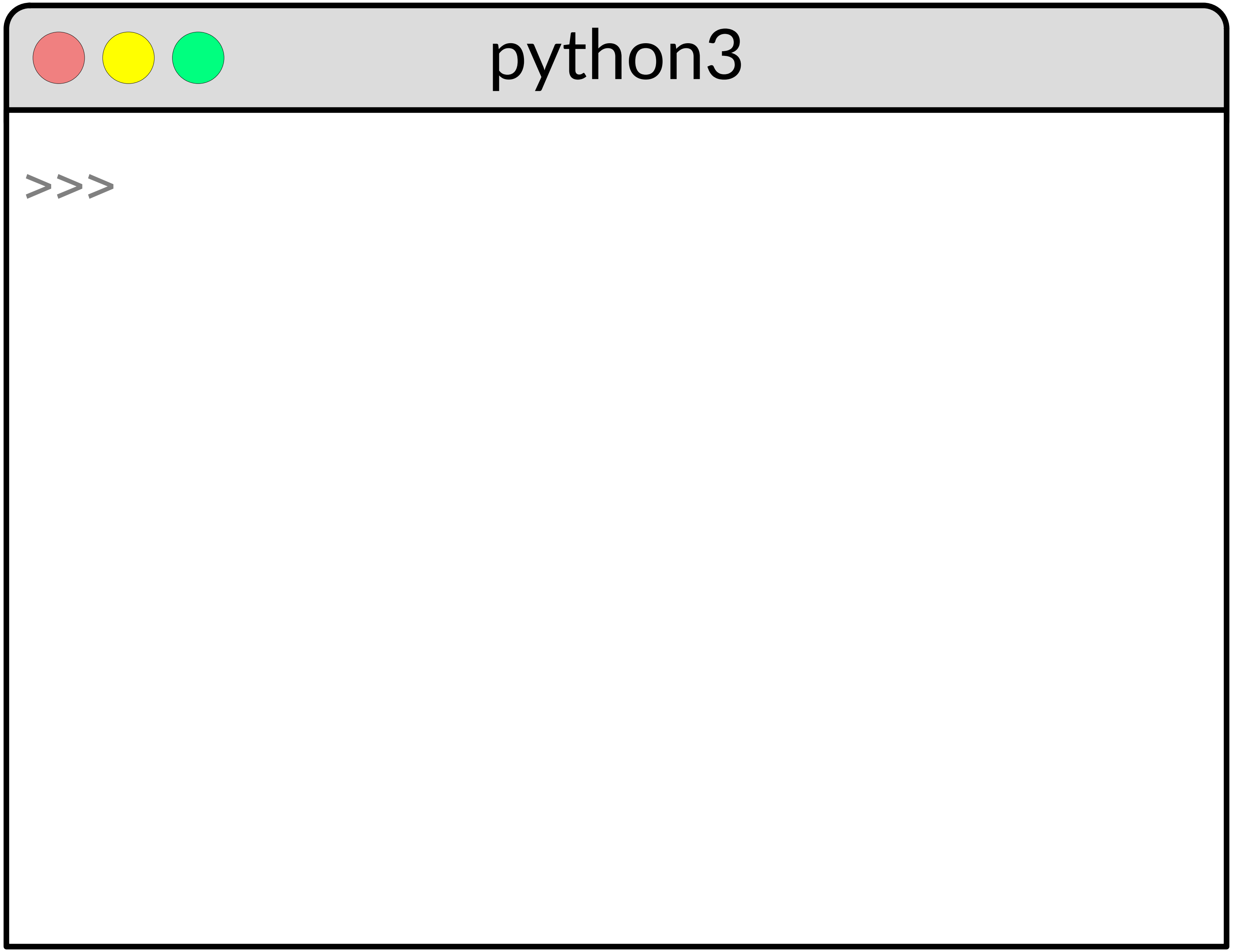
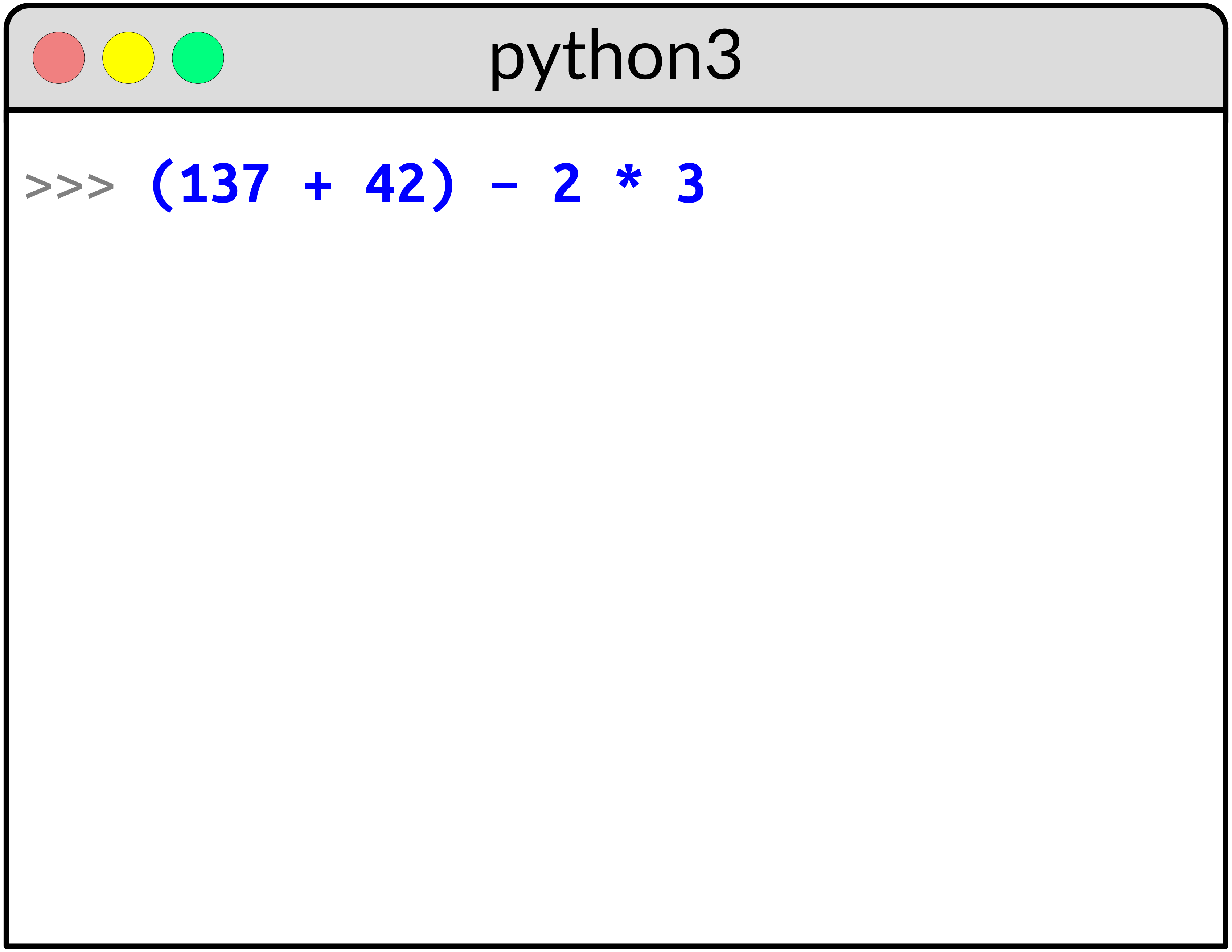
Context-Free Grammars

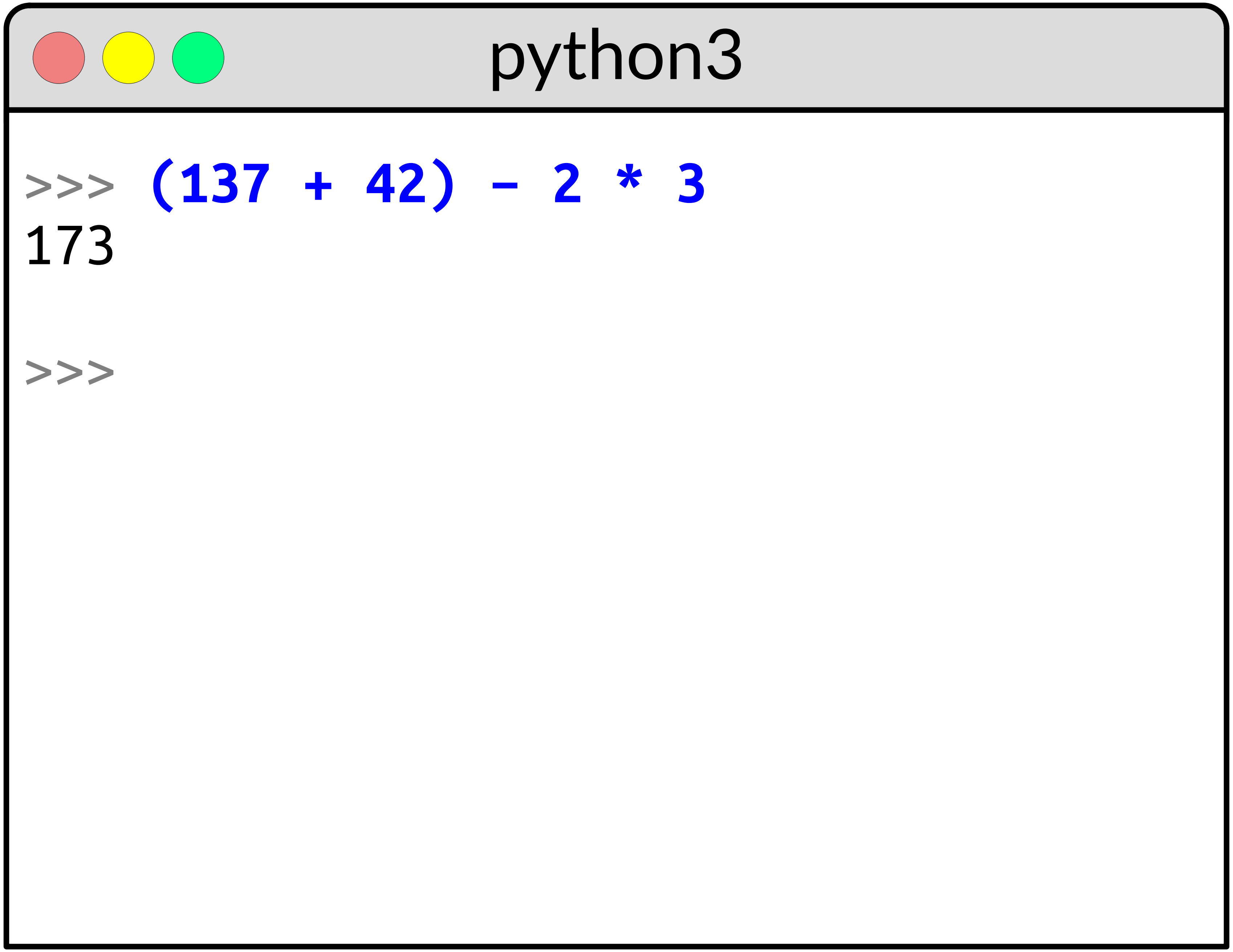
A Motivating Question


python3

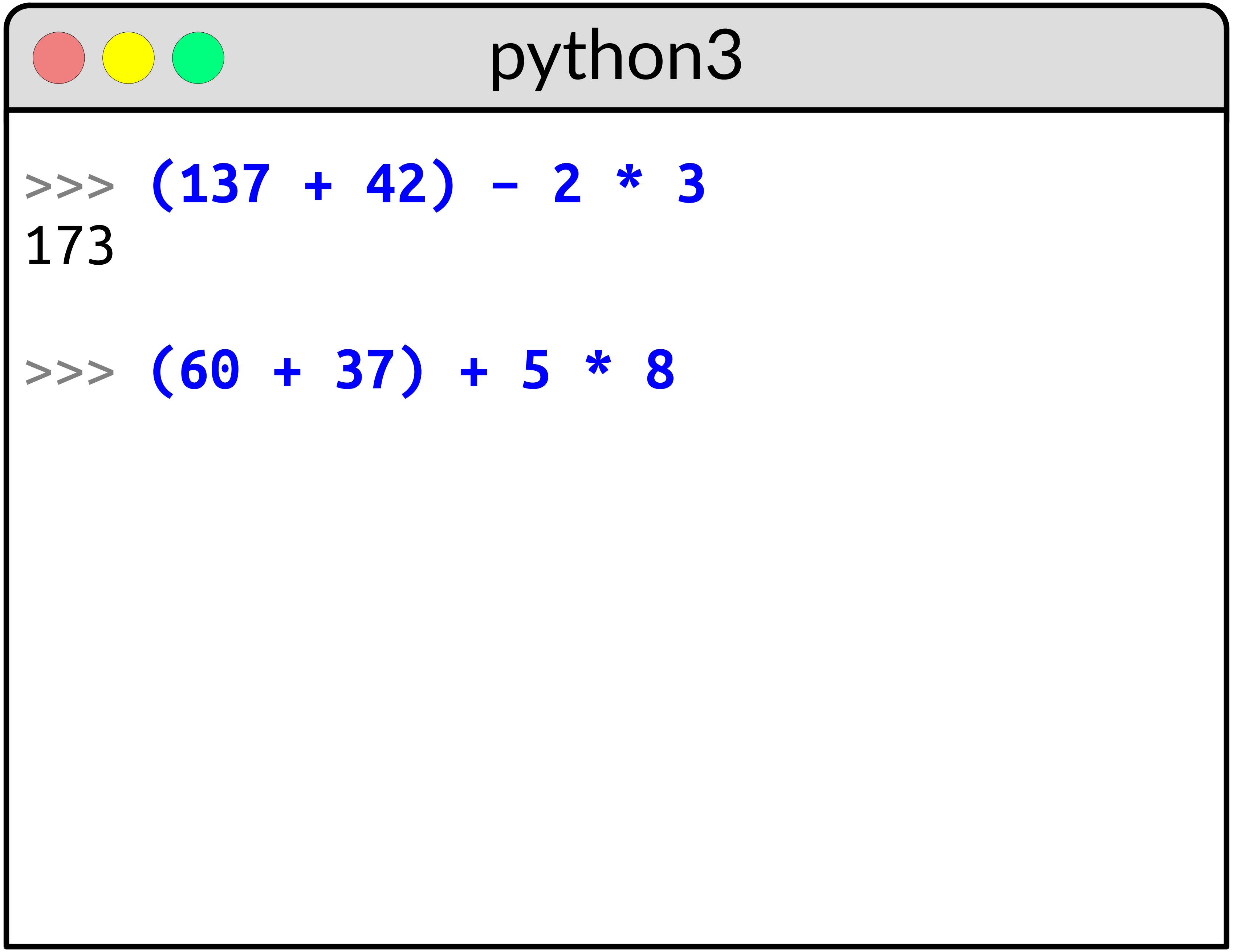
```
>>>
```



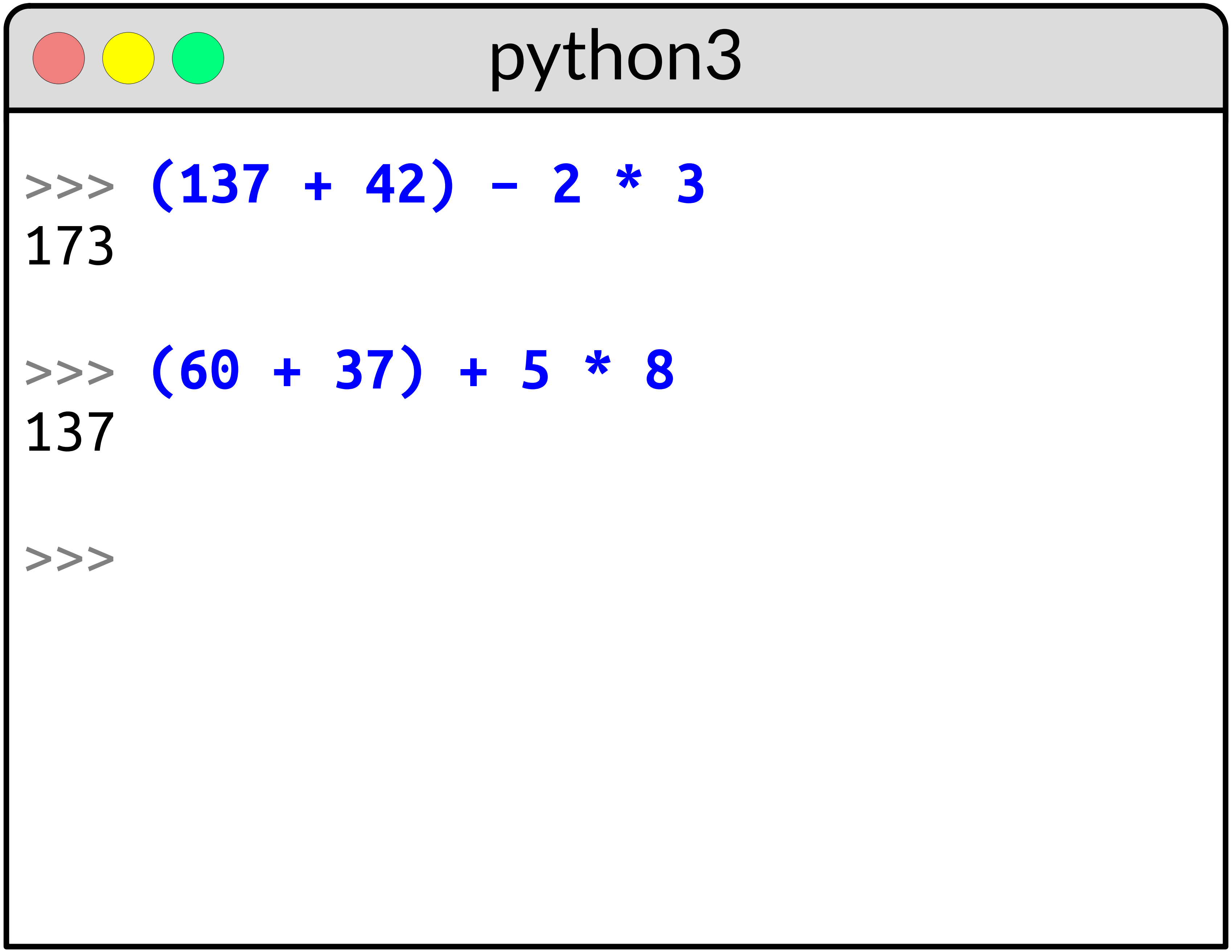
```
>>> (137 + 42) - 2 * 3
```



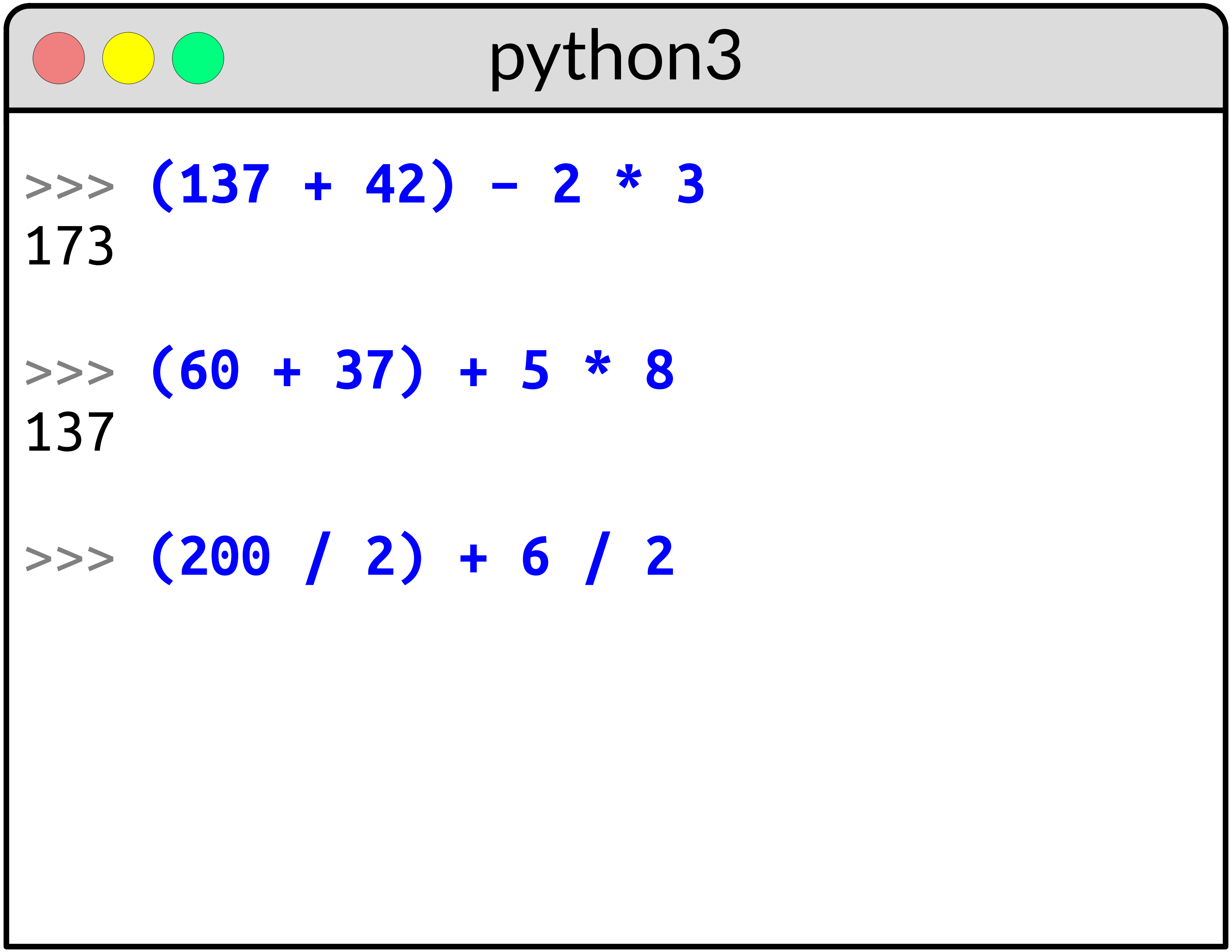
```
>>> (137 + 42) - 2 * 3
173


>>>
```



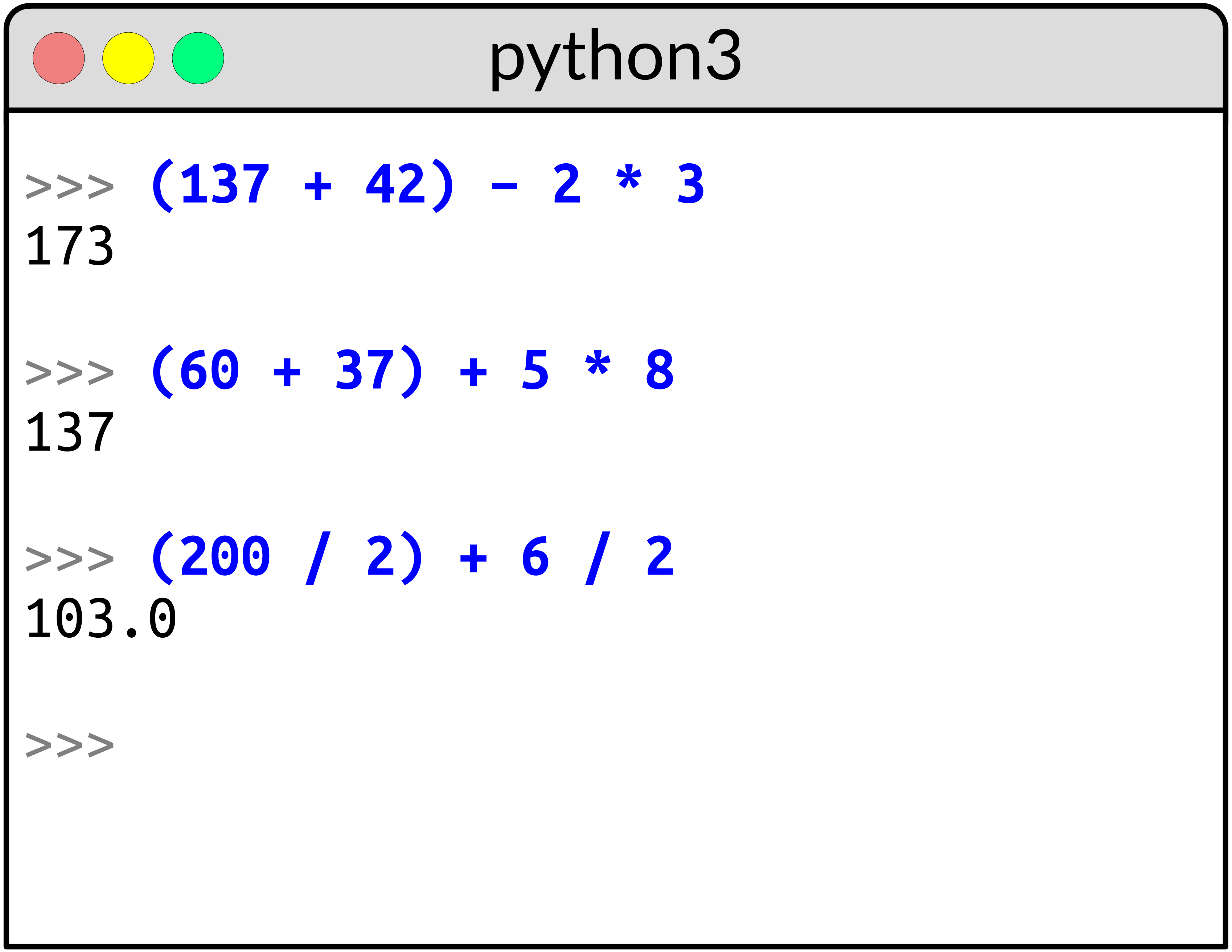
```
>>> (137 + 42) - 2 * 3
173

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



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

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

>>>
```



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

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

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



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

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

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

>>>
```

Mad Libs for Arithmetic

$$(\; \underset{\textbf{Int}}{\underline{\qquad}} \; \underset{\textbf{Op}}{\underline{\qquad}} \; \underset{\textbf{Int}}{\underline{\qquad}} \;) \; \underset{\textbf{Op}}{\underline{\qquad}} \; \underset{\textbf{Int}}{\underline{\qquad}} \; \underset{\textbf{Op}}{\underline{\qquad}} \; \underset{\textbf{Int}}{\underline{\qquad}}$$

Slide credit: Amy Liu

Mad Libs for Arithmetic

