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:

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

$$
\begin{array}{ll}
 & \mathbf{E} \\
\Rightarrow & \mathbf{E\ Op\ E} \\
\Rightarrow & \mathbf{E\ Op}\ \texttt{(}\mathbf{E}\texttt{)} \\
\Rightarrow & \mathbf{E\ Op}\ \texttt{(}\mathbf{E\ Op\ E}\texttt{)} \\
\Rightarrow & \mathbf{E}\ \texttt{*}\ \texttt{(}\mathbf{E\ Op\ E}\texttt{)} \\
\Rightarrow & \texttt{int * (}\mathbf{E\ Op\ E}\texttt{)} \\
\Rightarrow & \texttt{int * (int }\mathbf{Op\ E}\texttt{)} \\
\Rightarrow & \texttt{int * (int }\mathbf{Op}\texttt{ int)} \\
\Rightarrow & \texttt{int * (int + int)}
\end{array}
$$

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 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.
  - 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. 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

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

A Notational Shorthand

$$\boxed{\begin{array}{l} \mathbf{E} \to \texttt{int} \mid \mathbf{E}\ \mathbf{Op}\ \mathbf{E} \mid \texttt{(}\mathbf{E}\texttt{)} \\ \mathbf{Op} \to \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \end{array}}$$

Derivations

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

$$\begin{aligned}
& \mathbf{E} \\
\Rightarrow\ & \mathbf{E\ Op\ E} \\
\Rightarrow\ & \mathbf{E\ Op}\ \texttt{(}\mathbf{E}\texttt{)} \\
\Rightarrow\ & \mathbf{E\ Op}\ \texttt{(}\mathbf{E\ Op\ E}\texttt{)} \\
\Rightarrow\ & \mathbf{E}\ \texttt{*}\ \texttt{(}\mathbf{E\ Op\ E}\texttt{)} \\
\Rightarrow\ & \texttt{int * (}\mathbf{E\ Op\ E}\texttt{)} \\
\Rightarrow\ & \texttt{int * (int } \mathbf{Op\ E}\texttt{)} \\
\Rightarrow\ & \texttt{int * (int } \mathbf{Op}\ \texttt{int)} \\
\Rightarrow\ & \texttt{int * (int + int)}
\end{aligned}$$

- A sequence of steps where nonterminals are replaced by the right-hand side of a production is called a ***derivation***.
- If string $\boldsymbol{\alpha}$ derives string $\boldsymbol{\omega}$, we write $\boldsymbol{\alpha} \Rightarrow^* \boldsymbol{\omega}$.
- In the example on the left, we see $\mathbf{E} \Rightarrow^*$ `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 derivable from the start symbol.
- Note: $\boldsymbol{\omega}$ must be in Σ*, 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 $L = \mathscr{L}(G)$.
- 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** → ***ω***. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → **a*b**

A → Aa | 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`
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:

$$S \rightarrow \texttt{a(b ∪ c*)}$$

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:

$$\mathbf{S} \rightarrow \mathtt{a(b\ \cup\ c^*)}$$
$$\mathbf{X} \rightarrow \mathtt{b}\ |\ \mathtt{c^*}$$

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 → **`aX`**

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** → ***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

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:

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

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:

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

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*:

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

- What strings can this generate?

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

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

a	S	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?

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

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	S	b	b

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

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

The Language of a Grammar

- Consider the following CFG *G*:

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

- What strings can this generate?

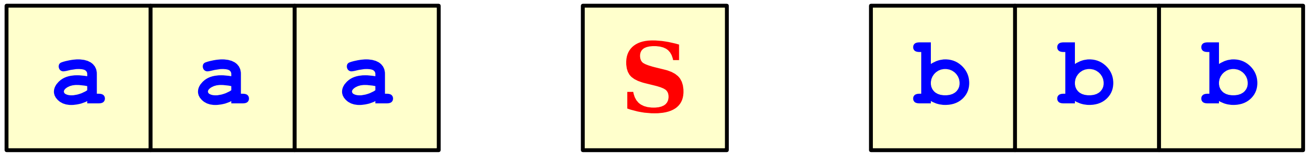
a	a	a	S	b	b	b

The Language of a Grammar

- Consider the following CFG G:

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

- 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?

