

# 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 procedure for listing off all 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 objects:
  - 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 converted by a string of terminals and nonterminals, and
  - A **start symbol** (which must be a nonterminal) that begins the derivation.

$$E \rightarrow \text{int}$$

$$E \rightarrow E \text{ Op } E$$

$$E \rightarrow (E)$$

$$\text{Op} \rightarrow +$$

$$\text{Op} \rightarrow -$$

$$\text{Op} \rightarrow *$$

$$\text{Op} \rightarrow /$$

# Some CFG Notation

- Capital letters in **Bold Red Uppercase** will represent nonterminals.
  - i.e. **A, B, C, D**
- Lowercase letters in **blue monospace** will represent terminals.
  - i.e. **t, u, v, w**
- Lowercase Greek letters in *gray italics* will represent arbitrary strings of terminals and nonterminals.
  - i.e.  *$\alpha, \gamma, \omega$*

# 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

$\begin{aligned} \mathbf{E} &\rightarrow \mathbf{E Op E} \mid \mathbf{int} \mid (\mathbf{E}) \\ \mathbf{Op} &\rightarrow + \mid * \mid - \mid / \end{aligned}$
--

$\mathbf{E}$   
 $\Rightarrow \mathbf{E Op E}$   
 $\Rightarrow \mathbf{E Op (E)}$   
 $\Rightarrow \mathbf{E Op (E Op E)}$   
 $\Rightarrow \mathbf{E * (E Op E)}$   
 $\Rightarrow \mathbf{int * (E Op E)}$   
 $\Rightarrow \mathbf{int * (int Op E)}$   
 $\Rightarrow \mathbf{int * (int Op int)}$   
 $\Rightarrow \mathbf{int * (int + int)}$

- A sequence of steps where nonterminals are replaced by the right-hand side of a production is called a *derivation*.
- If string  $\alpha$  derives string  $\omega$ , we write  $\alpha \Rightarrow^* \omega$ .
- In the example on the left, we see  $\mathbf{E} \Rightarrow^* \mathbf{int * (int + int)}$ .

# The Language of a Grammar

- If  $G$  is a CFG with alphabet  $\Sigma$  and start symbol  $\mathbf{S}$ , then the *language of  $G$*  is the set

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

- That is,  $\mathcal{L}(G)$  is the set of strings derivable from the start symbol.
- Note:  $\omega$  must be in  $\Sigma^*$ , the set of strings made from terminals. Strings involving nonterminals aren't in the language.

# Context-Free Languages

- A language  $L$  is called a ***context-free language*** (or CFL) if there is a CFG  $G$  such that  $L = \mathcal{L}(G)$ .
- 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  $A \rightarrow \omega$ . They do not have the regular expression operators  $*$  or  $|$ .
- However, we can convert regular expressions to CFGs as follows:

$$S \rightarrow a^*b$$

# 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:

$$S \rightarrow Ab$$

# 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:

$$\begin{aligned} S &\rightarrow Ab \\ A &\rightarrow Aa \mid \epsilon \end{aligned}$$

# 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:

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



# 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:

$$S \rightarrow aX$$

$$X \rightarrow (b | c^*)$$

# 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:

$$S \rightarrow aX$$
$$X \rightarrow b \mid C$$

# 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:

$$S \rightarrow aX$$

$$X \rightarrow b \mid C$$

$$C \rightarrow Cc \mid \epsilon$$

# 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  $L$ . ■
- ***Fun exercise:*** Instead, show how to convert a DFA/NFA into a CFG.

# The Language of a Grammar

- Consider the following CFG  $G$ :

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

- What strings can this generate?

# The Language of a Grammar

- Consider the following CFG  $G$ :

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

- What strings can this generate?

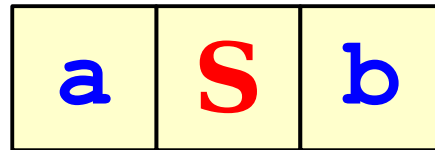
**S**

# The Language of a Grammar

- Consider the following CFG  $G$ :

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

- What strings can this generate?





# The Language of a Grammar

- Consider the following CFG  $G$ :

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

- What strings can this generate?

a

S

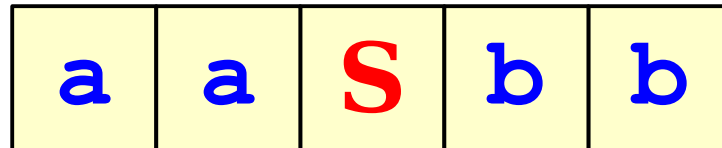
b

# The Language of a Grammar

- Consider the following CFG  $G$ :

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

- What strings can this generate?