$$(\underset{\text{Int}}{\underline{\;26\;}} \; \underset{\text{Op}}{\underline{\;+\;}} \; \underset{\text{Int}}{\underline{\;42\;}} \;) \; \underset{\text{Op}}{\underline{\;*\;}} \; \underset{\text{Int}}{\underline{\;2\;}} \; \underset{\text{Op}}{\underline{\;+\;}} \; \underset{\text{Int}}{\underline{\;1\;}}$$

Slide credit: Amy Liu

Mad Libs for Arithmetic

(____ ____ ____) ____ ____ ____ ____

Int **Op** **Int** **Op** **Int** **Op** **Int**

Slide credit: Amy Liu

Mad Libs for Arithmetic

$$(\; \underset{\textbf{Int}}{\underline{\;7\;}} \; \underset{\textbf{Op}}{\underline{\;*\;}} \; \underset{\textbf{Int}}{\underline{\;5\;}} \;) \; \underset{\textbf{Op}}{\underline{\;/\;}} \; \underset{\textbf{Int}}{\underline{\;5\;}} \; \underset{\textbf{Op}}{\underline{\;-\;}} \; \underset{\textbf{Int}}{\underline{\;49\;}}$$

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

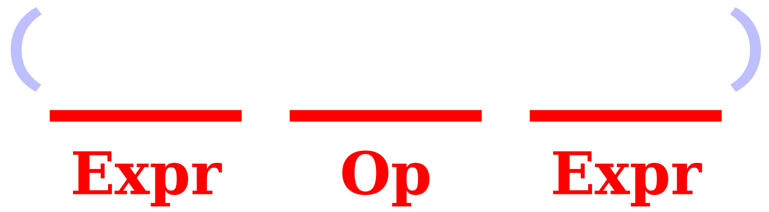
(______)

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



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

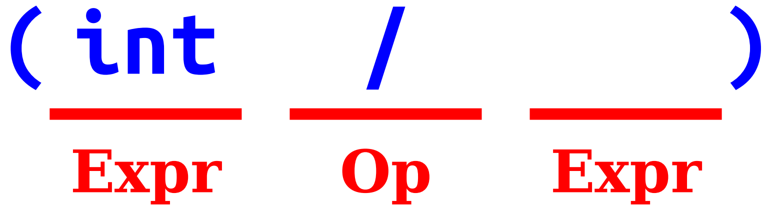
(int ____ ____ ____)

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.

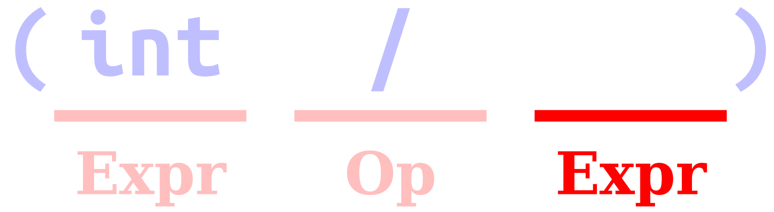
Recursive Mad Libs



What can an arithmetic expression be?

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

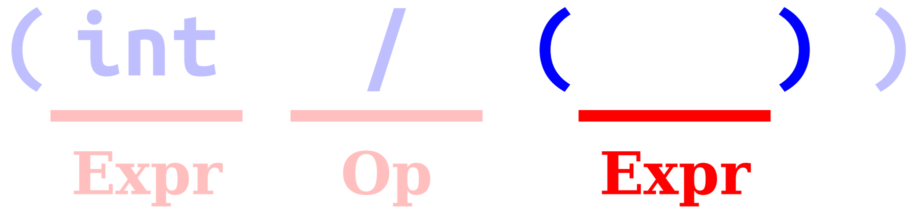
Recursive Mad Libs



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



