

Context-Free Grammars

Recap from Last Time

- If L is a language and S is a **distinguishing set** for L that contains **infinitely** many strings, then L is not regular
- Distinguishing set: A set of strings, each of which represent information that requires a distinct state to represent it
 - e.g., “we've seen 2 a's, we've seen 3 a's, ...”
- Infinitely many strings in a distinguishing set implies there is no finite number of states that can represent all the required information.

New Stuff

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

Mad Libs for Arithmetic

$$\left(\underbrace{26}_{\text{Int}} \underbrace{+}_{\text{Op}} \underbrace{42}_{\text{Int}} \right) \underbrace{*}_{\text{Op}} \underbrace{2}_{\text{Int}} \underbrace{+}_{\text{Op}} \underbrace{1}_{\text{Int}}$$

Mad Libs for Arithmetic

()

<u>Int</u>	<u>Op</u>	<u>Int</u>		<u>Op</u>	<u>Int</u>	<u>Op</u>	<u>Int</u>
------------	-----------	------------	--	-----------	------------	-----------	------------

Mad Libs for Arithmetic

$$\begin{array}{ccccccc} (& 7 & * & 5 &) & / & 5 & - & 49 \\ \hline & \text{Int} & \text{Op} & \text{Int} & & \text{Op} & \text{Int} & \text{Op} & \text{Int} \end{array}$$

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?

Recursive Mad Libs

Expr

Recursive Mad Libs

Expr

What can an arithmetic expression be?

Recursive Mad Libs

int

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
Expr Op Expr

A single number.

Two expressions joined by an operator.

Recursive Mad Libs

int	+	
<hr/>	<hr/>	<hr/>
Expr	Op	Expr

What can an arithmetic expression be?

int
Expr Op Expr

A single number.

Two expressions joined by an operator.

Recursive Mad Libs

int + Expr
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

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	+			
<hr/>	<hr/>	<hr/>	<hr/>	<hr/>
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		
<hr/>	<hr/>	<hr/>	<hr/>	<hr/>
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	×	
<hr/>	<hr/>	<hr/>	<hr/>	<hr/>
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
<hr/>	<hr/>	<hr/>	<hr/>	<hr/>
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
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(

Expr)

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs



What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(Expr Op Expr)

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(**int** **Op**)
Expr Op Expr

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(**int** **/** **Expr**)
Expr **Op** **Expr**

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(int /)
Expr Op Expr

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(int / (Expr))

Expr **Op** **Expr**

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

$(\text{int} \ / \ (\ \))$

Expr Op Expr

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(int / (Expr))

Expr **Op** **Expr**

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(int / (Expr Op Expr))

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(int / ())

Expr **Op** **Expr** **Op** **Expr**

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(int / (int))

Expr **Op** **Expr** **Op** **Expr**

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(int / (int +))

Expr **Op** **Expr** **Op** **Expr**

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

Recursive Mad Libs

(int / (int + int))

Expr **Op** **Expr** **Op** **Expr**

What can an arithmetic expression be?

int
Expr Op Expr
(Expr)

A single number.

Two expressions joined by an operator.

A parenthesized expression.

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

More on the details of these rules in a bit.

Warning: CFGs are not finite automata - be prepared for something completely different!

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 → **x**

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 → **x**

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**

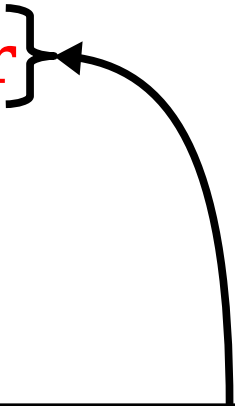
⇒ **int / int**

Arithmetic Expressions

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

Expr \rightarrow **int**
Expr \rightarrow **Expr Op Expr**
Expr \rightarrow (**Expr**)
Op \rightarrow +
Op \rightarrow -
Op \rightarrow ×
Op \rightarrow /

Expr
 \Rightarrow **Expr Op Expr**
 \Rightarrow **Expr Op** **int**
 \Rightarrow **int Op** **int**
 \Rightarrow **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 → **x**

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** (sometimes 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 \rightarrow **int**

Expr \rightarrow **Expr Op Expr**

Expr \rightarrow **(Expr)**

Op \rightarrow **+**

Op \rightarrow **-**

Op \rightarrow **x**

Op \rightarrow **/**

Some CFG Notation

- In today's slides, capital letters in **Bold Red Uppercase** represent nonterminals.
 - e.g. **A**, **B**, **C**, **D**
- Lowercase letters in **blue monospace** represent terminals.
 - e.g. **t**, **u**, **v**, **w**
- Lowercase Greek letters in *gray italics* 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 \rightarrow **int** | **Expr Op Expr** | **(Expr)**

Op \rightarrow **+** | **-** | **×** | **/**

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

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

- That is, $N(G)$ 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

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

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

$$\begin{aligned} Q &\rightarrow Qa \mid dH \\ H &\rightarrow bHb \mid c \end{aligned}$$

Which of the following strings are in $\hat{N}(G)$?

dca
dc
cad
bcb
dHaa

Go to
[PollEv.com/cs103spr25](https://pollev.com/cs103spr25)