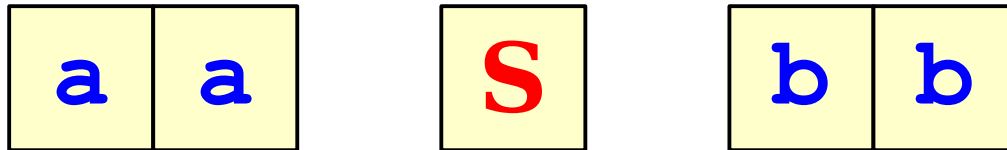


# The Language of a Grammar

- Consider the following CFG  $G$ :

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

- What strings can this generate?

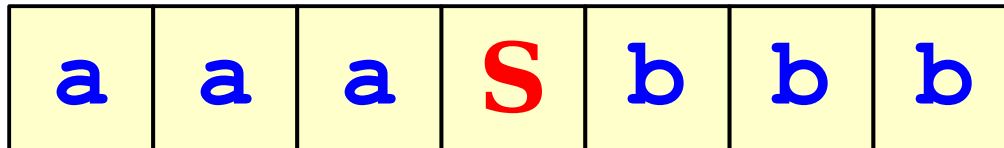


# The Language of a Grammar

- Consider the following CFG  $G$ :

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

- What strings can this generate?

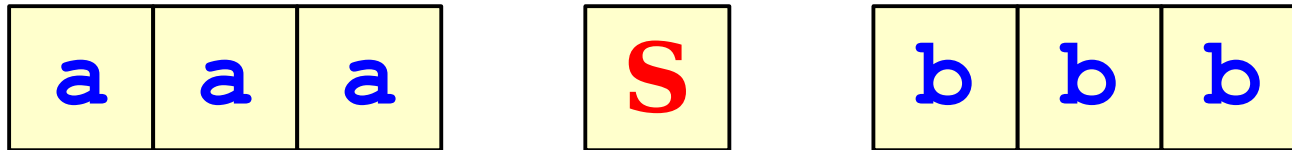


# The Language of a Grammar

- Consider the following CFG  $G$ :

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

- What strings can this generate?

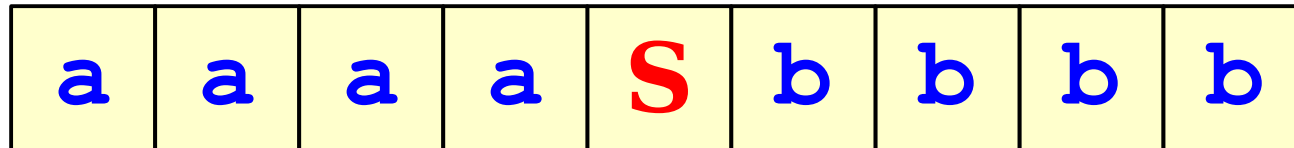


# The Language of a Grammar

- Consider the following CFG  $G$ :

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

- What strings can this generate?



# 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
---	---	---	---	---	---	---	---

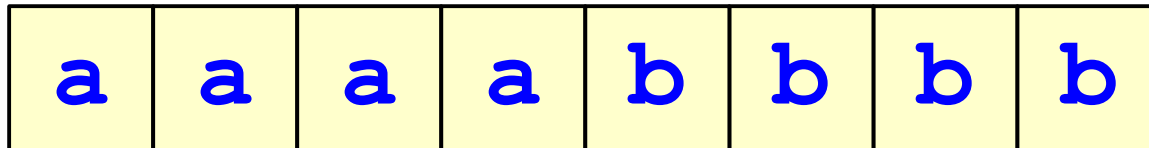


# The Language of a Grammar

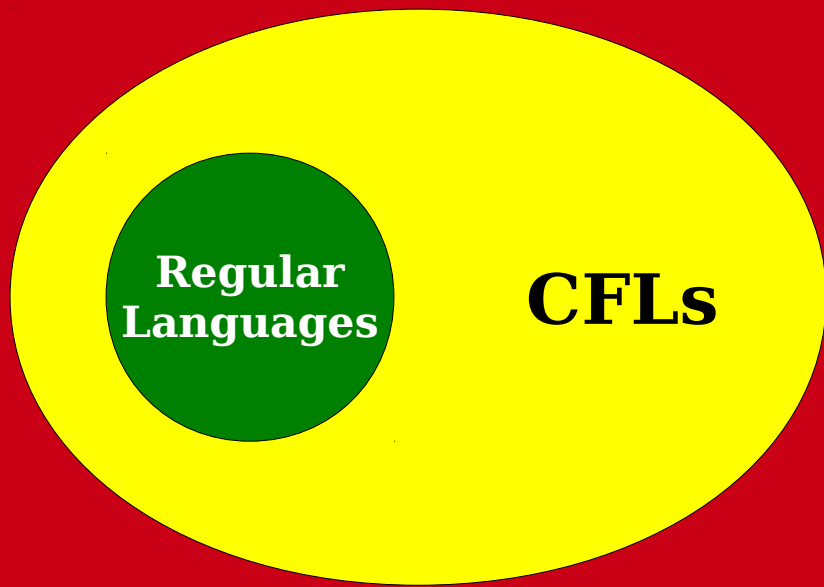
- Consider the following CFG  $G$ :