What can an arithmetic expression be?

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

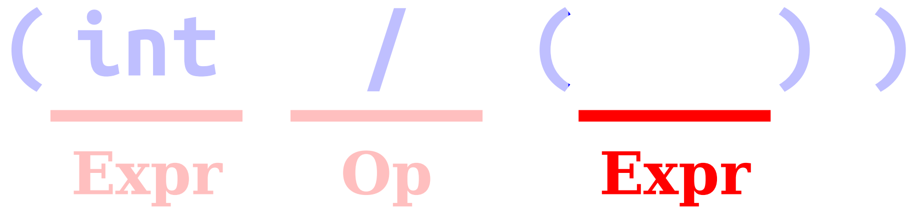
Recursive Mad Libs

$$(\underbrace{\texttt{int}}_{\textbf{Expr}}\ \underbrace{\texttt{/}}_{\textbf{Op}}\ (\underbrace{\qquad}_{\textbf{Expr}})\,)$$

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



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

$$(\ \underset{\textbf{Expr}}{\underline{\texttt{int}}}\ \ \underset{\textbf{Op}}{\underline{\ /\ }}\ \ (\ \underset{\textbf{Expr}}{\underline{\qquad}}\ \ \underset{\textbf{Op}}{\underline{\qquad}}\ \ \underset{\textbf{Expr}}{\underline{\qquad}}\ )\ )$$

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

$$(\,\underset{\textbf{Expr}}{\underline{\texttt{int}}}\;\;\underset{\textbf{Op}}{\underline{\;/\;}}\;\;(\,\underset{\textbf{Expr}}{\underline{\texttt{int}}}\;\;\underset{\textbf{Op}}{\underline{\;+\;}}\;\;\underset{\textbf{Expr}}{\underline{\texttt{int}}}\,)\;)$$

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.

$$
\begin{array}{l}
\mathbf{Expr} \rightarrow \texttt{int} \\
\mathbf{Expr} \rightarrow \mathbf{Expr}\ \mathbf{Op}\ \mathbf{Expr} \\
\mathbf{Expr} \rightarrow \texttt{(}\mathbf{Expr}\texttt{)} \\
\mathbf{Op} \rightarrow \texttt{+} \\
\mathbf{Op} \rightarrow \texttt{-} \\
\mathbf{Op} \rightarrow \times \\