Context-Free Languages

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

Context-Free Languages

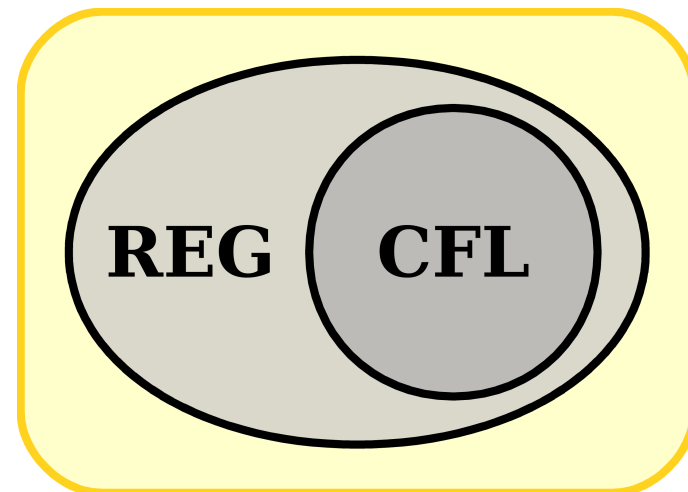
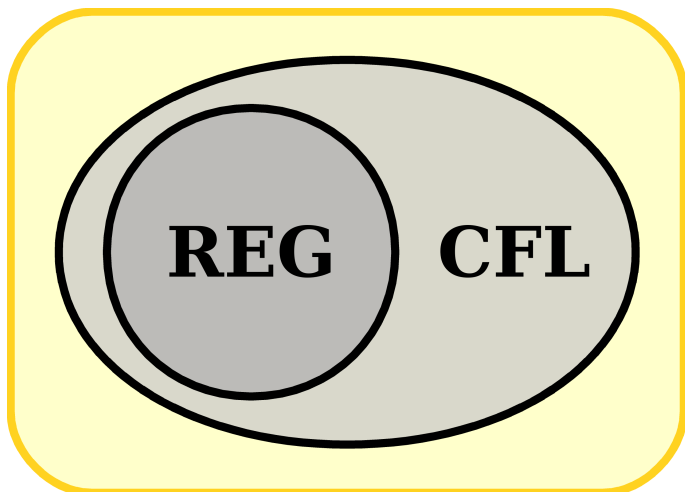
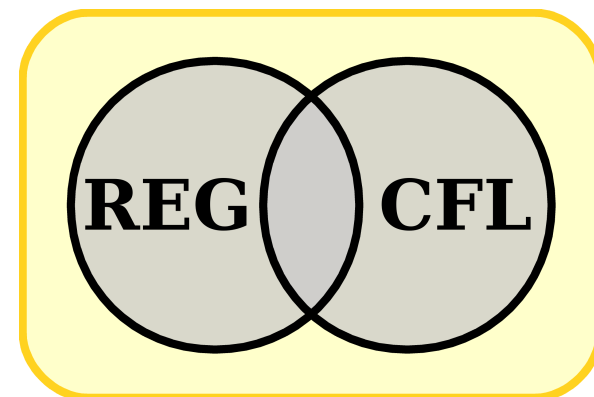
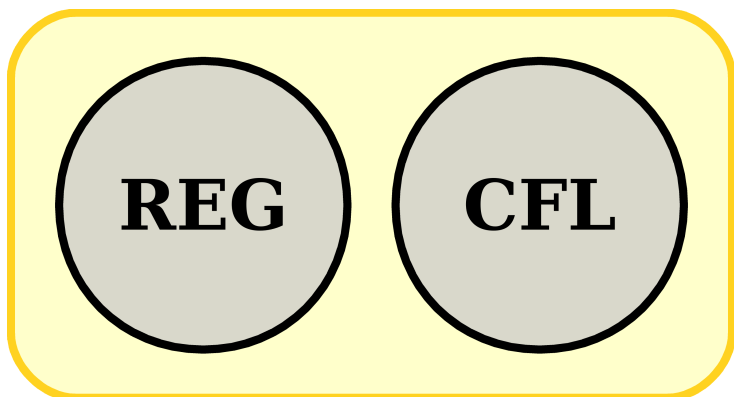
A language L is called a ***context-free language*** (or CFL) if there is a CFG G such that $L = N(G)$.

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 $A \rightarrow \omega$. They do not have the regular expression operators $*$ or \square .
- You can use the symbols $*$ and \square if you'd like in a CFG, but they just stand for themselves.
- Consider this CFG G :

$$S \rightarrow a*b$$

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

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.

a (b \square ϵ) c

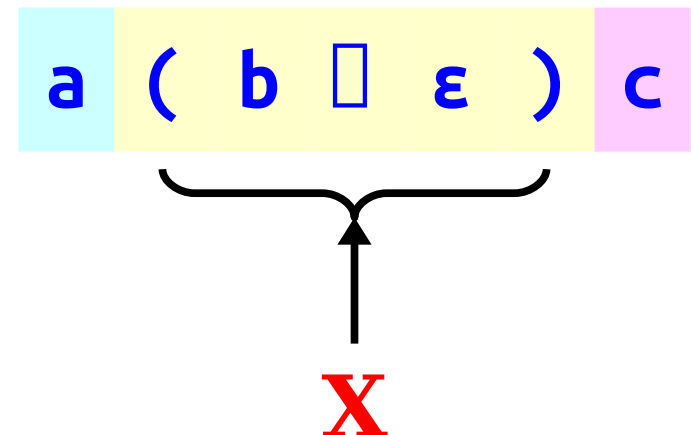
CFGs and Regular Expressions

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

a (b \square ϵ) c

CFGs and Regular Expressions

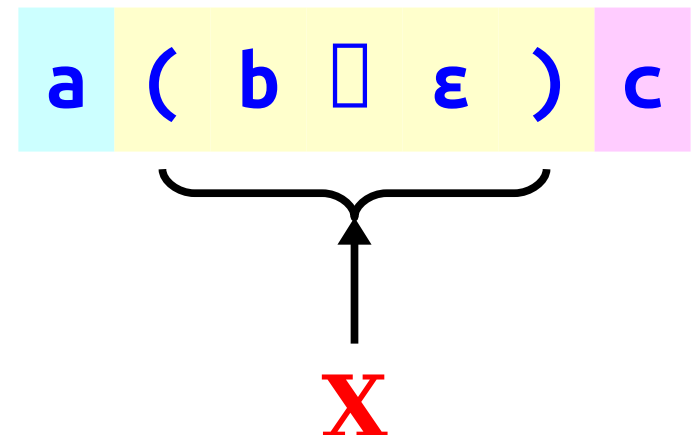
- **Theorem:** Every regular language is context-free.
- **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.



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.