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

- What strings can this generate?



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



**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?

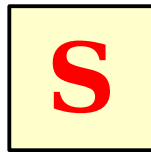
- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded “memory.”

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

# Why the Extra Power?

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

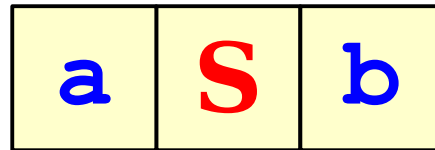
$$S \rightarrow aSb \mid \epsilon$$



# Why the Extra Power?

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

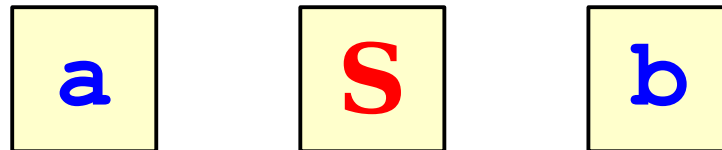
$$S \rightarrow aSb \mid \epsilon$$



# Why the Extra Power?

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

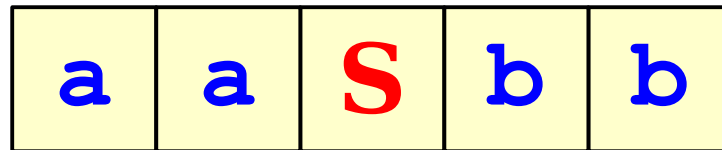
$$S \rightarrow aSb \mid \epsilon$$



# Why the Extra Power?

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

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

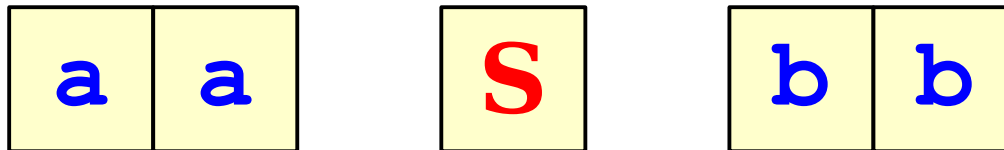




# Why the Extra Power?

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

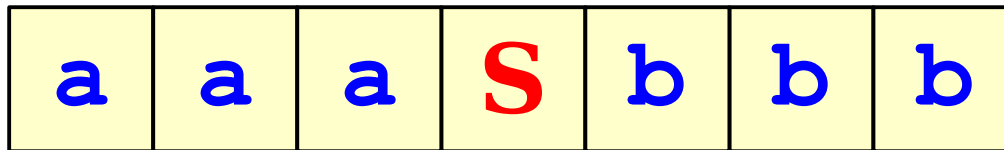
$$S \rightarrow aSb \mid \epsilon$$



# Why the Extra Power?

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

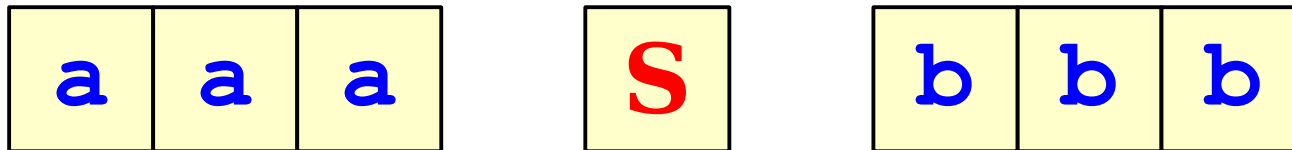
$$S \rightarrow aSb \mid \epsilon$$



# Why the Extra Power?

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

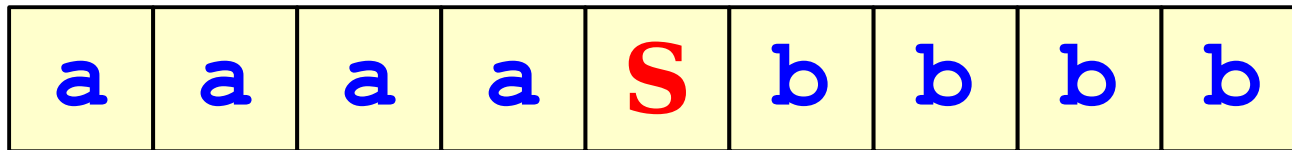
$$S \rightarrow aSb \mid \epsilon$$



# Why the Extra Power?

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

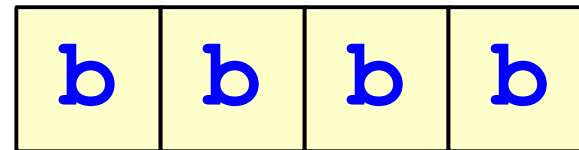
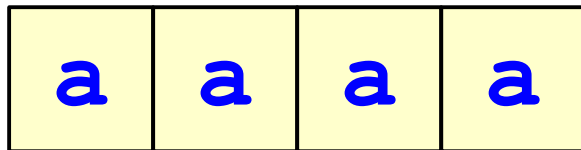
$$S \rightarrow aSb \mid \epsilon$$



# Why the Extra Power?

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