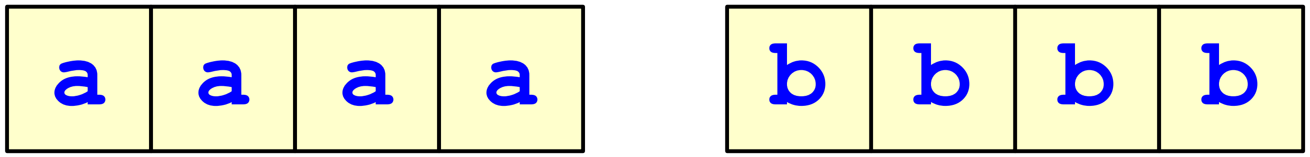
a	**a**	**a**	**a**	**S**	**b**	**b**	**b**	**b**

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?



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

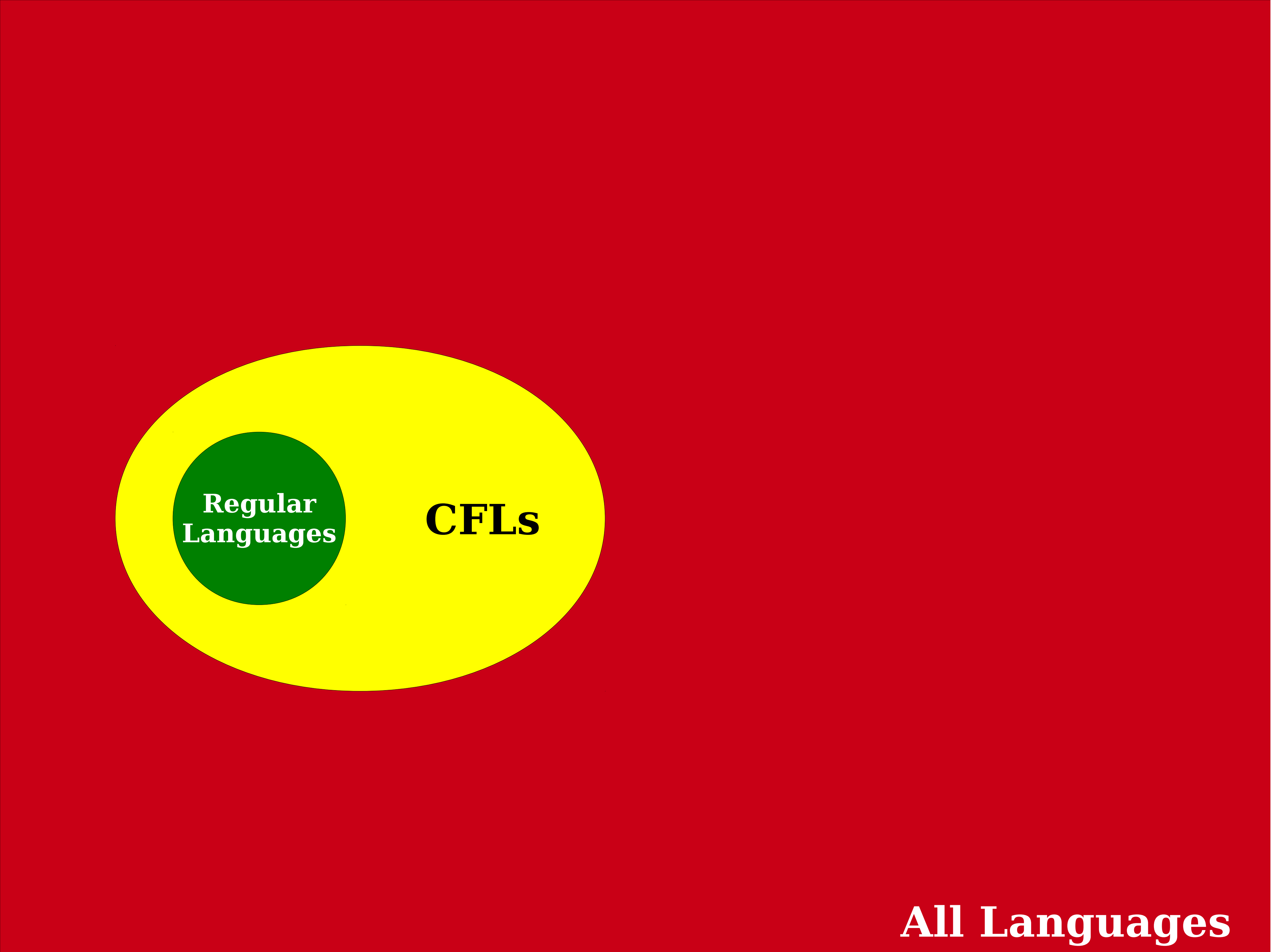
- Consider the following CFG G:

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

- What strings can this generate?

a	a	a	a	b	b	b	b

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


Regular Languages
CFLs
All Languages

Why the Extra Power?

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

Why the Extra Power?

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

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

Why the Extra Power?

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

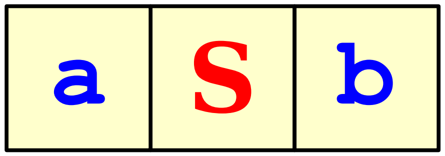
S → **aSb** | **ε**

$\boxed{\mathbf{S}}$

Why the Extra Power?

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

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



Why the Extra Power?

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

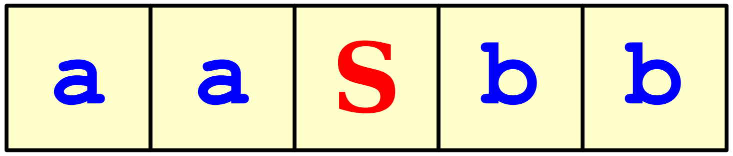
$$\mathbf{S} \rightarrow \texttt{a}\mathbf{S}\texttt{b} \mid \varepsilon$$

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

Why the Extra Power?

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

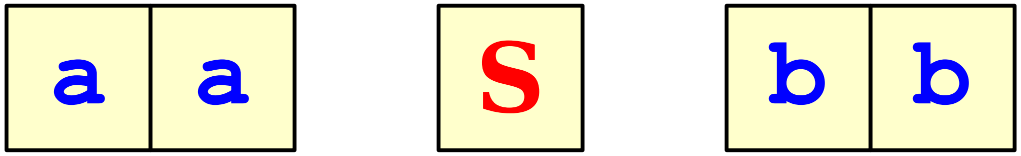
$$S \rightarrow aSb \mid \varepsilon$$



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."

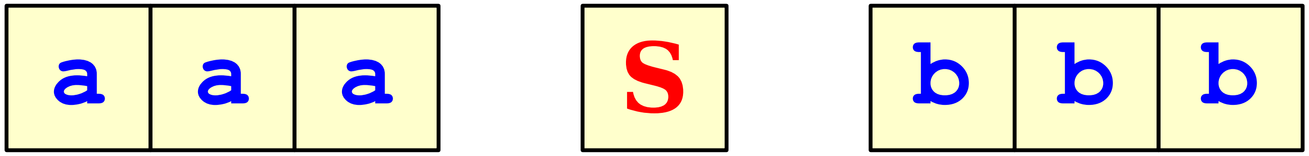
S → aSb | ε

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."

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



Why the Extra Power?

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

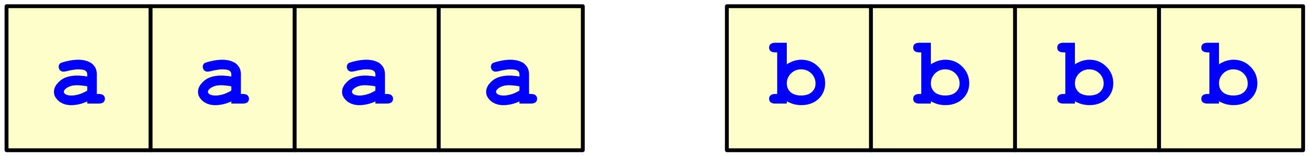
$$S \rightarrow aSb \mid \varepsilon$$

a	a	a	a	S	b	b	b	b

Why the Extra Power?