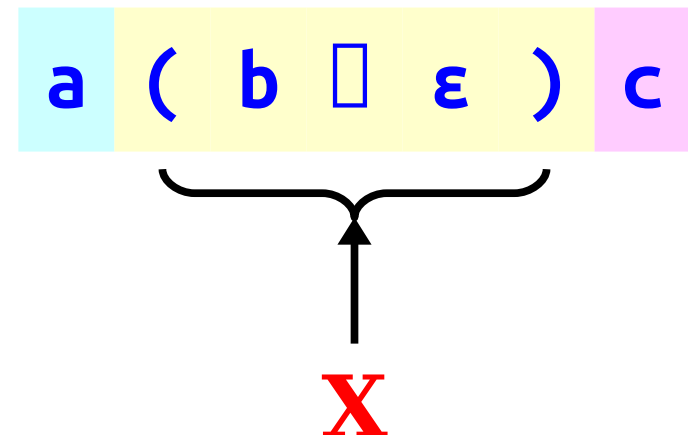
$$S \rightarrow aXc$$



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.

$$\begin{array}{l} S \rightarrow aXc \\ X \rightarrow b \mid \varepsilon \end{array}$$

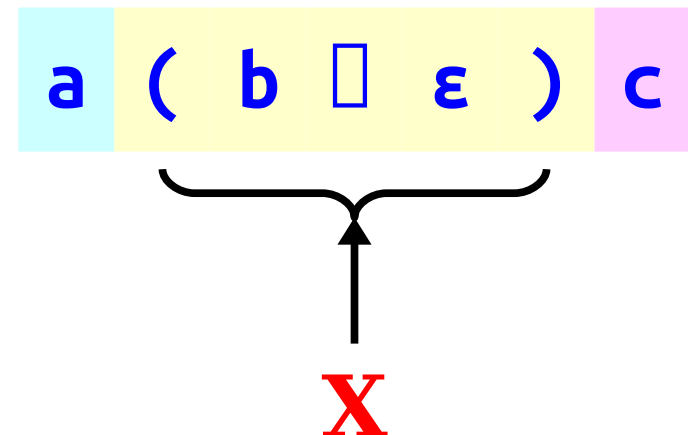


CFGs and Regular Expressions

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

$$\begin{array}{l} S \rightarrow aXc \\ X \rightarrow b \mid \varepsilon \end{array}$$

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.

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

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.

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

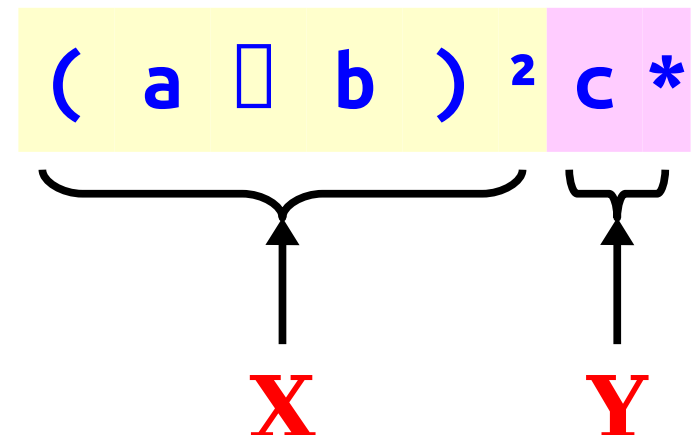
CFGs and Regular Expressions

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

(a □ b) ² c *

CFGs and Regular Expressions

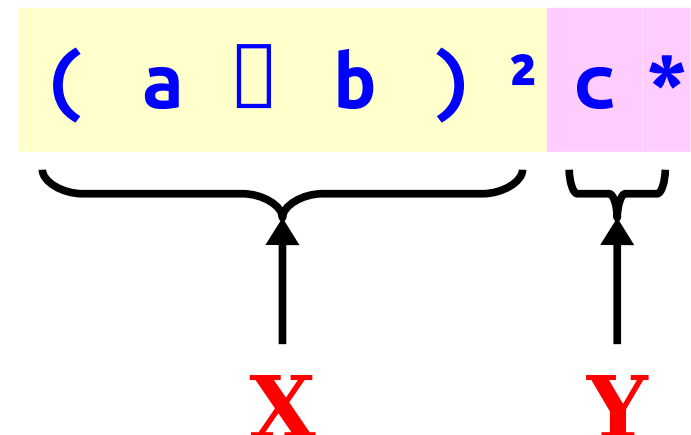
- **Theorem:** Every regular language is context-free.
- **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.



CFGs and Regular Expressions

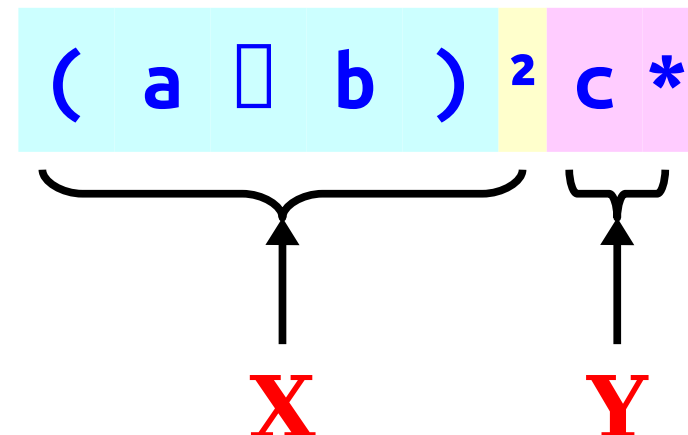
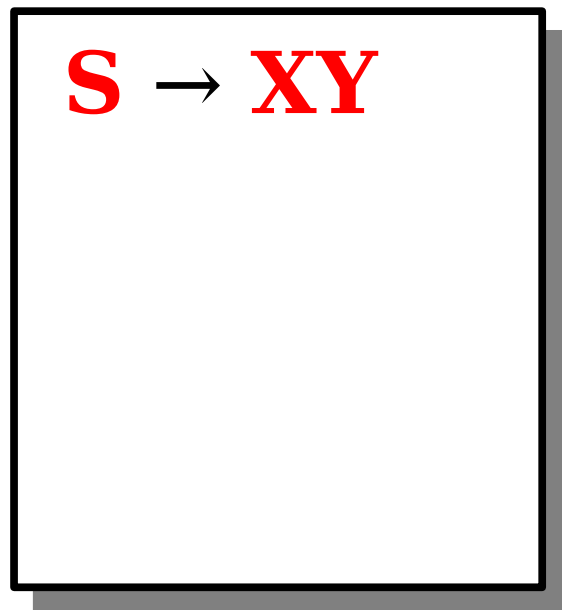
- **Theorem:** Every regular language is context-free.
- **Proof idea:** Show how to convert an arbitrary regular expression into a context-free grammar.