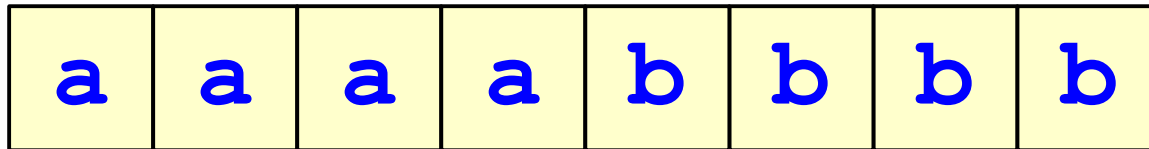
$$S \rightarrow aSb \mid \epsilon$$



# 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!**

# oSTEM@Stanford Spring Mixer

**Out in Science, Technology, Engineering, and Mathematics**

**When:** 4 pm, Friday, May 8<sup>th</sup>

**Where:** Terman Fountain (behind Thornton Center)

**What:** oSTEM forum, meeting and mixer for the queer and allied STEM community.

*Come eat burritos and meet new people!*





Your Questions!

“Did you get my email?”

I'm really behind on my email right now. I'll do my best to reply to everyone as soon as I can. Sorry for the delays, everyone!

“Where on campus was your favorite place to relax?”

There used to be (still is?) a fire escape from the third floor of one of the buildings in the quad down to ground level. You used to be able to (still can?) climb up from ground level and relax up top. It was (still is?) a great view.

“When was a moment when you doubted yourself? How did/do you pick yourself back up?”

“Could you tell us about some of the tougher challenges - personal or otherwise - that you've had to face either generally in your life and how you managed to overcome these? What have those experiences taught you?”

Boy, that's tough.

My advice: learn what your “failure mode” is. Everyone is different. Make sure your friends know what it is so that they can help you. Make sure you're not isolated.

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  $\Sigma = \{a, b\}$  and let  $L = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}$
- We can design a CFG for  $L$  by thinking inductively:
  - Base case:  $\epsilon$ ,  $a$ , and  $b$  are palindromes.
  - If  $w$  is a palindrome, then  $aw$  and  $bw$  are palindromes.

