Context-Free Grammars

First: finish up showing a language is NOT regular, using the Myhill-Nerode Theorem

The Myhill-Nerode Theorem

Theorem: Let L be a language over Σ. If there is a set $S \subseteq \Sigma^*$ with the following properties, then L is not regular:

- S is infinite (that is, S contains infinitely many strings).
- The strings in S are *pairwise distinguishable relative to* L. That is,

$$\forall x \in S.\ \forall y \in S.\ (x \neq y \rightarrow x \not\equiv_L y).$$

Proof: Let L be an arbitrary language over Σ. Let $S \subseteq \Sigma^*$ be an infinite set of strings with the following property: if $x, y \in S$ and $x \neq y$, then $x \not\equiv_L y$. We will show that L is not regular.

Suppose for the sake of contradiction that L is regular. This means that there must be some DFA D for L. Let k be the number of states in D. Since there are infinitely many strings in S, we can choose $k+1$ distinct strings from S and consider what happens when we run D on all of those strings. Because there are only k states in D and we've chosen $k+1$ strings from S, by the pigeonhole principle we know that at least two strings from S must end in the same state in D. Choose any two such strings and call them x and y.

Because x and y are distinct strings in S, we know that $x \not\equiv_L y$. Our earlier theorem therefore tells us that when we run D on inputs x and y, they must end up in different states. But this is impossible – we chose x and y precisely because they end in the same state when run through D.

We have reached a contradiction, so our assumption must have been wrong. Thus L is not a regular language. ■

Using the Myhill-Nerode Theorem

Theorem: The language $E = \{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$ is not regular.

Theorem: The language $E = \{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$ is not regular.

To use the Myhill-Nerode theorem, we need to find an infinite set of strings that are pairwise distinguishable relative to E.

Theorem: The language $E = \{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$ is not regular.

To use the Myhill-Nerode theorem, we need to find an infinite set of strings that are pairwise distinguishable relative to E.

We know that any two strings of the form $\mathbf{a}^n$ and $\mathbf{a}^m$, where $n \neq m$, are distinguishable.

Theorem: The language $E = \{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$ is not regular.

To use the Myhill-Nerode theorem, we need to find an infinite set of strings that are pairwise distinguishable relative to E.

We know that any two strings of the form $\mathbf{a}^n$ and $\mathbf{a}^m$, where $n \neq m$, are distinguishable.

So pick the set $S = \{ \mathbf{a}^n \mid n \in \mathbb{N} \}$.

Theorem: The language $E = \{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$ is not regular.

To use the Myhill-Nerode theorem, we need to find an infinite set of strings that are pairwise distinguishable relative to E.

We know that any two strings of the form $\mathbf{a}^n$ and $\mathbf{a}^m$, where $n \neq m$, are distinguishable.

So pick the set $S = \{ \mathbf{a}^n \mid n \in \mathbb{N} \}$.

Notice that S isn't a subset of E. That's okay: we never said that S needs to be a subset of E!

Approaching Myhill-Nerode

- The challenge in using the Myhill-Nerode theorem is finding the right set of strings to use.
- ***General intuition:***
  - Start by thinking about what information a computer “must” remember in order to answer correctly.
  - Choose a group of strings that all require different information.
  - Prove that those strings are distinguishable relative to the language in question.

Tying Everything Together

- One of the intuitions we hope you develop for DFAs is to have each state in a DFA represent some key piece of information the automaton has to remember.
- If you only need to remember one of finitely many pieces of information, that gives you a DFA.
  - ***You can formalize this!*** Take CS154 for details.
- If you need to remember one of infinitely many pieces of information, you can use the Myhill-Nerode theorem to prove that the language has no DFA.

Where We're Going

- What does computation look like with unbounded memory?
- What problems can you solve with unbounded-memory computers?
- What does it even mean to “solve” such a problem?
- And how do we know the answers to any of these questions?

Context-Free Grammars

Describing Languages

- We've seen two models for the regular languages:
  - ***Finite automata*** accept precisely the strings in the language.
  - ***Regular expressions*** describe precisely the strings in the language.
- Finite automata ***recognize*** strings in the language.
  - Perform a computation to determine whether a specific string is in the language.
- Regular expressions ***match*** strings in the language.
  - Describe the general shape of all strings in the language.

Context-Free Grammars

- A ***context-free grammar*** (or ***CFG***) is an entirely different formalism for defining a class of languages.
- ***Goal:*** Give a description of a language by recursively describing the structure of the strings in the language.
- CFGs are best explained by example...