\mathbf{Op} \rightarrow \texttt{/}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathbf{Expr} \\
\Rightarrow & \mathbf{Expr}\ \mathbf{Op}\ \mathbf{Expr} \\
\Rightarrow & \mathbf{Expr}\ \mathbf{Op}\ \texttt{int} \\
\Rightarrow & \texttt{int}\ \mathbf{Op}\ \texttt{int} \\
\Rightarrow & \texttt{int / int}
\end{array}
$$

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 → **/**

$$
\begin{aligned}
& \textbf{Expr} \\
\Rightarrow\ & \textbf{Expr Op Expr} \\
\Rightarrow\ & \textbf{Expr Op int} \\
\Rightarrow\ & \textbf{int Op int} \\
\Rightarrow\ & \textbf{int / int}
\end{aligned}
$$

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 → `/`

$$
\begin{aligned}
& \textbf{Expr} \\
\Rightarrow\ & \textbf{Expr Op Expr} \\
\Rightarrow\ & \textbf{Expr Op } \texttt{int} \\
\Rightarrow\ & \texttt{int}\textbf{ Op }\texttt{int} \\
\Rightarrow\ & \texttt{int / int}
\end{aligned}
$$

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. $\boldsymbol{\alpha}$, $\boldsymbol{\gamma}$, $\boldsymbol{\omega}$
- 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 → **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 ⇒* **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

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

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

S → **Sa** | **dT**
T → **bTb** | **c**

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

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?

