

Context-Free Grammars

Context-Free Grammars

- A ***context-free grammar*** (or ***CFG***) is a new 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:

$E \rightarrow \text{int}$
$E \rightarrow E \text{ Op } E$
$E \rightarrow (E)$
$\text{Op} \rightarrow +$
$\text{Op} \rightarrow -$
$\text{Op} \rightarrow *$
$\text{Op} \rightarrow /$

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

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*),

$E \rightarrow \text{int}$

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

$E \rightarrow (E)$

$\text{Op} \rightarrow +$

$\text{Op} \rightarrow -$

$\text{Op} \rightarrow *$

$\text{Op} \rightarrow /$



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

$$\begin{array}{l} \mathbf{E} \rightarrow \mathit{int} \mid \mathbf{E} \mathbf{Op} \mathbf{E} \mid (\mathbf{E}) \\ \mathbf{Op} \rightarrow + \mid - \mid * \mid / \end{array}$$

- The vertical bar could be read as “or,” a delimiter separating the options for each nonterminal.

Derivations

$E \rightarrow E \text{ Op } E \mid \text{int} \mid (E)$
$\text{Op} \rightarrow + \mid * \mid - \mid /$

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

- 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 $E \Rightarrow^* \text{int} * (\text{int} + \text{int})$.

The Language of a Grammar

- If G is a CFG with alphabet Σ 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 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

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

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

$S \rightarrow Sa \mid dT$

$T \rightarrow bTb \mid c$

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

dc **dca** **cad**
bcb**** **d**T**aa**

Answer at [Pollevo.com/cs103](https://www.pollevo.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 $L = \mathcal{L}(G)$.
- 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 \cup .
- 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 \cup .
- 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 \cup .
- However, we can convert regular expressions to CFGs as follows:

$$S \rightarrow a^*b$$
$$A \rightarrow Aa \mid \epsilon$$

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 \cup .
- However, we can convert regular expressions to CFGs as follows:

$$\begin{aligned} S &\rightarrow a^*b \\ A &\rightarrow Aa \mid \epsilon \end{aligned}$$

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 \cup .
- 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 \cup .
- 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 $A \rightarrow \omega$. They do not have the regular expression operators $*$ or \cup .
- 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 $A \rightarrow \omega$. They do not have the regular expression operators $*$ or \cup .
- 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 $A \rightarrow \omega$. They do not have the regular expression operators $*$ or \cup .
- 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 $A \rightarrow \omega$. They do not have the regular expression operators $*$ or \cup .
- 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 \cup .
- However, we can convert regular expressions to CFGs as follows:

$$\begin{aligned} S &\rightarrow aX \\ X &\rightarrow b \mid c^* \end{aligned}$$

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 \cup .
- However, we can convert regular expressions to CFGs as follows:

$$S \rightarrow aX$$

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

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

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 \cup .
- However, we can convert regular expressions to CFGs as follows:

$$S \rightarrow aX$$

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

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

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 \cup .
- 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 . ■
- ***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 :

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

- 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?

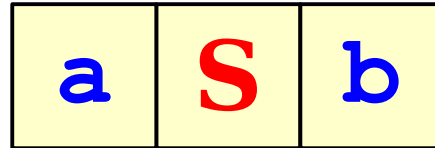
S

The Language of a Grammar

- Consider the following CFG G :

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

- What strings can this generate?

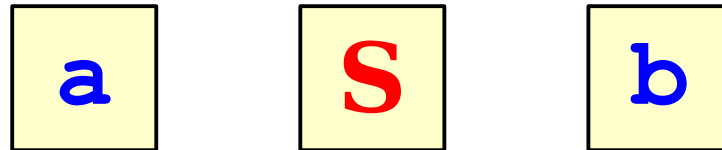


The Language of a Grammar

- Consider the following CFG G :

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

- What strings can this generate?

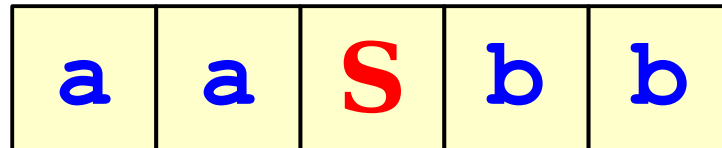


The Language of a Grammar

- Consider the following CFG G :

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

- What strings can this generate?