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

E → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

⇒ **E** `*` `(`**E Op E**`)`

⇒ `int *` `(`**E Op E**`)`

⇒ `int * (int` **Op E**`)`

⇒ `int * (int` **Op** `int)`

⇒ `int * (int + int)`

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

E → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

E
⇒ **E Op E**
⇒ **E Op** `int`
⇒ `int` **Op** `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)

E → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

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.

E → `int`

E → **E Op E**

E → `(`**E**`)`

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

E → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

A Notational Shorthand

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

Derivations

E → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

E
⇒ **E Op E**
⇒ **E Op** `(`**E**`)`
⇒ **E Op** `(`**E Op E**`)`
⇒ **E** `*` `(`**E Op E**`)`
⇒ `int *` `(`**E Op E**`)`
⇒ `int * (int` **Op E**`)`
⇒ `int * (int` **Op** `int)`
⇒ `int * (int + int)`

- A sequence of 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 **E** $\Rightarrow^*$ `int * (int + int)`.

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 $\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$:

$$\mathbf{S} \to \mathbf{Sa} \mid \mathbf{dT}$$

$$\mathbf{T} \to \mathbf{bTb} \mid \mathbf{c}$$

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

dc **dca** **cad**

bcb **dTaa**

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **a number**.

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:
  - What languages are context-free?
  - How are context-free and regular languages related?

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → **`a*b`**

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → **a*** b

From Regexes to CFGs

- CFGs consist purely of production rules of the form $A \to \omega$. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

$$S \to a^*b$$

$$A \to Aa \mid \varepsilon$$

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ω. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → **a*b**

A → **A**a | ε

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → **A**`b`

A → **A**`a` | **ε**

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ω. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → **a(b ∪ c*)**

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

$$S \rightarrow a(b \cup c^*)$$

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

$$S \rightarrow a(b \cup c^*)$$

$$X \rightarrow b \mid c^*$$

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ω. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → `a(b ∪ c*)`

X → `b | c*`

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ω. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → **a**X

X → **b** | **c***

From Regexes to CFGs

- CFGs consist purely of production rules of the form $A \rightarrow \omega$. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

$$S \rightarrow aX$$
$$X \rightarrow b \mid c^*$$

From Regexes to CFGs

- CFGs consist purely of production rules of the form $A \rightarrow \omega$. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

$$S \rightarrow aX$$
$$X \rightarrow b \mid c^*$$
$$C \rightarrow Cc \mid \varepsilon$$

From Regexes to CFGs

- CFGs consist purely of production rules of the form $A \rightarrow \omega$. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

$$S \rightarrow aX$$
$$X \rightarrow b \mid c^*$$
$$C \rightarrow Cc \mid \varepsilon$$

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ω. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → **a****X**

X → **b** | **C**

C → **C****c** | **ε**

Regular Languages and CFLs

- ***Theorem:*** Every regular language is context-free.
- ***Proof Idea:*** Use the construction from the previous slides to convert a regular expression for L into a CFG for L. ■
- ***Problem Set 8 Exercise:*** Instead, show how to convert a DFA/NFA into a CFG.

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?

The Language of a Grammar

- Consider the following CFG *G*:

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

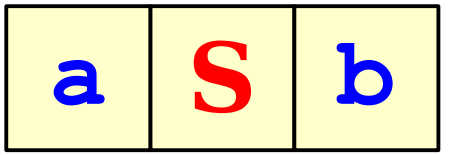
- What strings can this generate?

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?



The Language of a Grammar

- Consider the following CFG *G*:

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

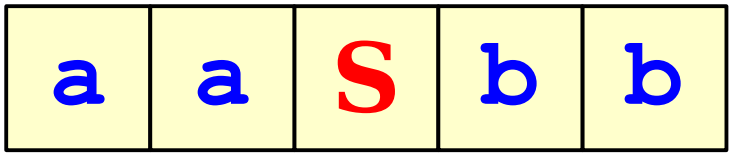
- What strings can this generate?

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?



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?

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?

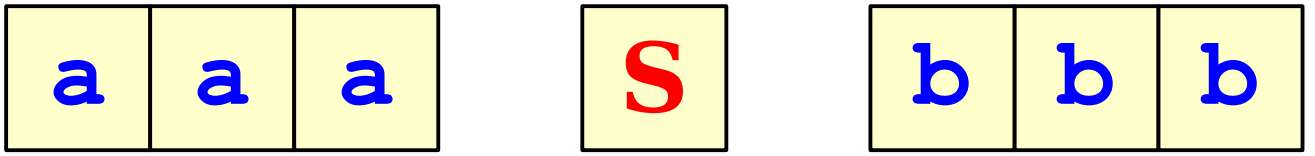
a	a	a	S	b	b	b

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?

