# 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:

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

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

# 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** → `/`

$$
\begin{aligned}
& \mathbf{E} \\
\Rightarrow\ & \mathbf{E\ Op\ E} \\
\Rightarrow\ & \mathbf{E\ Op}\ \texttt{int} \\
\Rightarrow\ & \texttt{int}\ \mathbf{Op}\ \texttt{int} \\
\Rightarrow\ & \texttt{int / int}
\end{aligned}
$$

# 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

- 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

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

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

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

# The Language of a Grammar

- If $G$ is a CFG with alphabet $\Sigma$ 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 $\Sigma^*$, 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 $L = \mathscr{L}(G)$.
- 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 $\mathbf{A} \to \boldsymbol{\omega}$. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

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

# From Regexes to CFGs

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

$$\mathbf{S} \to \mathbf{A}\mathtt{b}$$
$$\mathbf{A} \to \mathbf{A}\mathtt{a} \mid \varepsilon$$

# 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:

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

# 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`**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:

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

# 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:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}$$
$$\mathbf{X} \rightarrow \mathtt{b} \mid \mathtt{c}^*$$

# 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:

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}$$
$$\mathbf{X} \rightarrow \mathtt{b} \mid \mathbf{C}$$

# From Regexes to CFGs

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

$$\mathbf{S} \to \mathtt{a}\mathbf{X}$$
$$\mathbf{X} \to \mathtt{b} \mid \mathbf{C}$$
$$\mathbf{C} \to \mathbf{C}\mathtt{c} \mid \varepsilon$$

# 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 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?

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

# 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 | 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?

| | | |
|---|---|---|
| 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?

| | | | | |
|---|---|---|---|---|
| **a** | **a** | **S** | **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 | S | 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 | 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?

| | | | | | | |
|---|---|---|---|---|---|---|
| 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?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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?

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 $G$:

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

- 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."

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

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

# Why the Extra Power?

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

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

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

# 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 | S | b |

# 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** | **S** | **b** | **b** |

# 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 | S | b | b |

# Why the Extra Power?

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

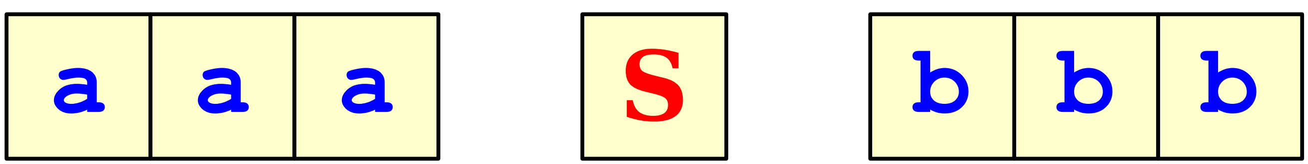
$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{S}\mathtt{b} \mid \varepsilon$$

| | | | | | | |
|---|---|---|---|---|---|---|
| a | a | a | **S** | b | b | b |

# 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."

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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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** → **aSb** | **ε**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | a | a | | b | b | b | b |

# Why the Extra Power?

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

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | a | a | a | b | b | b | b |

Time-Out for Announcements!

# Problem Set Seven

- Problem Set Six was due at the start of today's lecture.
  - Want to use late days? Submit up to Monday at 3:00PM.
- Problem Set Seven goes out now. It's due next Friday.
  - Play around with the Myhill-Nerode theorem and the limits of regular languages!
  - Play around with your very own CFGs!

# Midterms Graded

- Midterms have been graded. They're available for pickup in the Gates building.
  - SCPD students: we've sent the exams back to the SCPD office. You should hear back from them soon.
- Solutions and stats are available in the Gates building in the normal handout filing cabinet.

# Midterm Regrades

- If you believe that we made a grading error on the exam, you can submit it for a regrade. To do so, fill out the form online, staple it to your exam, and hand it to Keith by next Friday.
- Please only submit regrades if you
  - believe that we actually graded your exam incorrectly, and
  - you've talked about the exam with the course staff and they agree with you.
- ***Your score can go down if you ask for a regrade***. Please be sure you really want to ask for it before you submit a regrade request.

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 $\boldsymbol{\omega}$ is a palindrome, then **`a`**$\boldsymbol{\omega}$**`a`** and **`b`**$\boldsymbol{\omega}$**`b`** are palindromes.

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

# 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 Σ = { (, ) } 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 $L = \{ w \in \Sigma^* \mid w$ 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: A Caveat

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

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

$$\mathbf{S} \to \mathbf{aSb}$$

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

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

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

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

# Finding a Build Order

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

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

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

# 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

$$
\begin{array}{lcl}
\mathbf{BLOCK} & \to & \mathbf{STMT} \\
 & \mid & \texttt{\{}\ \mathbf{STMTS}\ \texttt{\}} \\
 & & \\
\mathbf{STMTS} & \to & \varepsilon \\
 & \mid & \mathbf{STMT}\ \mathbf{STMTS} \\
 & & \\
\mathbf{STMT} & \to & \mathbf{EXPR}\texttt{;} \\
 & \mid & \texttt{if (}\mathbf{EXPR}\texttt{)}\ \mathbf{BLOCK} \\
 & \mid & \texttt{while (}\mathbf{EXPR}\texttt{)}\ \mathbf{BLOCK} \\
 & \mid & \texttt{do}\ \mathbf{BLOCK}\ \texttt{while (}\mathbf{EXPR}\texttt{);} \\
 & \mid & \mathbf{BLOCK} \\
 & \mid & \ldots \\
 & & \\
\mathbf{EXPR} & \to & \texttt{identifier} \\
 & \mid & \texttt{constant} \\
 & \mid & \mathbf{EXPR}\ \texttt{+}\ \mathbf{EXPR} \\
 & \mid & \mathbf{EXPR}\ \texttt{-}\ \mathbf{EXPR} \\
 & \mid & \mathbf{EXPR}\ \texttt{*}\ \mathbf{EXPR} \\
 & \mid & \ldots
\end{array}
$$

# 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?