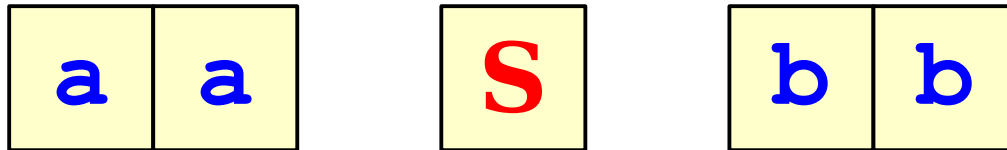


The Language of a Grammar

- Consider the following CFG G :

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

- What strings can this generate?

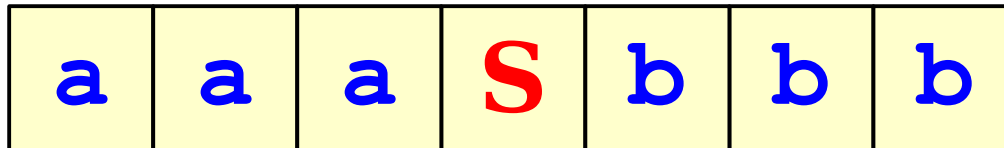


The Language of a Grammar

- Consider the following CFG G :

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

- What strings can this generate?

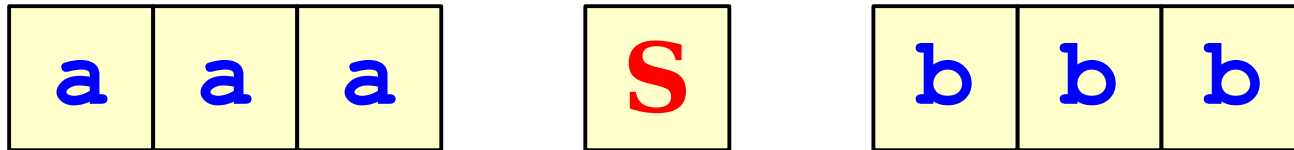


The Language of a Grammar

- Consider the following CFG G :

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

- What strings can this generate?

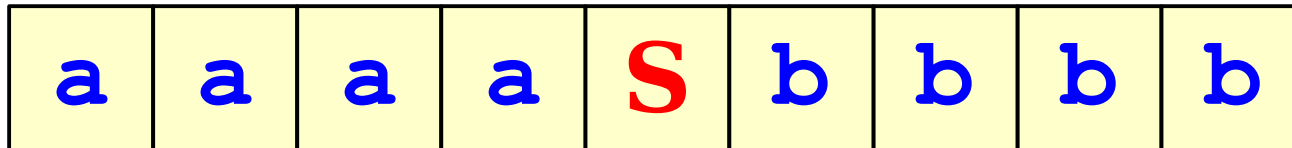


The Language of a Grammar

- 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
---	---	---	---

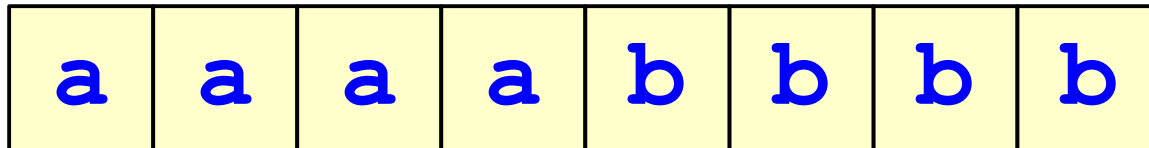
b	b	b	b
---	---	---	---

The Language of a Grammar

- Consider the following CFG G :

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

- What strings can this generate?