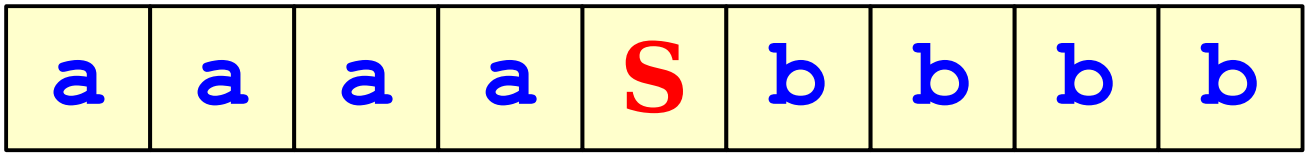


The Language of a Grammar

- Consider the following CFG G:

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

- What strings can this generate?



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

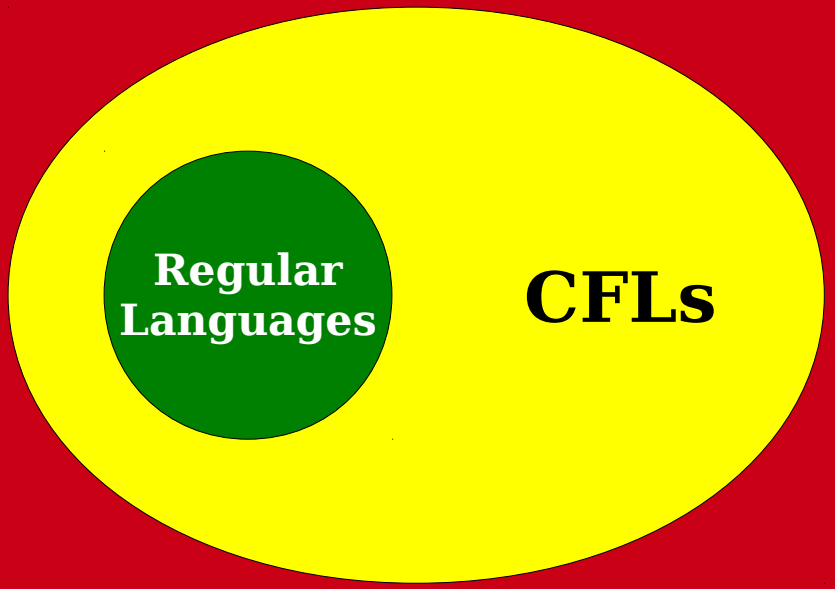
- Consider the following CFG G:

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

- What strings can this generate?

a	a	a	a	b	b	b	b

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



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."

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

Why the Extra Power?

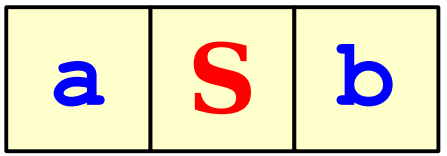
- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded "memory."

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

Why the Extra Power?

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

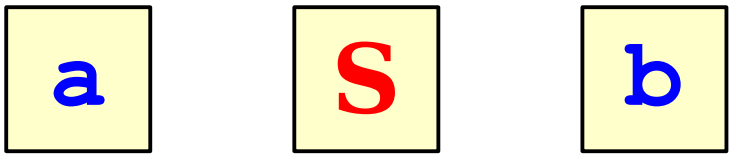
$$S \rightarrow aSb \mid \varepsilon$$



Why the Extra Power?

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

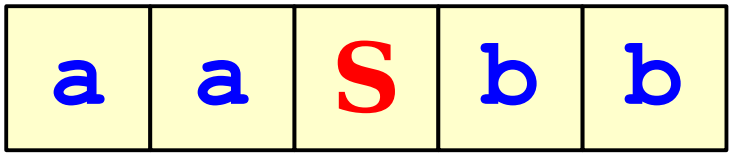
$$S \rightarrow aSb \mid \varepsilon$$



Why the Extra Power?

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

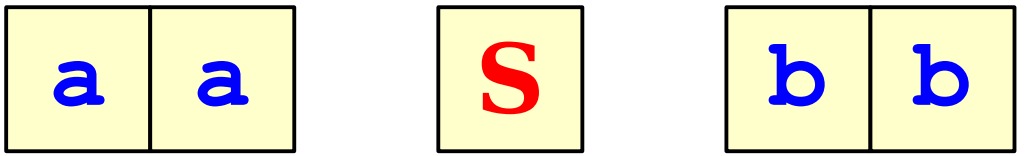
$$S \rightarrow aSb \mid \varepsilon$$



Why the Extra Power?