S → **XY**



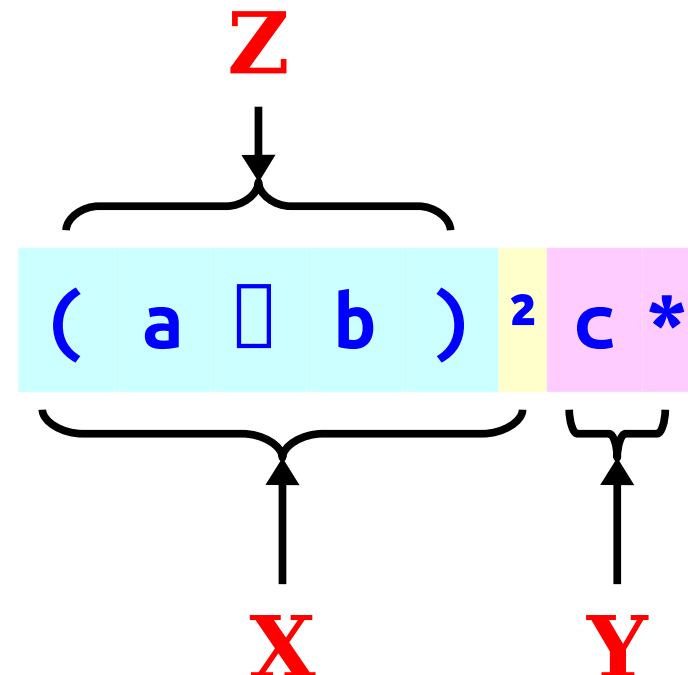
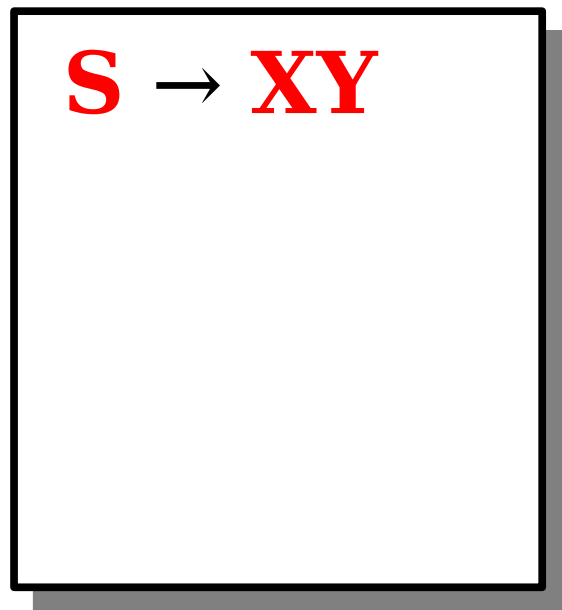
CFGs and Regular Expressions

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



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.