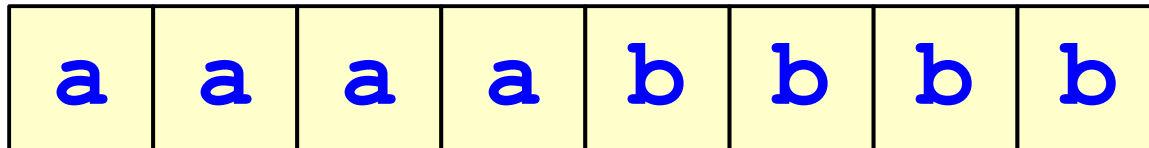


The Language of a Grammar

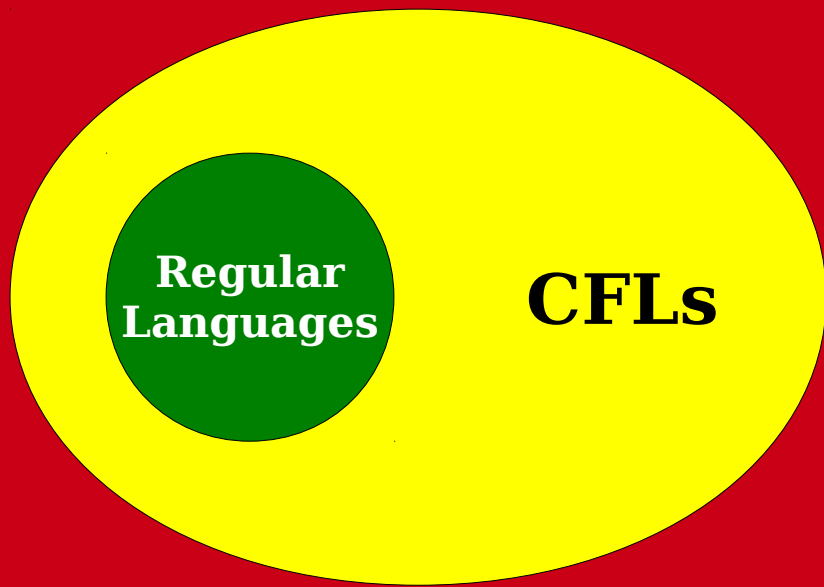
- Consider the following CFG G :

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

- 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?

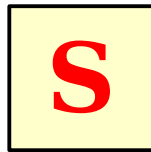
- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded “memory.”

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

Why the Extra Power?

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

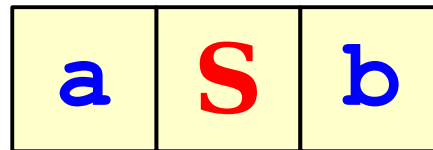
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

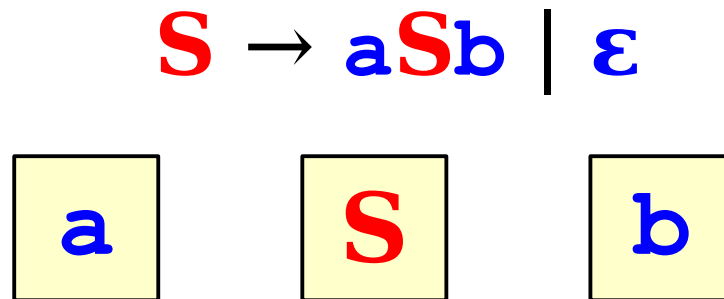
- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded “memory.”

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



Why the Extra Power?

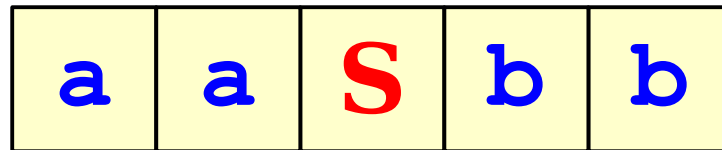
- Why do CFGs have more power than regular expressions?
- ***Intuition:*** Derivations of strings have unbounded “memory.”



Why the Extra Power?