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

S → **a****S****b** | **ε**



Why the Extra Power?

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

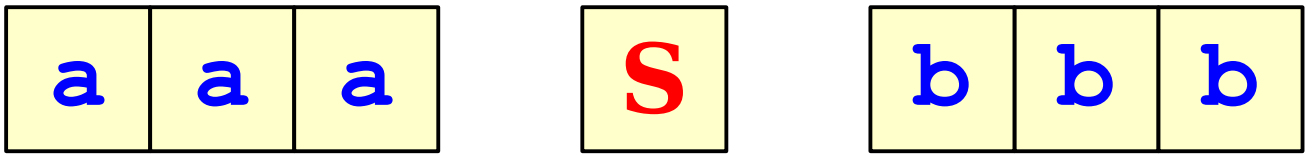
S → aSb | ε

a	a	a	S	b	b	b

Why the Extra Power?

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

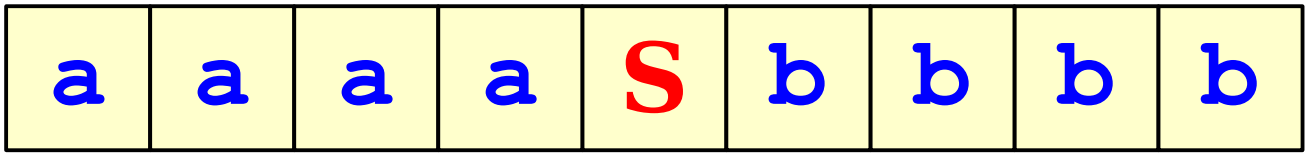
S → **a**S**b** | **ε**



Why the Extra Power?

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

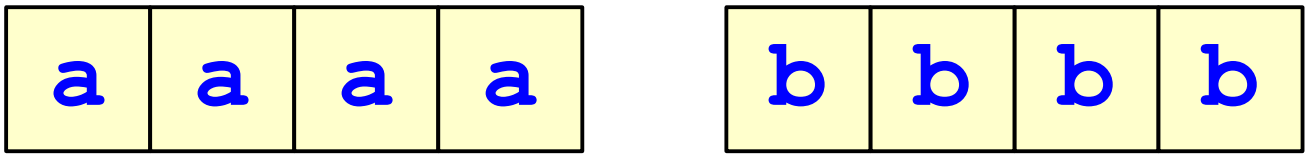
$$S \rightarrow aSb \mid \varepsilon$$



Why the Extra Power?

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

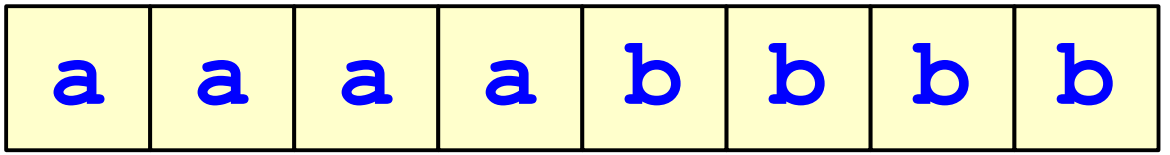
$$S \rightarrow aSb \mid \varepsilon$$



Why the Extra Power?

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

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



Time-Out for Announcements!

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

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

Designing CFGs

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

(())()

(()())(()())

((((()))(()))))

ε

()()

Designing CFGs

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

((()(()))(()))(()))((()))