Context-Free Languages

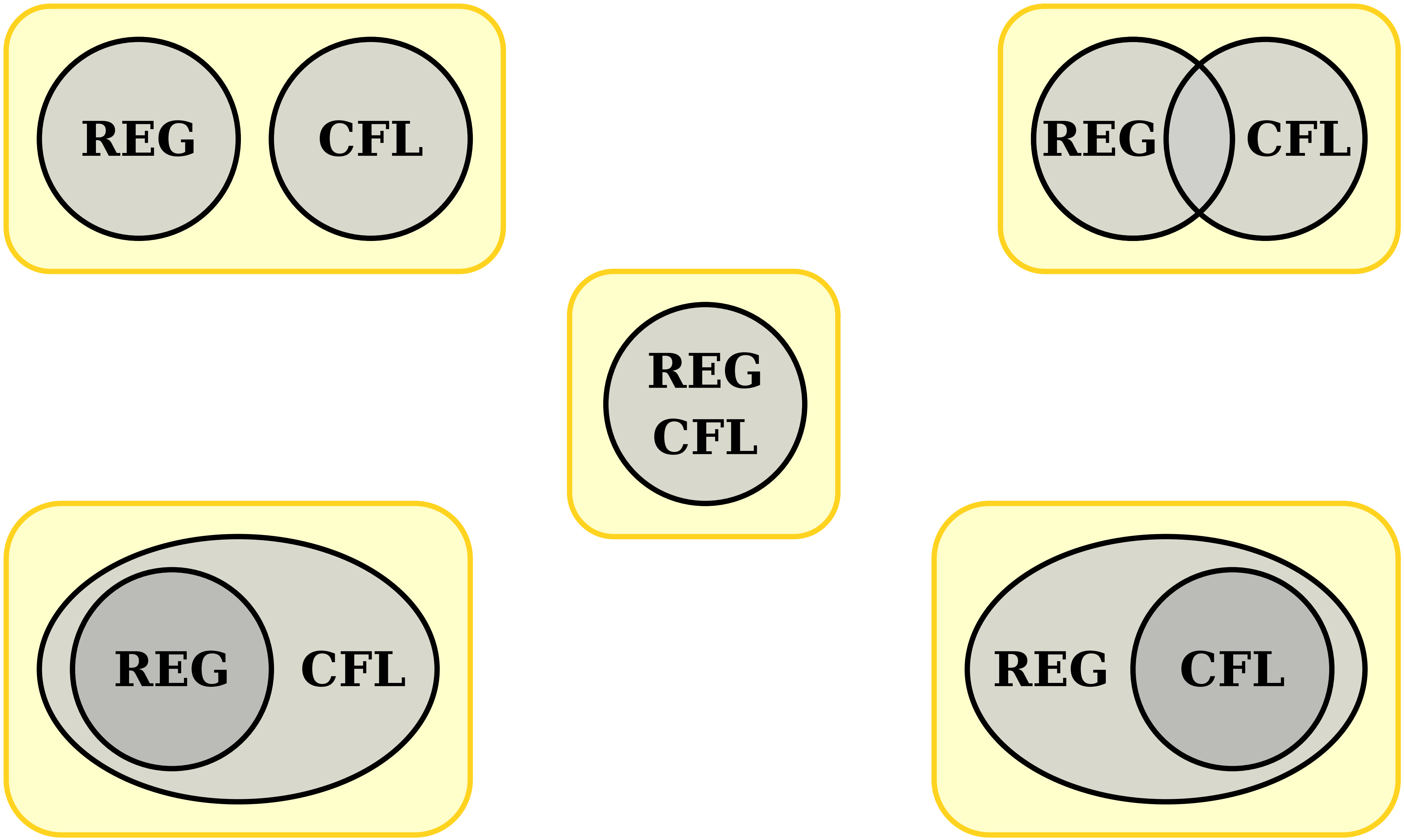
A language L is called a ***context-free language*** (or CFL) if there is a CFG G such that $L = \mathscr{L}(G)$.

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:

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

- 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

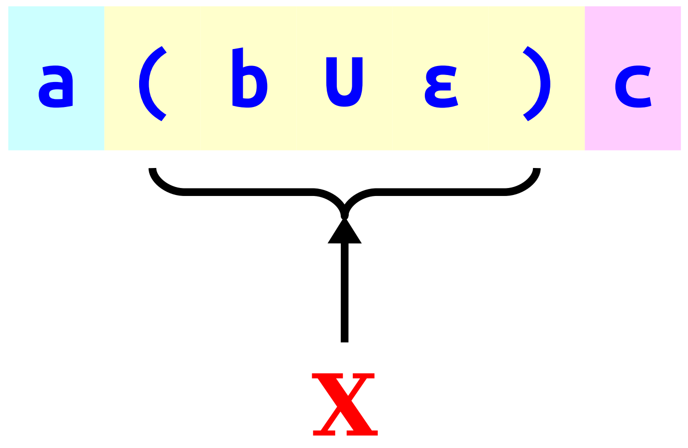
CFGs and Regular Expressions

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

a (b ∪ ε) c

CFGs and Regular Expressions

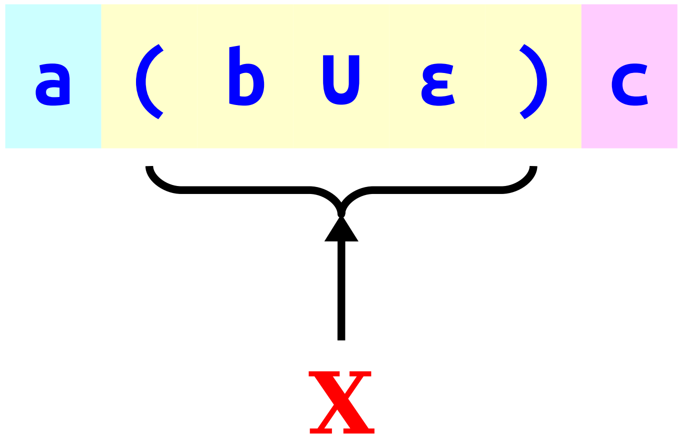
- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.



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.

$$\boxed{\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}\mathtt{c}}$$


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

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.

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

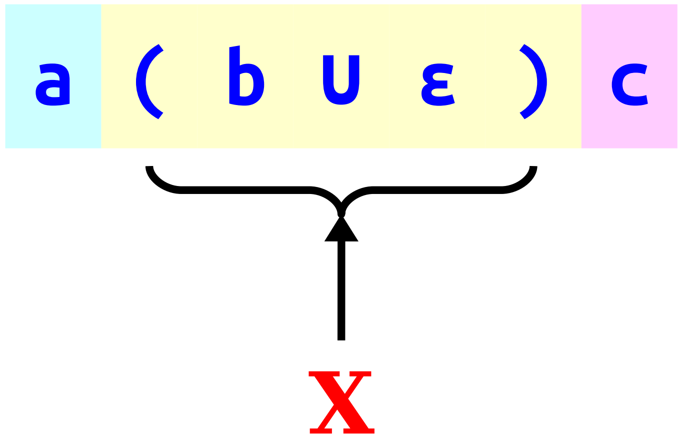
$$a\underbrace{(\, b \cup \varepsilon \,)}_{X}c$$

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.

$$\boxed{\begin{aligned} \mathbf{S} &\to \mathtt{a}\mathbf{X}\mathtt{c} \\ \mathbf{X} &\to \mathtt{b} \mid \varepsilon \end{aligned}}$$

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.