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

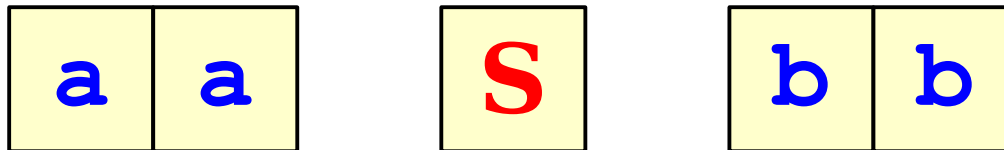
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

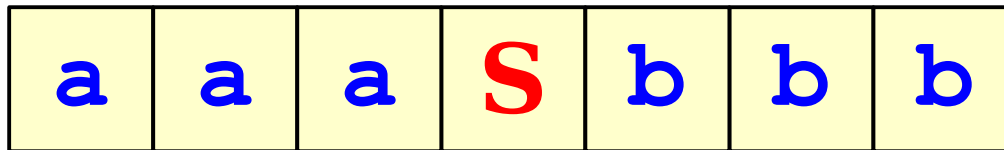
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

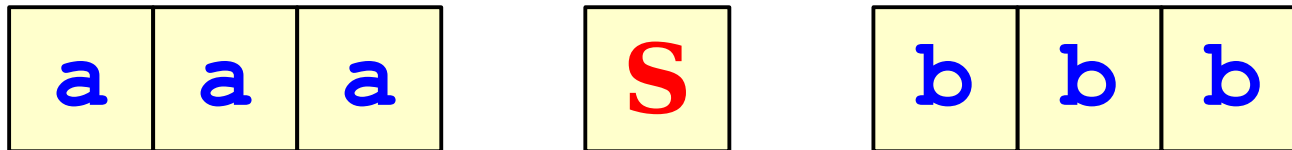
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

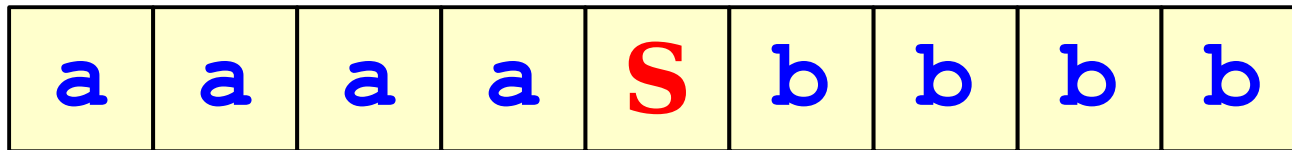
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

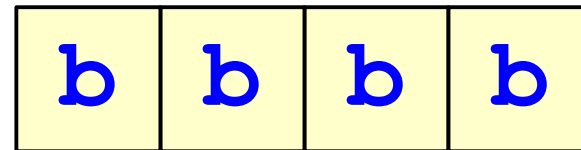
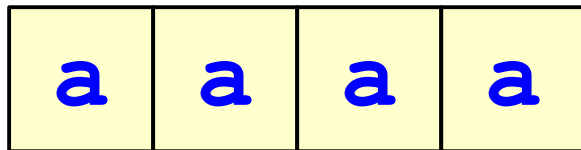
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

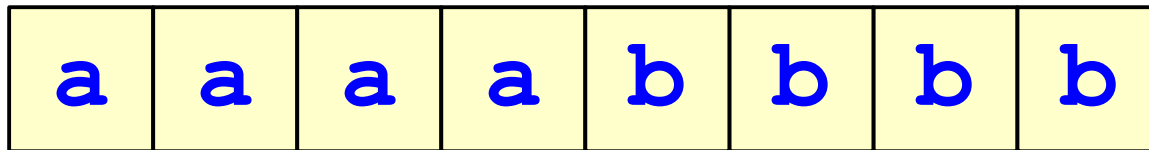
$$S \rightarrow aSb \mid \epsilon$$



Why the Extra Power?

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

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



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 $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ 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.

S \rightarrow ε | **a** | **b** | **aSa** | **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 :

$(())()$

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

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

ϵ

$(())$

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 balanced parentheses: $((() (())) (())) (()) ((()))$. A vertical dashed red line is positioned between the closing parenthesis of the first sub-expression $((() (()))$ and the opening parenthesis of the second sub-expression $(())$, illustrating the recursive step of matching the first opening parenthesis with its corresponding closing 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.