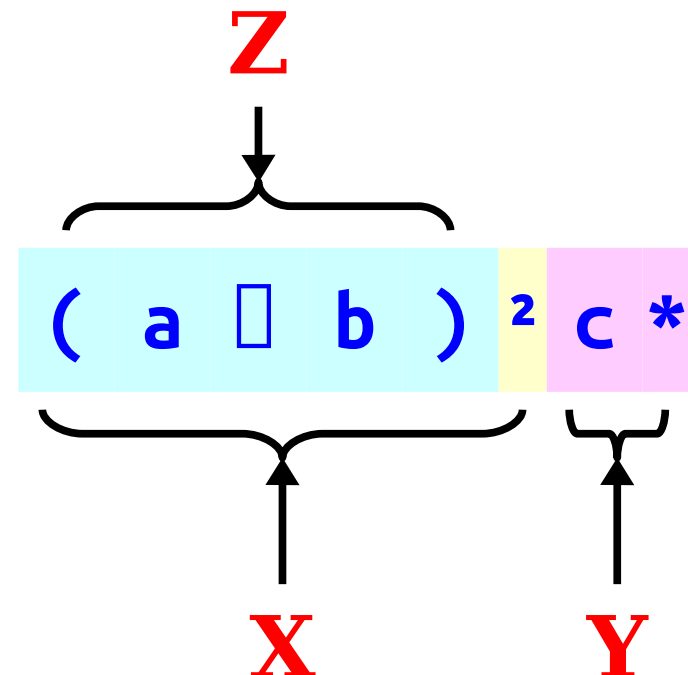


CFGs and Regular Expressions

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

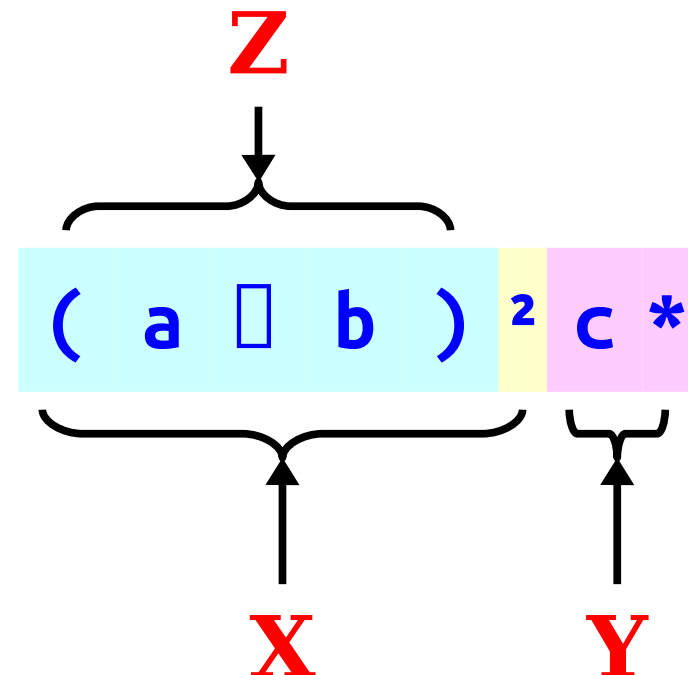
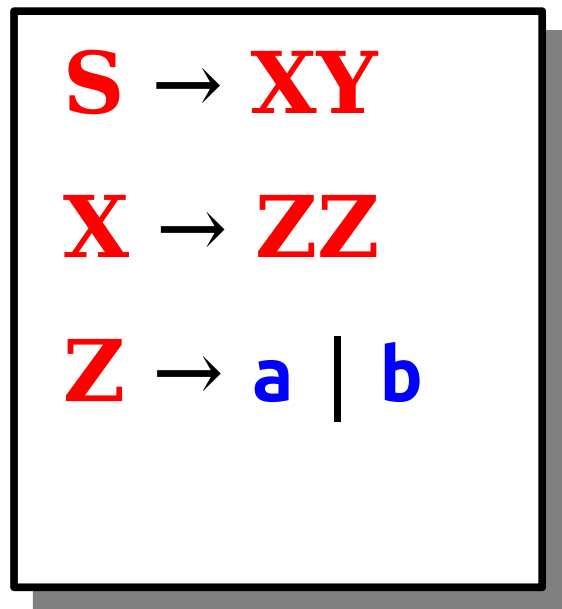
$S \rightarrow XY$

$X \rightarrow ZZ$



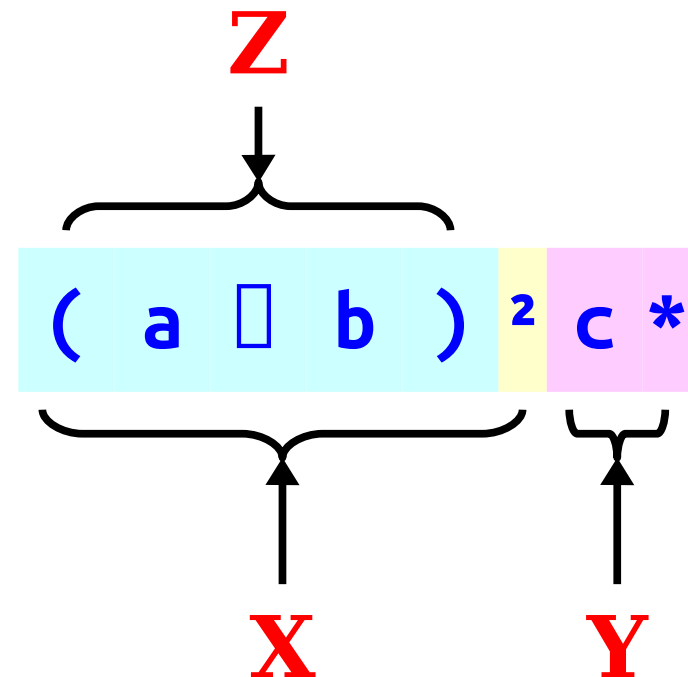
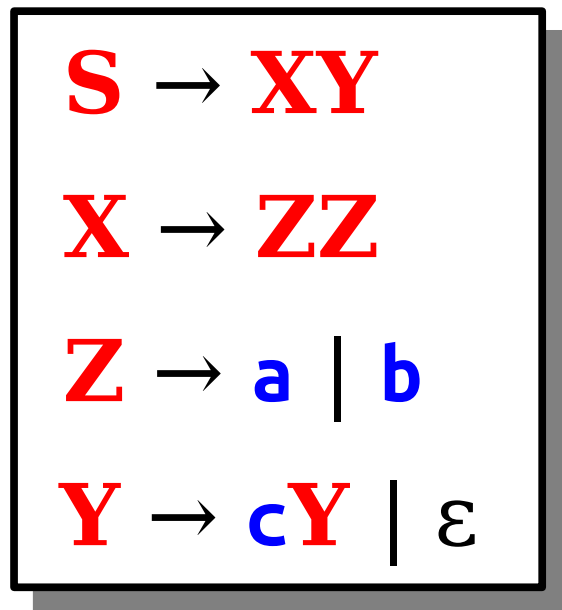
CFGs and Regular Expressions