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

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

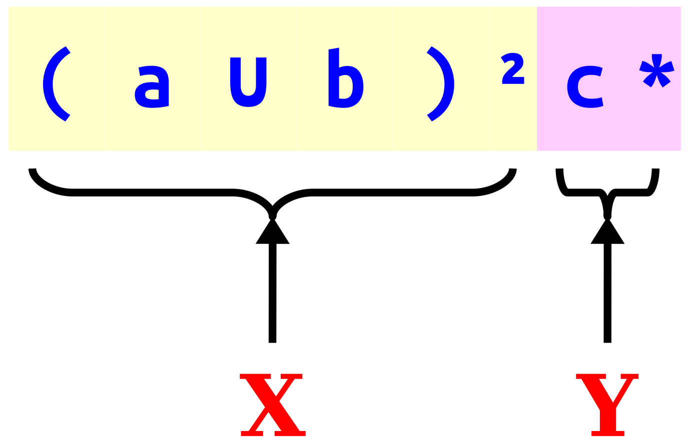
- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

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

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**

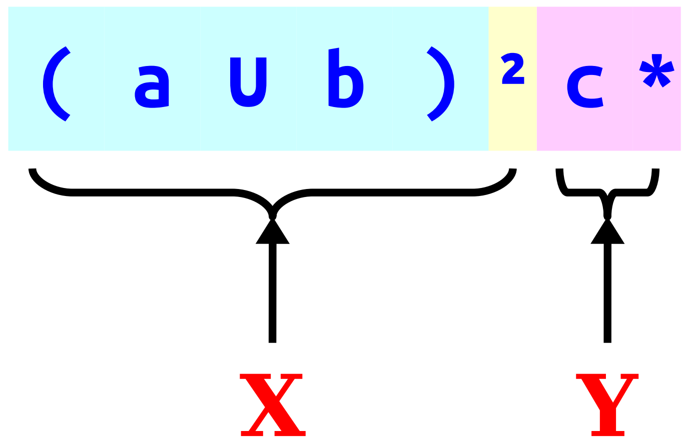


$$\underbrace{(\mathtt{a} \cup \mathtt{b})^2}_{\mathbf{X}}\ \underbrace{\mathtt{c}^*}_{\mathbf{Y}}$$

CFGs and Regular Expressions

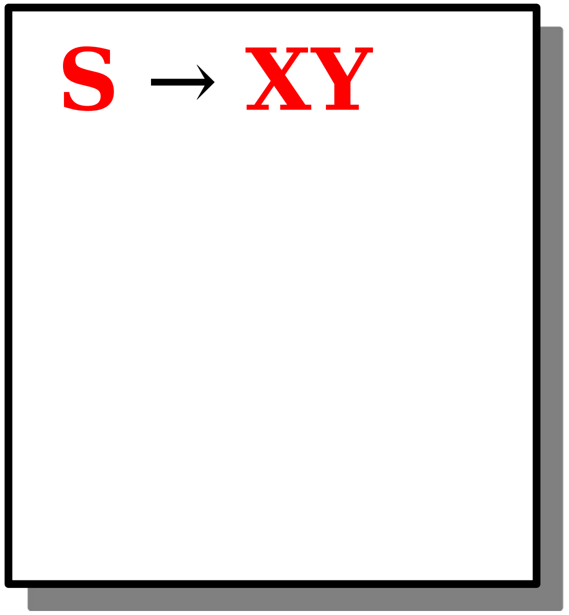
- ***Theorem:*** Every regular language is context-free.
- ***Proof idea:*** Show how to convert an arbitrary regular expression into a context-free grammar.

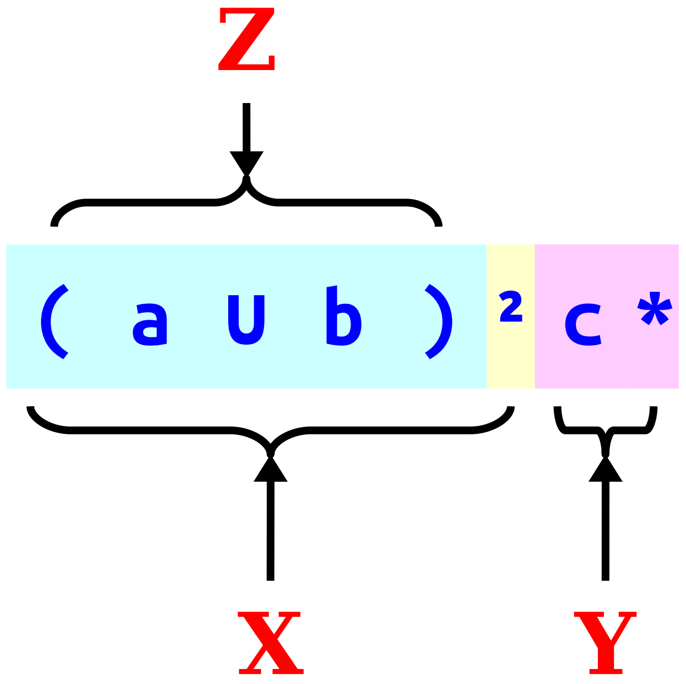
$$\mathbf{S} \rightarrow \mathbf{XY}$$



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.



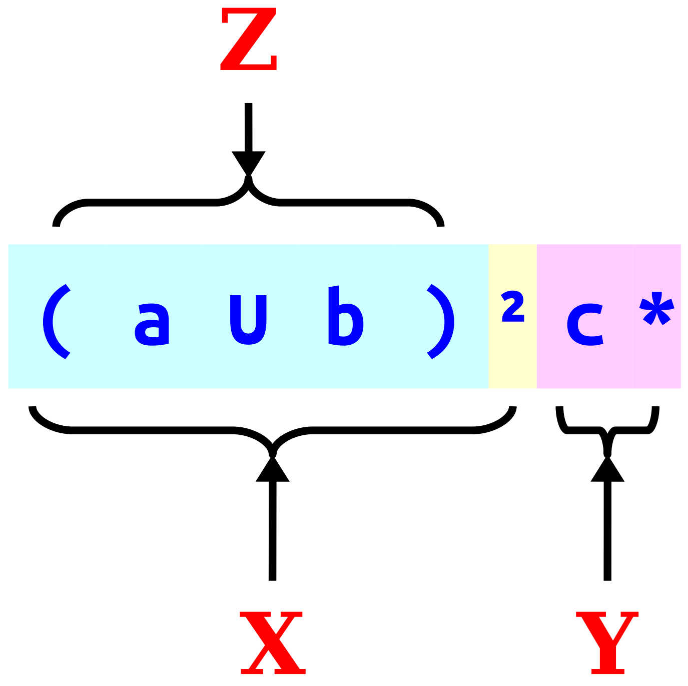


CFGs and Regular Expressions

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

S → **XY**

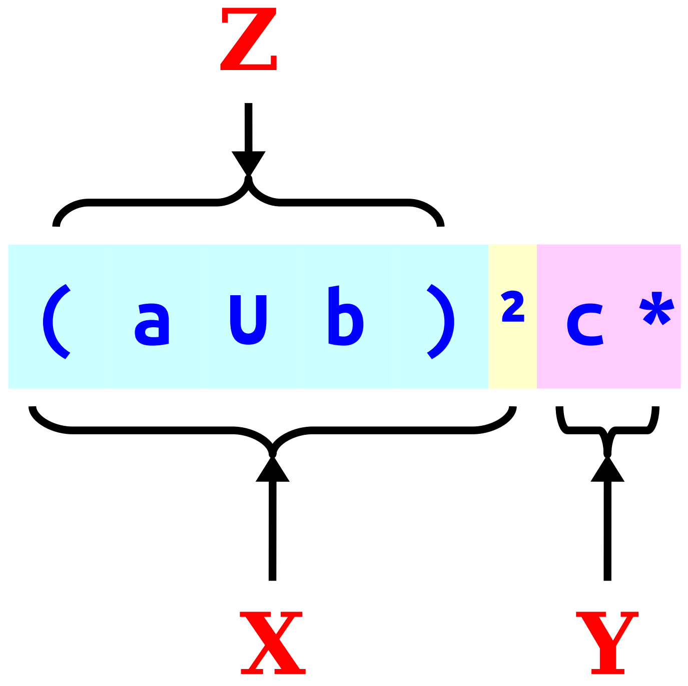
X → **ZZ**



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.