$((()())()) \mid ((())((())))$

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

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

How many of the following CFGs have language L ?

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

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

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

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

Answer at [PollEv.com/cs103](https://www.pollEv.com/cs103) or
text **CS103** to **22333** once to join, then a **number**.

Designing CFGs

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

How many of the following CFGs have language L ?

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

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

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

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

Answer at [PollEv.com/cs103](https://www.pollEv.com/cs103) or
text **CS103** to **22333** once to join, then a **number**.

Designing CFGs

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

How many of the following CFGs have language L ?

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

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

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

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

Answer at [PollEv.com/cs103](https://www.pollEv.com/cs103) or
text **CS103** to **22333** once to join, then a **number**.

Designing CFGs

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

How many of the following CFGs have language L ?

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

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

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

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

Answer at [PollEv.com/cs103](https://www.pollEv.com/cs103) or
text **CS103** to **22333** once to join, then a **number**.

Designing CFGs

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

How many of the following CFGs have language L ?

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

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

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

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

Answer at [PollEv.com/cs103](https://www.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 $\{ \mathbf{a}, \mathbf{b} \}^*$ 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 $\Sigma = \{a, \varepsilon\}$ and let $L = \{a^n \varepsilon a^n \mid n \in \mathbb{N}\}$.
- Is the following a CFG for L ?

$$S \rightarrow X \varepsilon X$$

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

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

Finding a Build Order

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

$$\mathbf{S} \rightarrow \underline{?} \mid \mathbf{aSa}$$

\mathbf{S}

$\Rightarrow \mathbf{aS a}$

$\Rightarrow \mathbf{aaS a a}$

$\Rightarrow \mathbf{aaaS a a a}$

$\Rightarrow \mathbf{aaa\underline{?}aaa}$

Function Prototypes

- Let $\Sigma = \{\mathbf{void}, \mathbf{int}, \mathbf{double}, \mathbf{name}, (,), ,, ;\}$.
- 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

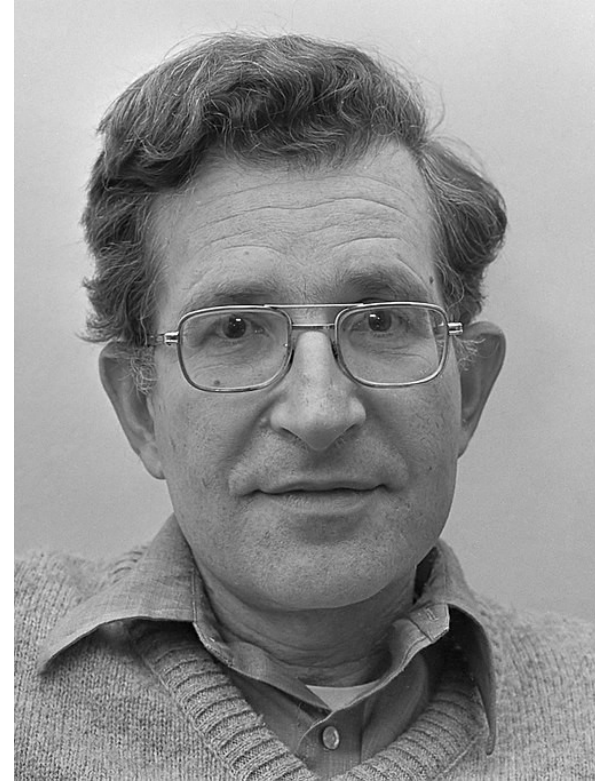
- One of the key steps in a compiler is figuring out what a program “means.”
- This is usually done by defining a grammar showing the high-level structure of a programming language.
- There are certain classes of grammars (LL(1) grammars, LR(1) grammars, LALR(1) grammars, etc.) for which it's easy to 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

- Invented CFGs!
- Helped found entire fields of linguistics and cognitive science
- Today, perhaps more broadly known for politics than linguistics
 - Anti-capitalism, anti-imperialism, anti-war
 - Drawing on linguistics expertise, wrote extensively on state propaganda (*Manufacturing Consent*)



PC: Hans Peters / Anefo (via Wikimedia)

Next Time

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