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

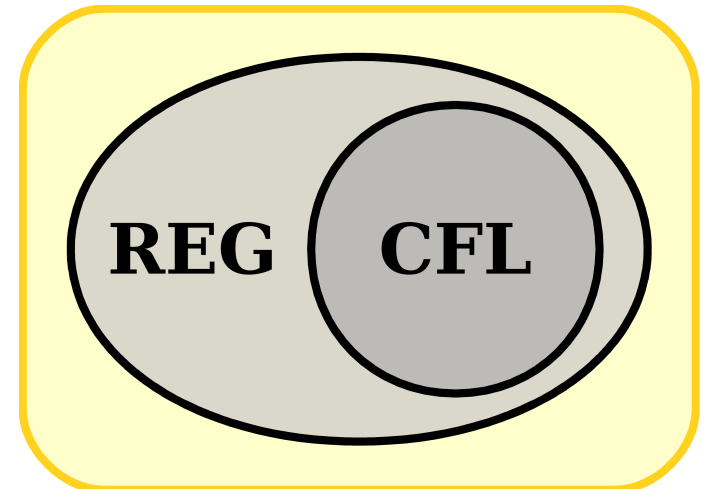
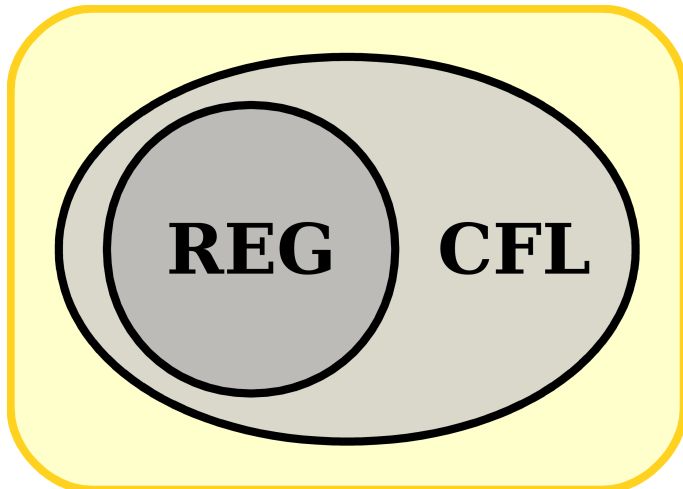
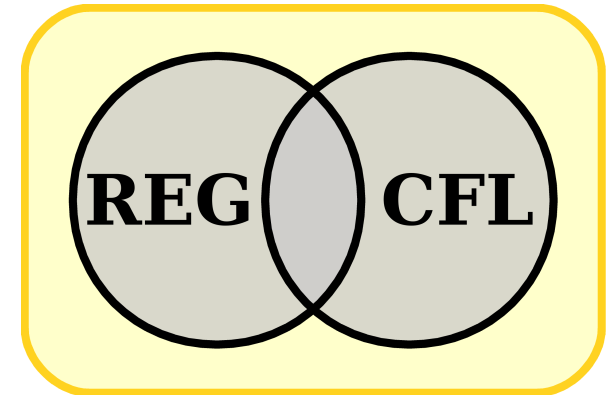
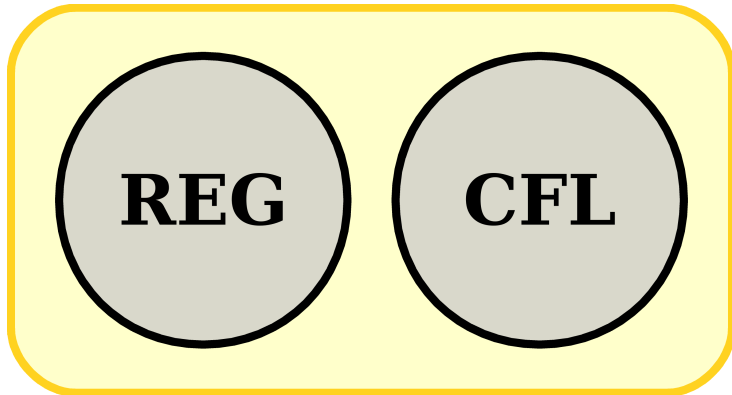


CFGs and Regular Expressions

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



Two ~~Five~~ Possibilities



The Language of a Grammar

- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this grammar generate?

The Language of a Grammar

- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this grammar generate?

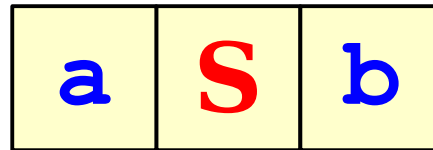
S

The Language of a Grammar

- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this grammar generate?



The Language of a Grammar

- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this grammar generate?

a

S

b

The Language of a Grammar

- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this grammar generate?

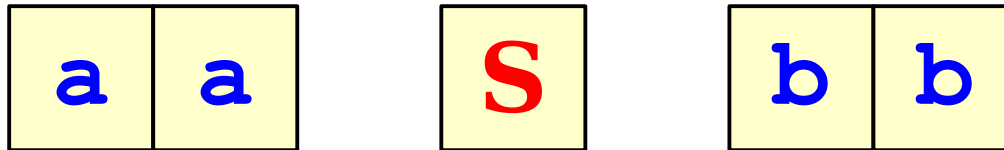
a	a	S	b	b
---	---	---	---	---

The Language of a Grammar

- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this grammar generate?



The Language of a Grammar

- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this grammar generate?

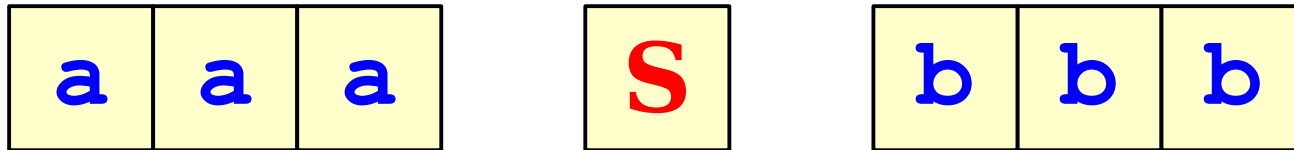
a	a	a	S	b	b	b
---	---	---	---	---	---	---

The Language of a Grammar

- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this grammar generate?



The Language of a Grammar

- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this grammar generate?

a	a	a	a	S	b	b	b	b
---	---	---	---	---	---	---	---	---

The Language of a Grammar

- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this grammar generate?

a	a	a	a
---	---	---	---

b	b	b	b
---	---	---	---

The Language of a Grammar

- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this generate?

