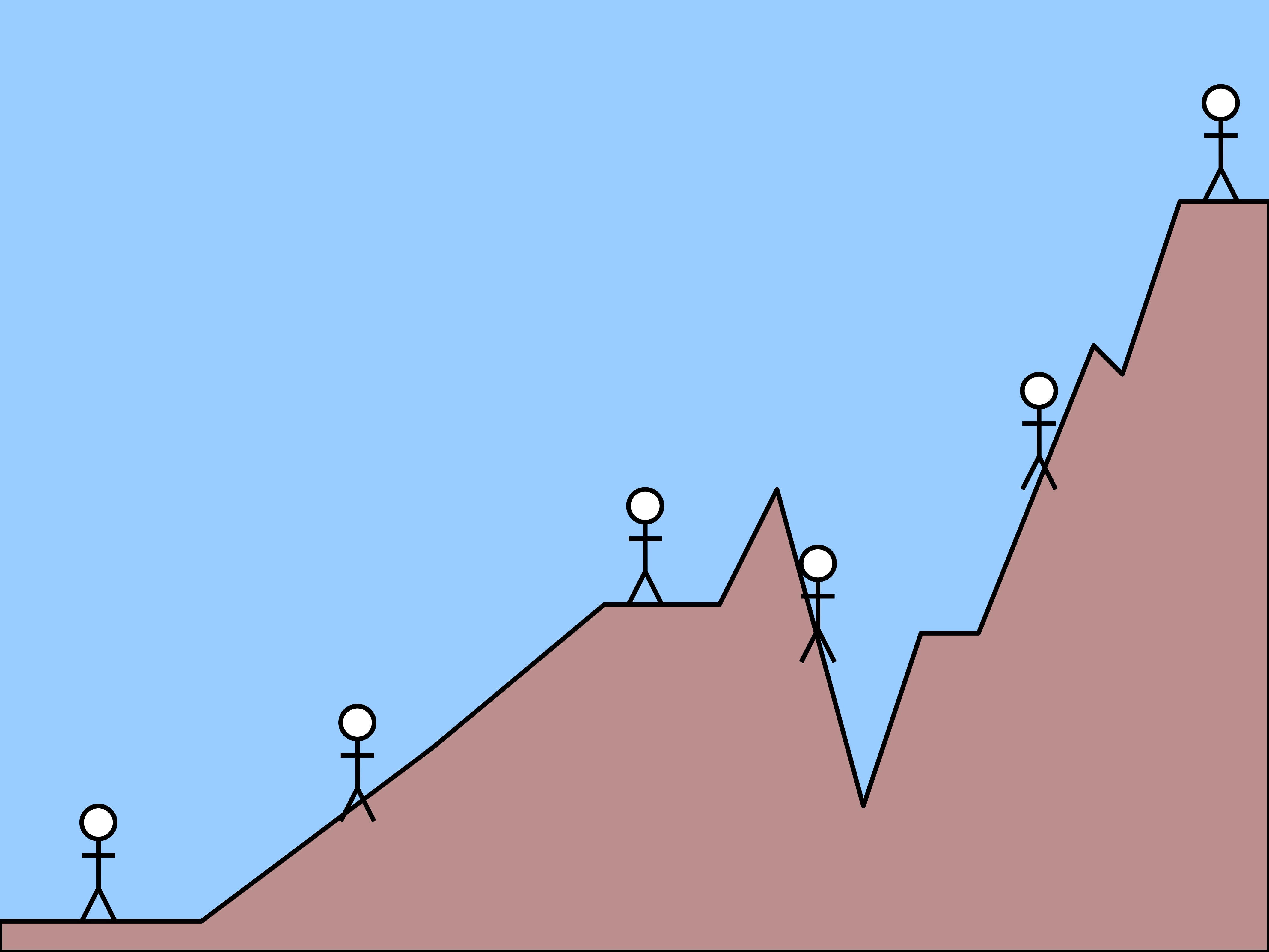
- Problem Set Six was due today at 1:00PM.
  - You can extend the deadline to Saturday at 1:00PM using a late day.
- Problem Set Seven goes out today. It's due next Friday at 1:00PM.
  - It's all about regular expressions, properties of regular languages, and gives a first glimpse at nonregular languages.
  - We've tuned the length given that you have a midterm next Monday.

Second Midterm Logistics

- Our second midterm exam is next ***Monday, November 11th*** from ***7PM – 10PM***. Check the website for your seating assignment; note that it has changed since the first midterm.
- 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.
- Students with OAE accommodations: you should have heard from us with alternate exam locations. If you haven't, contact us ASAP.

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?

Back to CS103!

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.

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

Designing CFGs

- Let Σ = {**{**, **}**} 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 $\Sigma = \{\texttt{\{}, \texttt{\}}\}$ 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 Σ = {**{**, **}**} and let L = {$w \in \Sigma^*$ | 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 Σ = {**{**, **}**} and let L = { w ∈ Σ* | 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

- 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**

Designing CFGs

- Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ 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**

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**

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**

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.

CFG Caveats II

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

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

- What strings in $\{\mathtt{a}, \mathtt{b}\}^*$ 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 $\Sigma = \{\mathtt{a}, \overset{?}{=}\}$ and let $L = \{\mathtt{a}^n \overset{?}{=} \mathtt{a}^n \mid n \in \mathbb{N}\}$.
- Is the following a CFG for L?

$$\mathbf{S} \to \mathbf{X} \overset{?}{=} \mathbf{X}$$

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

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

Finding a Build Order

- Let $\Sigma = \{\texttt{a}, \overset{?}{=}\}$ and let $L = \{\texttt{a}^n \overset{?}{=} \texttt{a}^n \mid n \in \mathbb{N}\}$.
- To build a CFG for L, we need to be more clever with how we construct the string.
  - 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:

$$S \rightarrow \overset{?}{=} \mid \texttt{a}S\texttt{a}$$

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

Function Prototypes

- Let Σ = {`void`, `int`, `double`, `name`, `(`, `)`, `,`, `;`}.
- Let's write a CFG for C-style function prototypes!
- Examples:
  - `void name(int name, double name);`
  - `int name();`
  - `int name(double name);`
  - `int name(int, int name, int);`
  - `void name(void);`

Function Prototypes

- Here's one possible grammar:
  - **S** → **Ret** `name` `(`**Args**`);`
  - **Ret** → **Type** | `void`
  - **Type** → `int` | `double`
  - **Args** → `ε` | `void` | **ArgList**
  - **ArgList** → **OneArg** | **ArgList**, **OneArg**
  - **OneArg** → **Type** | **Type** `name`
- Fun question to think about: what changes would you need to make to support pointer types?

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

- ***No class Monday.***
- ***Then, when we get back…***
  - ***Turing Machines***
    - What does a computer with unbounded memory look like?
    - How would you program it?