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

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



Why the Extra Power?

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

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

a	a	a	a	b	b	b	b

Time-Out for Announcements!

Problem Set Six

- Problem Set Six is due this Friday.
  - Be careful about using late days; the midterm is next Tuesday.
- As always, hit us up if you have any questions!

Midterm Exam Logistics

- The second midterm exam is next ***Tuesday, May 23rd***, from ***7:00PM – 10:00PM***. Locations are divvied up by last (family) name:
  - `Abb – Pag`: Go to Hewlett 200.
  - `Par – Tak`: Go to Sapp 114.
  - `Tan – Val`: Go to Hewlett 101.
  - `Var – Yim`: Go to Hewlett 102.
  - `You – Zuc`: Go to Hewlett 103.
- You're responsible for Lectures 00 – 13 and topics covered in PS1 – PS5. Later lectures and problem sets won't be tested. The focus is on PS3 – PS5 and Lectures 06 – 13.
- The exam is closed-book, closed-computer, and limited-note. You can bring a double-sided, 8.5" × 11" sheet of notes with you to the exam, decorated however you'd like.

Preparing for the Exam

- Up on the course website, you’ll find
  - four sets of extra practice problems (EPP4 – EPP7), with solutions; and
  - three practice midterm exams, with solutions.
- As always, feel free to stop by office hours or to visit Piazza if you have questions. We’re happy to help out!

Your Questions

“What is the best advice you have ever gotten EVER EVER EVER in your life? No pressure... :D”

I thought about this a bit and can't come to a definitive "best advice EVER EVER EVER," but here are three very good pieces of advice I've received:

1. "You only have a limited number of psychobucks to spend worrying or upset about things. Be strategic with where you spend them."

2. "If you can't make your opponent's point for them, you don't truly understand the issue."

3. "All models are wrong; some are useful."

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 $L = \{ w \in \Sigma^* \mid w$ 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 | aSa | bSb

Designing CFGs

- Let Σ = { **(**, **)** } and let L = {$w \in \Sigma^*$ | 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.

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

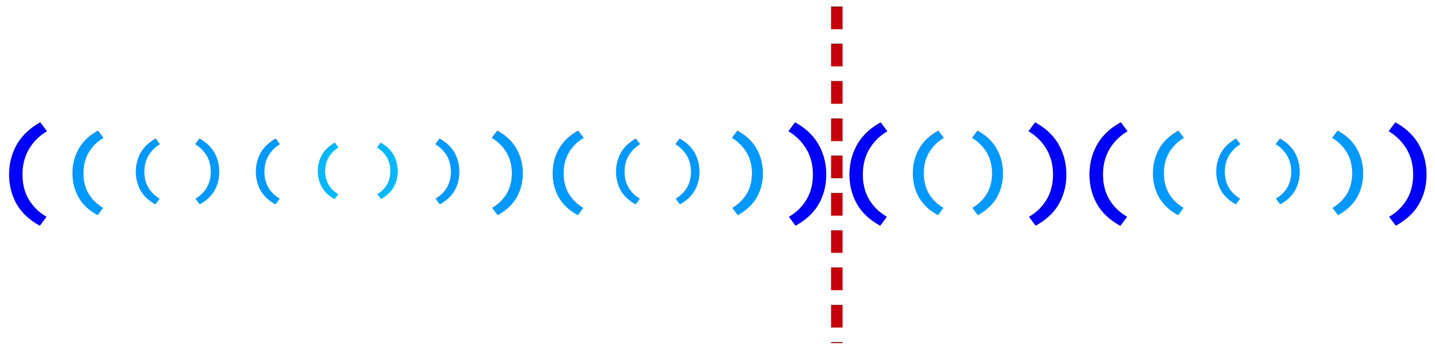
Designing CFGs

- Let Σ = { (,) } and let L = {w ∈ Σ* | 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.