$S \rightarrow \epsilon \mid a \mid b \mid aSa \mid bSb$

# Designing CFGs

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

$((()))$

$(())()$

$((()))((()))$

$((((()))((()))))$

$\epsilon$

$()()$



# Designing CFGs

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

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

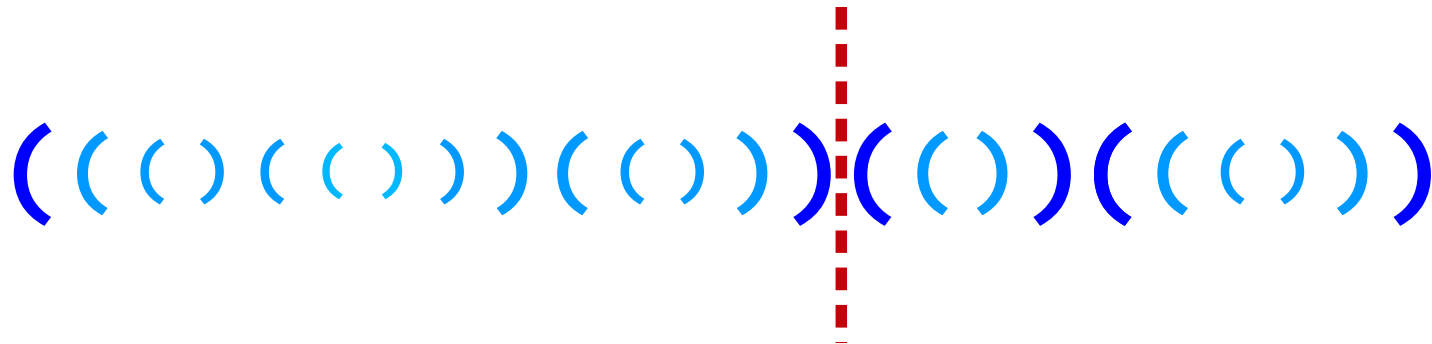
# Designing CFGs

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

$((()((())())((())))((())))$

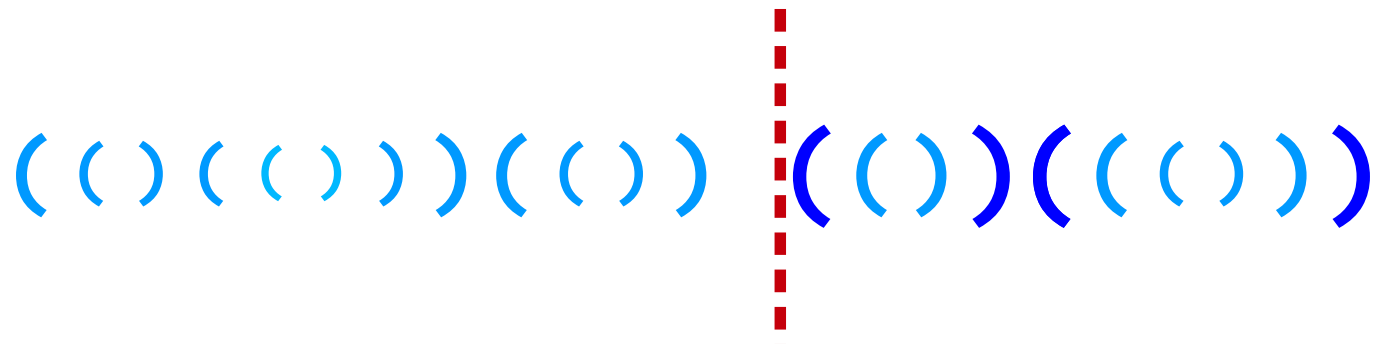
# Designing CFGs

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

  
The diagram shows a string of parentheses:  $( ( ) ( ( ) ) ( ( ) )$  followed by a vertical dashed red line, and then  $( ( ) ) ( ( ( ) ) )$ . The closing parenthesis of the first open parenthesis in the left part is the closing parenthesis immediately to the left of the dashed line.

# Designing CFGs

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

$$S \rightarrow (S)S \mid \epsilon$$

# Designing CFGs: A Caveat

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

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

- Can you derive the string **abba**?

# 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 the next problem set.

# CFG Caveats II

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

$$S \rightarrow aSb$$

- What strings can you derive?
  - Answer: **None!**
- What is the language of the grammar?
  - Answer:  $\emptyset$
- When designing CFGs, make sure your recursion actually terminates!



# CFG Caveats III

- When designing CFGs, remember that each nonterminal can be expanded out independently of the others.
- Let  $\Sigma = \{a, \overset{?}{=}\}$  and let  $L = \{a^n \overset{?}{=} a^n \mid n \in \mathbb{N}\}$ .
- Is the following a CFG for  $L$ ?

$$S \rightarrow X \overset{?}{=} X$$

$$X \rightarrow aX \mid \varepsilon$$

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

# Finding a Build Order

- Let  $\Sigma = \{a, \stackrel{?}{=}\}$  and let  $L = \{a^n \stackrel{?}{=} a^n \mid n \in \mathbb{N}\}$ .
- To build a CFG for  $L$ , we need to be more clever with how we construct the string.
  - If we build the strings of  $a$ 's independently of one another, then we can't enforce that they have the same length.
  - **Idea:** Build both strings of  $a$ 's at the same time.
- Here's one possible grammar based on that idea:

$$S \rightarrow \stackrel{?}{=} \mid aSa$$

$S$

$$\Rightarrow aSa$$

$$\Rightarrow aaSaa$$

$$\Rightarrow aaaSaaa$$

$$\Rightarrow aaa \stackrel{?}{=} aaa$$

# Function Prototypes

- Let  $\Sigma = \{\text{void, int, double, name, (, )}, ,, ;\}$ .
- 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** →  $\epsilon$  | **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.

# Next Time

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