a	a	a	a	b	b	b	b
---	---	---	---	---	---	---	---

The Language of a Grammar

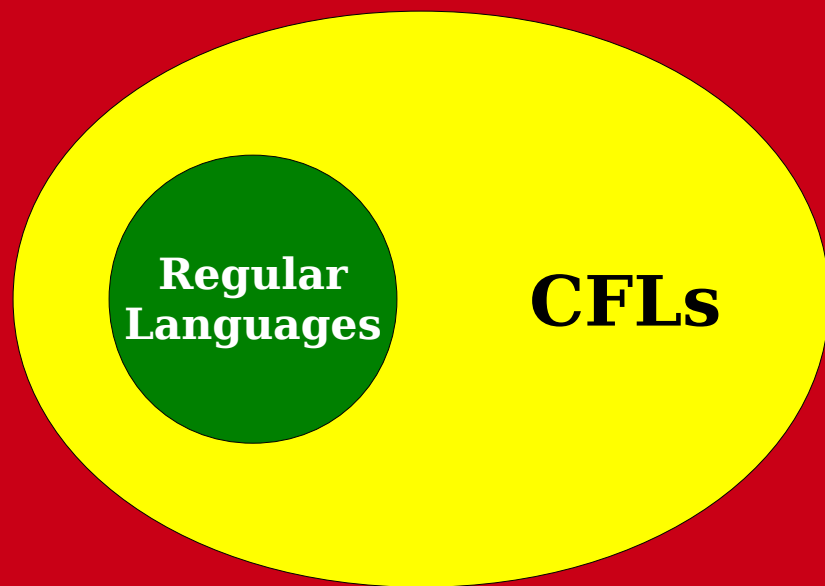
- Consider the following CFG G :

$$S \rightarrow aSb \mid \epsilon$$

- What strings can this generate?

a	a	a	a	b	b	b	b
---	---	---	---	---	---	---	---

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



All Languages

Why the Extra Power?

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

Why the Extra Power?

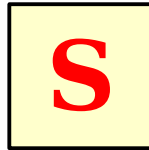
- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded “memory.”

$$S \rightarrow aSb \mid \epsilon$$

Why the Extra Power?

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

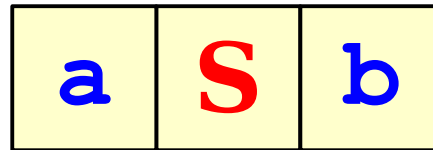
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

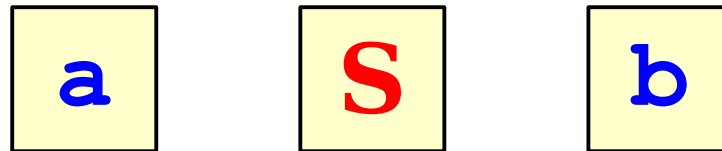
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

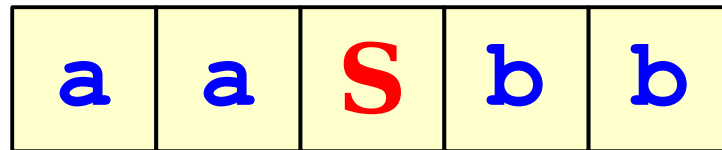
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

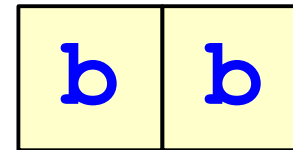
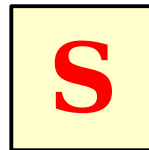
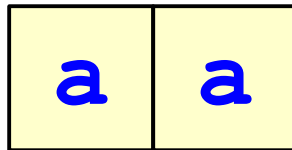
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

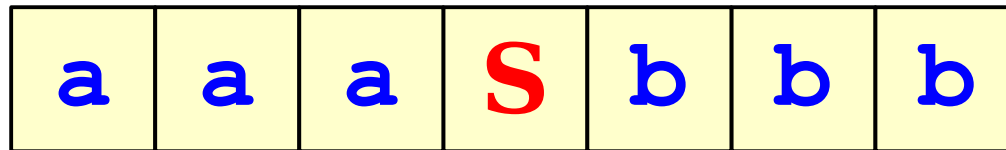
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

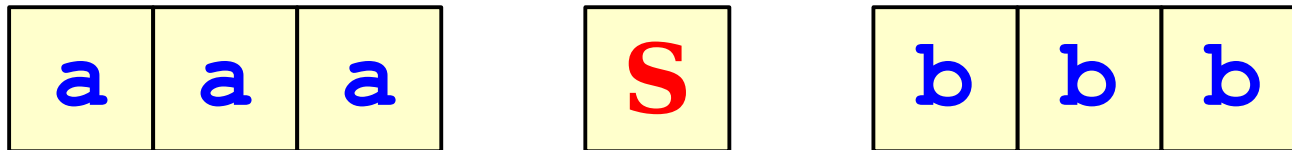
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

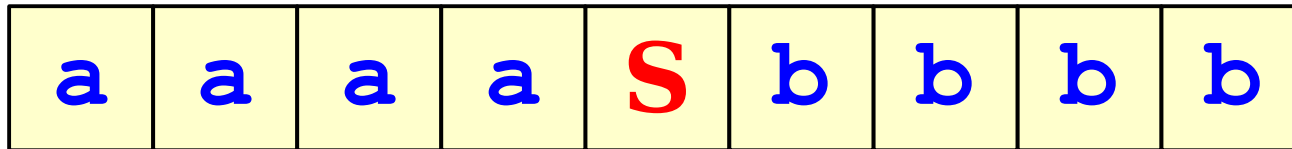
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

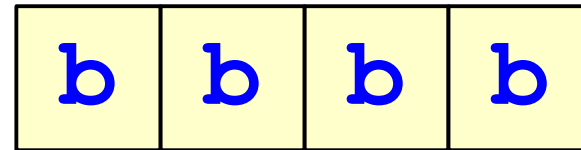
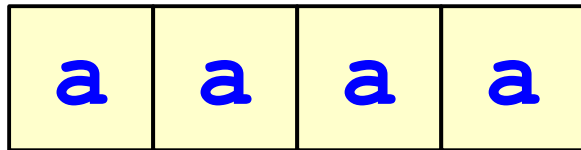
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

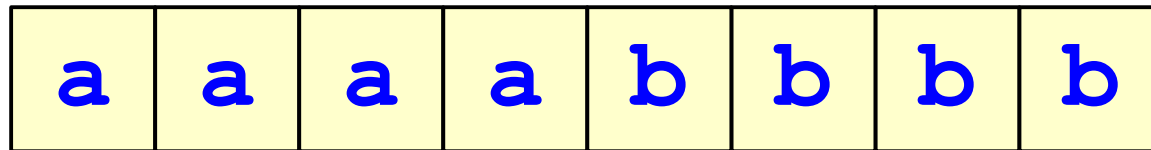
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

$$S \rightarrow aSb \mid \epsilon$$



Time-Out for Announcements!