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

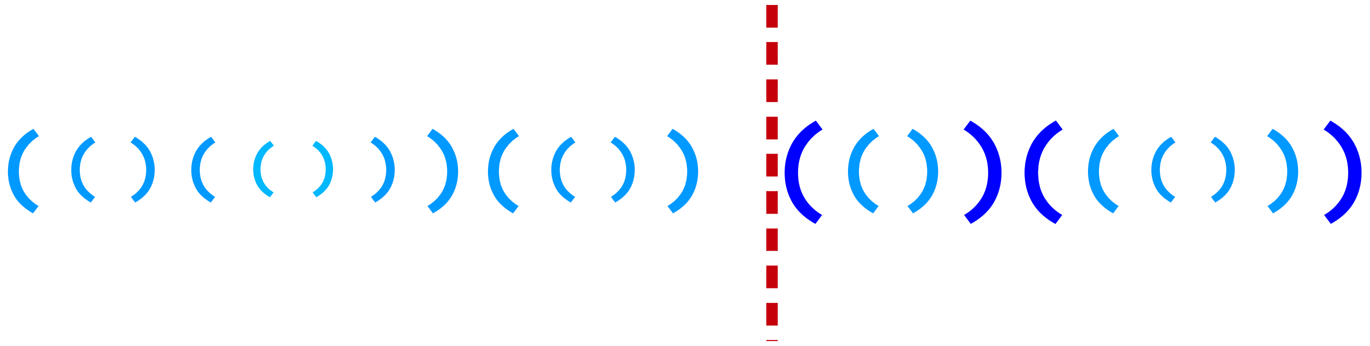
Designing CFGs

- 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 Σ = { (,) } and let L = {w ∈ Σ* | 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 Σ = { **(**, **)** } and let L = { w ∈ Σ* | 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.

S → (S) S | ε

Designing CFGs: A Caveat

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

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

- 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 Problem Set 8.

CFG Caveats II

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

$$\mathbf{S} \to \mathtt{a}\mathbf{S}\mathtt{b}$$

- What strings in {**a**, **b**}* can you derive?
  - Answer: ***None!***
- What is the language of the grammar?
  - Answer: **Ø**
- When designing CFGs, make sure your recursion actually terminates!

CFG Caveats III

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

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

$$\mathbf{X} \rightarrow \texttt{a}\mathbf{X} \mid \varepsilon$$

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

Finding a Build Order

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

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

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

Function Prototypes

- Let Σ = {**`void`**, **`int`**, **`double`**, **`name`**, **`(`**, **`)`**, **`,`**, **`;`**}.
- Let's write a CFG for C-style function prototypes!
- Examples:
  - **`void name(int name, double name);`**
  - **`int name();`**
  - **`int name(double name);`**
  - **`int name(int, int name, int);`**
  - **`void name(void);`**

Function Prototypes

- Here's one possible grammar:
  - **S** → **Ret** `name` `(`**Args**`)` `;`
  - **Ret** → **Type** | `void`
  - **Type** → `int` | `double`
  - **Args** → ε | `void` | **ArgList**
  - **ArgList** → **OneArg** | **ArgList**, **OneArg**
  - **OneArg** → **Type** | **Type** `name`
- Fun question to think about: what changes would you need to make to support pointer types?

Summary of CFG Design Tips

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

Applications of Context-Free Grammars

CFGs for Programming Languages

```
BLOCK   →  STMT
         |  { STMTS }

STMTS   →  ε
         |  STMT STMTS

STMT    →  EXPR;
         |  if (EXPR) BLOCK
         |  while (EXPR) BLOCK
         |  do BLOCK while (EXPR);
         |  BLOCK
         |  …

EXPR    →  identifier
         |  constant
         |  EXPR + EXPR
         |  EXPR – EXPR
         |  EXPR * EXPR
         |  …
```

Grammars in Compilers

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

Natural Language Processing

- By building context-free grammars for actual languages and applying statistical inference, it's possible for a computer to recover the likely meaning of a sentence.
  - In fact, CFGs were first called ***phrase-structure grammars*** and were introduced by Noam Chomsky in his seminal work *Syntactic Structures*.
  - They were then adapted for use in the context of programming languages, where they were called ***Backus-Naur forms***.
- Stanford's CoreNLP 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 do you program them?