$$
\begin{aligned}
\mathbf{S} &\rightarrow \mathbf{XY} \\
\mathbf{X} &\rightarrow \mathbf{ZZ} \\
\mathbf{Z} &\rightarrow \texttt{a} \mid \texttt{b}
\end{aligned}
$$



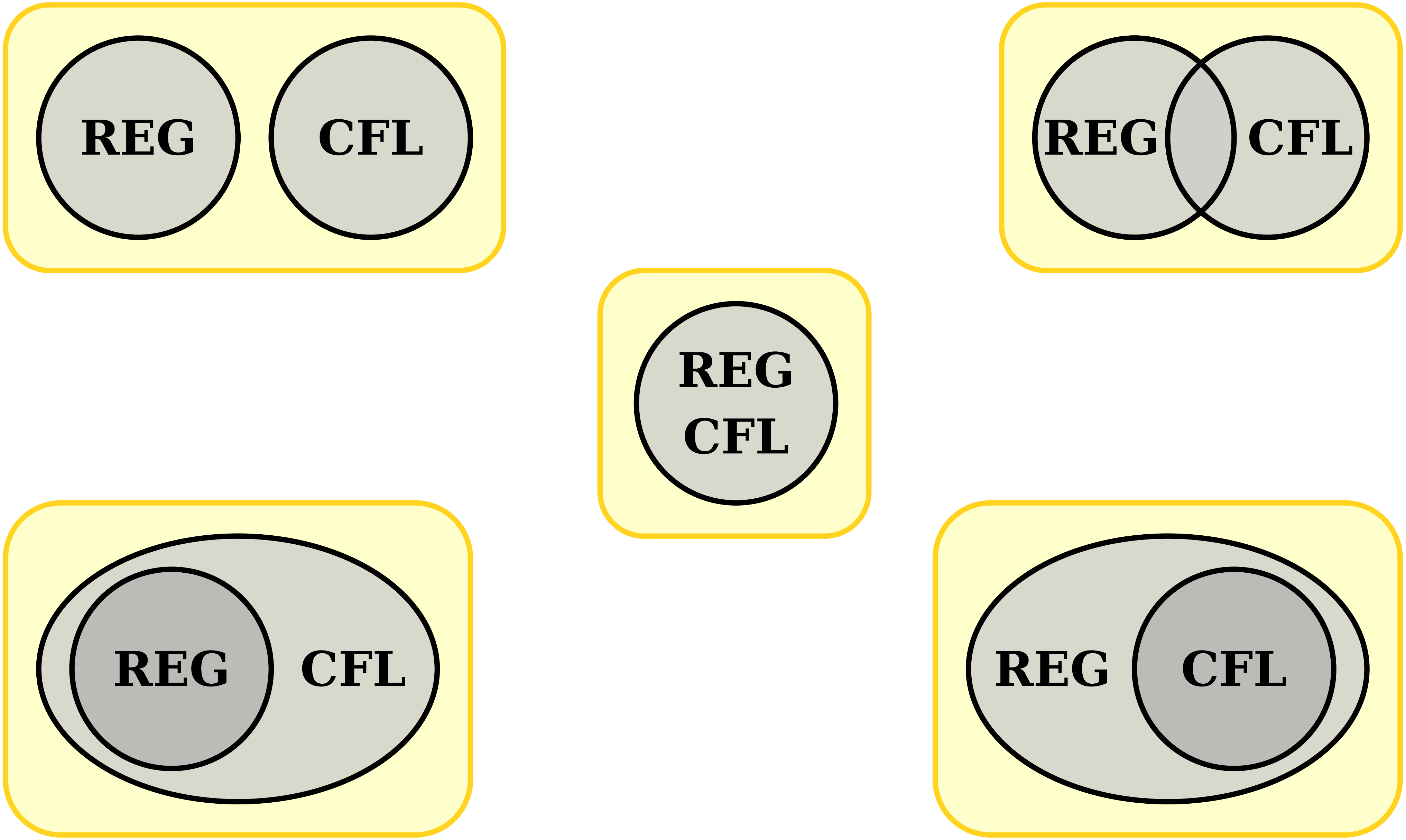
CFGs and Regular Expressions

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

$$S \to XY$$
$$X \to ZZ$$
$$Z \to \mathtt{a} \mid \mathtt{b}$$
$$Y \to \mathtt{c}Y \mid \varepsilon$$

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

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?

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

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

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

a	**S**	**b**

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

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

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

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

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

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

The Language of a Grammar

- Consider the following CFG *G*:

$$S \to aSb \mid \varepsilon$$

- What strings can this generate?

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

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

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

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

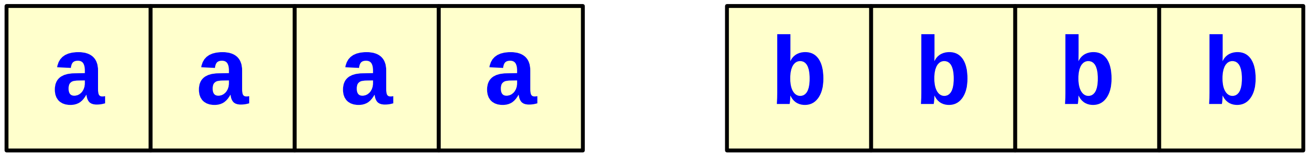
a	**a**	**a**	**a**	**S**	**b**	**b**	**b**	**b**

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?



The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

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

The Language of a Grammar

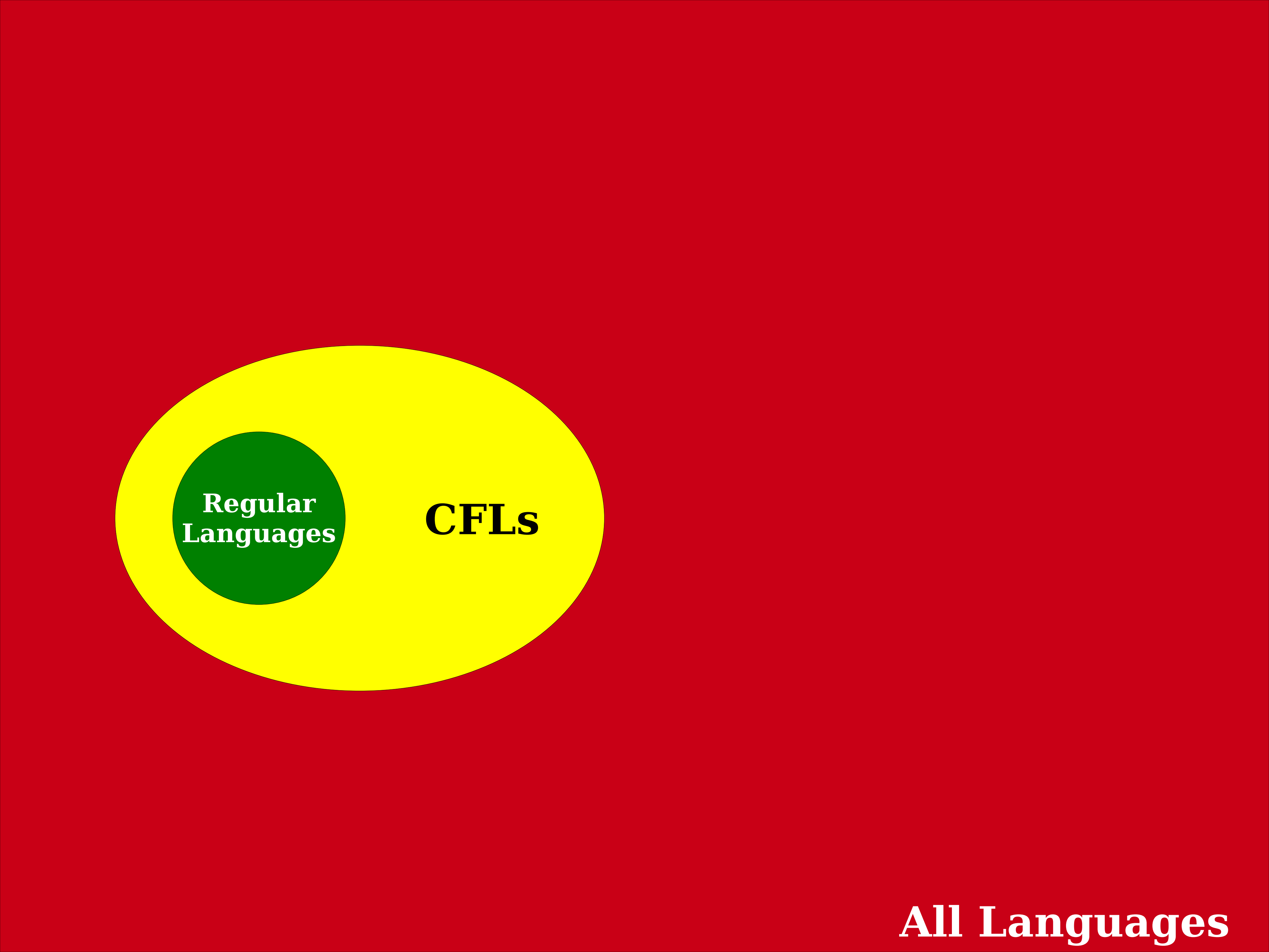
- Consider the following CFG G:

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

- 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

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."

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

a	**S**	**b**

Why the Extra Power?

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

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

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

Why the Extra Power?

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

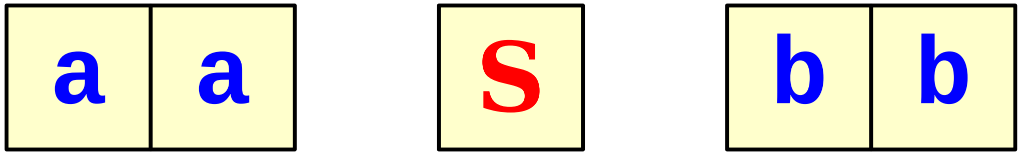
$$S \to aSb \mid \varepsilon$$

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."