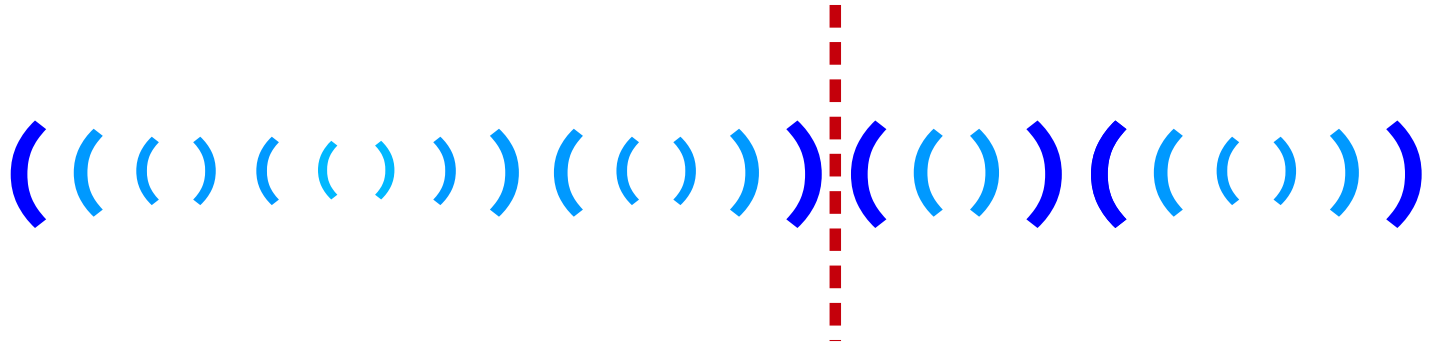
Designing CFGs

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

(((()(()))(()))(()))(((()))())

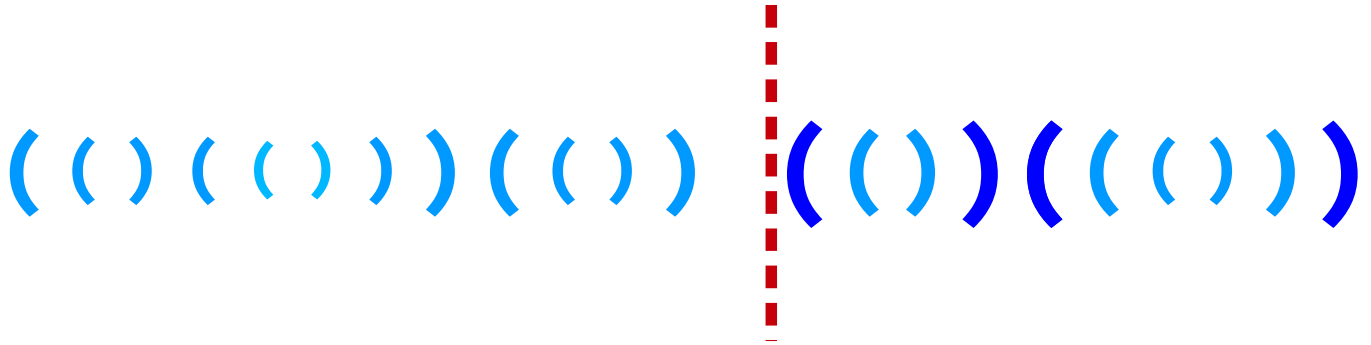
Designing CFGs

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



Designing CFGs

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



Designing CFGs

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

$$\mathbf{S} \rightarrow \mathbf{(S)S} \mid \boldsymbol{\varepsilon}$$

Designing CFGs

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

How many of the following CFGs have language L?

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

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

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

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

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **a number**.

Designing CFGs

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

How many of the following CFGs have language L?

S → **a**S**b** | **b**S**a** | **ε**

S → **ab**S | **ba**S | **ε**

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

S → S**ba**S | S**ab**S | **ε**

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **a number**.

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**

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **a number**.

Designing CFGs

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

How many of the following CFGs have language L?

S → **a**S**b** | **b**S**a** | **ε**

S → **ab**S | **ba**S | **ε**

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

S → S**ba**S | S**ab**S | **ε**

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **a number**.

Designing CFGs

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

How many of the following CFGs have language L?

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

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

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

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

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **a number**.

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 $\{ \mathbf{a}^n\mathbf{b}^n \mid n \in \mathbb{N} \}$?

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

- 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 $\Sigma = \{\mathbf{a}, \stackrel{?}{=}\}$ and let $L = \{\mathbf{a}^n \stackrel{?}{=} \mathbf{a}^n \mid n \in \mathbb{N}\}$.
- Is the following a CFG for L?

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

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

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

Finding a Build Order

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

$$\mathbf{S} \to \stackrel{?}{=} \mid \mathbf{aSa}$$

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

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***.
- Stanford's CoreNLP project is one place to look for an example of this.
- Want to learn more? Take CS124 or CS224N!

Biography Minute:
Noam Chomsky

PC: Hans Peters / Anefo (via Wikimedia)

- Invented CFGs!
- Helped found fields of linguistics and cognitive science

- Today, perhaps more well known for political writing than linguistics
  - Made it onto President Nixon's "Enemies List"
  - Anti-capitalism, anti-imperialism, anti-war
  - Drawing on linguistics expertise, written extensively on state propaganda (*Manufacturing Consent*)

Next Time

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