Second Midterm Logistics

- Our second midterm exam is ***Tuesday, May 20th*** from ***6-9 PM***. Seating assignments will be posted ... soon.
- Topic coverage is primarily lectures 06 – 15 (functions through DFAs & NFAs) and PS3 – PS6.
 - 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: Exam locations will be announced ... soon.

Our Advice

- ***Stay fed and rested.***

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!

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 $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ and let $L = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}$
- We can design a CFG for L by thinking inductively:
 - Base case: ε , \mathbf{a} , and \mathbf{b} are palindromes.
 - If ω is a palindrome, then $\mathbf{a}\omega\mathbf{a}$ and $\mathbf{b}\omega\mathbf{b}$ are palindromes.
 - No other strings are palindromes.

$$\mathbf{S} \rightarrow \varepsilon \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{aSa} \mid \mathbf{bSb}$$

Designing CFGs

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

$\{\{\{\}\}\}$

$\{\{\}\}\{\}$

$\{\{\}\}\{\{\}\}\{\{\}\}\{\}$

$\{\{\{\{\}\}\}\}\{\{\}\}\{\}$

ϵ

$\{\}\{\}$

Designing CFGs

- Let $\Sigma = \{\{, \}\}$ and let $L = \{w \in \Sigma^* \mid w \text{ is a string of balanced braces}\}$
- 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 $L = \{w \in \Sigma^* \mid w \text{ is a string of balanced braces}\}$
- 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 $L = \{ w \in \Sigma^* \mid w \text{ is a string of balanced braces} \}$
- 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.

The diagram shows a large blue curly brace on the left, followed by a sequence of 10 blue curly braces. A vertical dashed red line separates this sequence from another sequence of 8 blue curly braces on the right. The entire sequence is enclosed in a large blue curly brace on the far right.

Designing CFGs

- Let $\Sigma = \{\{, \}\}$ and let $L = \{w \in \Sigma^* \mid w \text{ is a string of balanced braces}\}$
- 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 $L = \{w \in \Sigma^* \mid w \text{ is a string of balanced braces}\}$
- 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.

$$S \rightarrow \{S\}S \mid \epsilon$$

Designing CFGs

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

Which of these CFGs have language L ?

$S \rightarrow aSb \mid bSa \mid \epsilon$

$S \rightarrow abS \mid baS \mid \epsilon$

$S \rightarrow abSba \mid baSab \mid \epsilon$

$S \rightarrow SbaS \mid SabS \mid \epsilon$

Go to
[PollEv.com/cs103spr25](https://pollev.com/cs103spr25)

Designing CFGs

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

Which of these CFGs have language L ?

$S \rightarrow aSb \mid bSa \mid \epsilon$

$S \rightarrow abS \mid baS \mid \epsilon$

$S \rightarrow abSba \mid baSab \mid \epsilon$

$S \rightarrow SbaS \mid SabS \mid \epsilon$

Designing CFGs

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

Which of these CFGs have language L ?

$S \rightarrow aSb \mid bSa \mid \epsilon$

$S \rightarrow abS \mid baS \mid \epsilon$

$S \rightarrow abSba \mid baSab \mid \epsilon$

$S \rightarrow SbaS \mid SabS \mid \epsilon$

Designing CFGs

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

Which of these CFGs have language L ?

$S \rightarrow aSb \mid bSa \mid \epsilon$

$S \rightarrow abS \mid baS \mid \epsilon$

$S \rightarrow abSba \mid baSab \mid \epsilon$

$S \rightarrow SbaS \mid SabS \mid \epsilon$

Designing CFGs

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

Which of these CFGs have language L ?

$S \rightarrow aSb \mid bSa \mid \epsilon$

$S \rightarrow abS \mid baS \mid \epsilon$

$S \rightarrow abSba \mid baSab \mid \epsilon$

$S \rightarrow SbaS \mid SabS \mid \epsilon$

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!

CFG Caveats II

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

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

- What strings in $\{\mathbf{a}, \mathbf{b}\}^*$ can you derive?
 - Answer: ***None!***
- What is the language of the grammar?
 - Answer: **\emptyset**
- 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 $\Sigma = \{a, \square\}$ and let $L = \{a^n \square a^n \mid n \in \mathbb{N}\}$.
- Is the following a CFG for L ?

$$S \rightarrow X \square X$$

$$X \rightarrow aX \mid \epsilon$$

$$\begin{aligned}
 &S \\
 \Rightarrow &X \square X \\
 \Rightarrow &aX \square X \\
 \Rightarrow &aaX \square X \\
 \Rightarrow &aa \square X \\
 \Rightarrow &aa \square \cancel{a} \\
 \Rightarrow &aa \square a
 \end{aligned}$$

Finding a Build Order

- Let $\Sigma = \{a, \square\}$ and let $L = \{a^n \square a^n \mid n \in \mathbb{N}\}$.
- 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 that implements that idea:

$$S \rightarrow \square \mid aSa$$

$$\begin{aligned}
 &S \\
 \Rightarrow &aSa \\
 \Rightarrow &aaSaa \\
 \Rightarrow &aaaSaaa \\
 \Rightarrow &aaa\square aaa
 \end{aligned}$$

Summary of CFG Design Tips

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

Applications of Context-Free Grammars

CFGs for Programming Languages

BLOCK	→	STMT { STMTS }
STMTS	→	ϵ STMT STMTS
STMT	→	EXPR; if (EXPR) BLOCK while (EXPR) BLOCK do BLOCK while (EXPR); BLOCK ...
EXPR	→	identifier constant EXPR + EXPR EXPR - EXPR EXPR * EXPR ...

Grammars in Compilers

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

Natural Language Processing

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

Next Time

Turing Machines

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