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



Why the Extra Power?

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

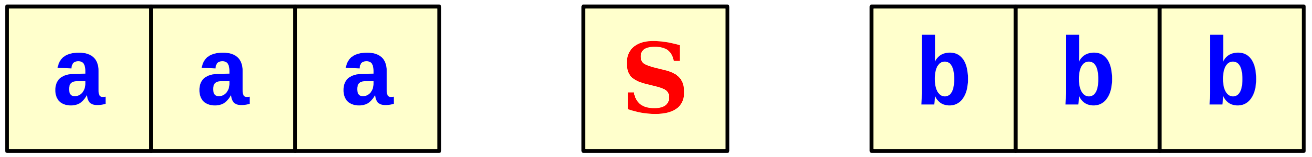
$$S \to aSb \mid \varepsilon$$

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."

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



Why the Extra Power?

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

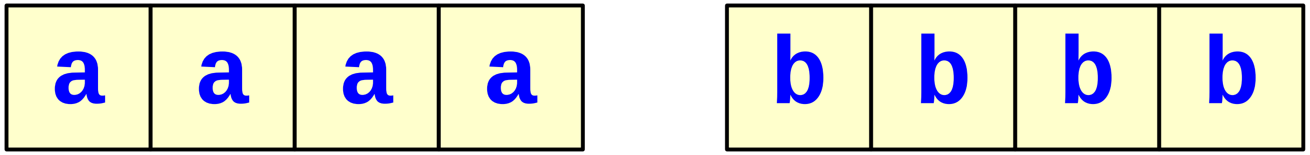
S → aSb | ε

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."

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



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**	**b**	**b**	**b**	**b**

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 Monday when you’ve finished the midterm.

Midterm Exam Logistics

- Our next midterm runs today (Friday, November 5th) through Sunday, November 7th at 2:30PM, Pacific time.
  - That’s 49 hours rather than the normal 48. Huzzah!
- 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.
- ***Best of luck on the exam – you can do this!***

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!

Your Questions

“have you ever felt like you were "running out of time" or "behind" your peers, and how did you deal with those feelings?”

Oh yeah, definitely. That's a human universal.

My recommendation is to take the statement "I am supposed to be further ahead by this point" and notice it's in the passive voice. Who, exactly, is doing the supposing? And why do they "suppose" it? And why does someone else supposing something make you feel bad?

It's one thing to think "wow, I know amazing people who have accomplished a lot, or done things I haven't done, etc." It's another to then say "and that diminishes me because someone specific expects me to have done something similar already." Keep those separate. Be proud of what you have done. Be inspired by those around you to do more. But don't feel guilty about it.

As for "running out of time" – most doors in life never slam shut and instead just get harder to open as time progresses.

“What is one moment in your life that you wish there was a camera there filming you?”

This one! And I got my wish.

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.

Designing CFGs

- Let Σ = {a, b} and let L = {w ∈ Σ* | 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 aωa and bωb are palindromes.
  - No other strings are palindromes.

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

Designing CFGs

- Let Σ = {**{**, **}**} and let L = {w ∈ Σ* | w is a string of balanced braces }
- Some sample strings in L:

{{{}}}

{{}}{}

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

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

ε

{}{}

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.

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

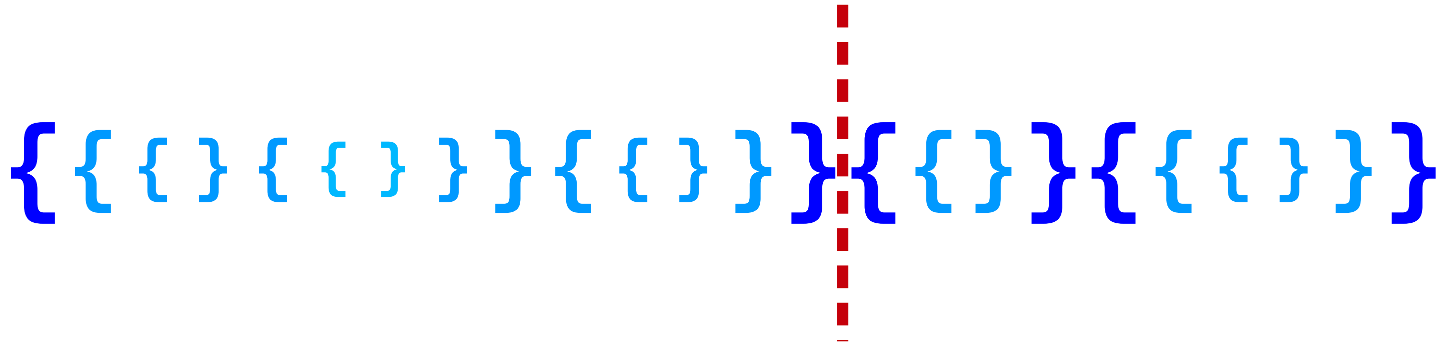
Designing CFGs

- Let 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.

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

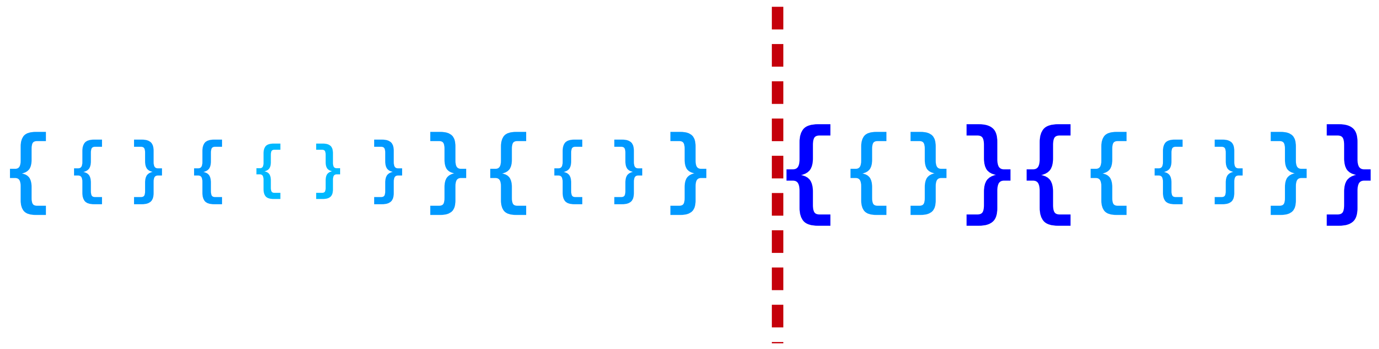
Designing CFGs

- Let 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.



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. Removing the first brace and the matching brace forms two new strings of balanced braces.

S → {S}S | ε

Designing CFGs

- Let Σ = {**a**, **b**} and let L = {w ∈ Σ* | w has the same number of **a**'s and **b**'s }

How many of the following CFGs have language L?

S → **aSb** | **bSa** | **ε**

S → **abS** | **baS** | **ε**

S → **abSba** | **baSab** | **ε**

S → **SbaS** | **SabS** | **ε**

Designing CFGs

- Let Σ = {**a**, **b**} and let L = {w ∈ Σ* | w has the same number of **a**'s and **b**'s }

How many of the following 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 $\}$

How many of the following 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 $\}$

How many of the following CFGs have language L?

S → **aSb** | **bSa** | **ε**

S → **abS** | **baS** | **ε**

S → **abSba** | **baSab** | **ε**

S → **SbaS** | **SabS** | **ε**

Designing CFGs

- Let Σ = {**a**, **b**} and let L = {w ∈ Σ* | w has the same number of **a**'s and **b**'s }

How many of the following 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 { $\mathtt{a}^n\mathtt{b}^n$ | $n \in \mathbb{N}$ }?

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

- 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 $\Sigma = \{\mathtt{a}, \overset{?}{=}\}$ 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 $\Sigma = \{\mathtt{a}, \stackrel{?}{=}\}$ and let $L = \{\mathtt{a}^n \stackrel{?}{=} \mathtt{a}^n \mid n \in \mathbb{N}\}$.
- 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:

$$\mathbf{S} \to \stackrel{?}{=} \mid \mathtt{a}\mathbf{S}\mathtt{a}$$

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

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 on Monday***
  - You earned it!
- ***Turing Machines***
  - What does a computer with unbounded memory look like?
  - How would you program it?