Context-Free Grammars

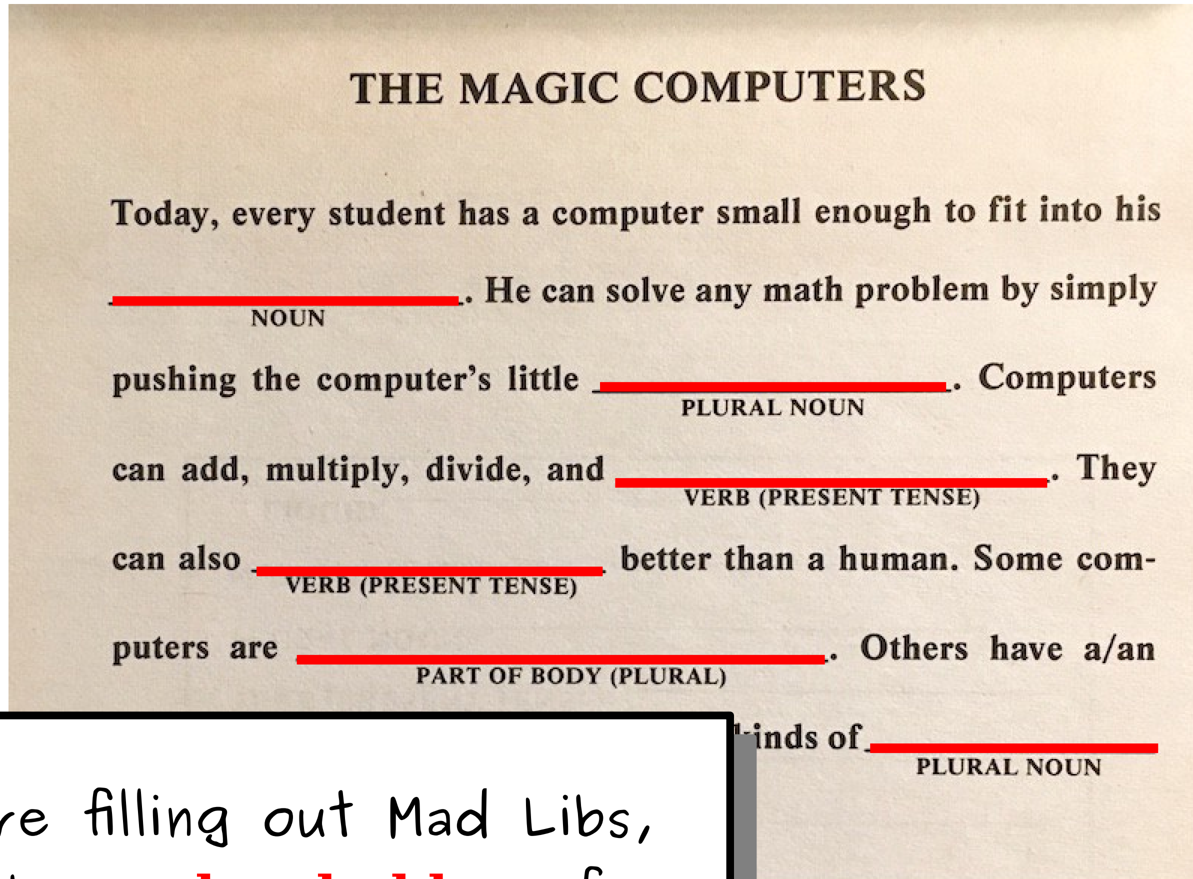
A Motivating Question

Terminal — python — 66×21

```
>>> (26 + 42) * 2 + 1
```

How does my computer know what this sequence of characters means? How can it determine whether or not this expression is even syntactically valid?

An Analogy: Mad Libs



THE MAGIC COMPUTERS

Today, every student has a computer small enough to fit into his ____________ NOUN. He can solve any math problem by simply pushing the computer's little ____________ PLURAL NOUN. Computers can add, multiply, divide, and ____________ VERB (PRESENT TENSE). They can also ____________ VERB (PRESENT TENSE) better than a human. Some computers are ____________ PART OF BODY (PLURAL). Others have a/an ... kinds of ____________ PLURAL NOUN

When you're filling out Mad Libs, you have these **placeholders** for different parts of speech.

Mad Libs for Arithmetic Expressions

Imagine I have a template like this:

(______ ______ ______) ______ ______ ______ ______

Int Op Int Op Int Op Int

Mad Libs for Arithmetic Expressions

Here's one way I could fill it out:

(	26	+	42	)	*	2	+	1
	Int	Op	Int		Op	Int	Op	Int

Mad Libs for Arithmetic Expressions

Here's another:

(	7	*	5	)	/	5	-	49
	Int	Op	Int		Op	Int	Op	Int

Imagine you have a computer that's pre-programmed with this template.

You could then enter a string and be able to check whether it is valid. You can also understand what individual pieces of the string mean based on which part of the template they're filling in.

Mad Libs for Arithmetic Expressions

This is nice but I can only make expressions of the form **(Int Op Int) Op Int Op Int**

(______ ______ ______) ______ ______ ______ ______

Int Op Int Op Int Op Int

But there are many valid arithmetic expressions that don't follow this pattern!

Mad Libs for Arithmetic Expressions

Idea: could we come up with a set of rules for generating valid arithmetic Mad Libs templates?

Eg. **Int Op Int**, **(Int Op (Int Op Int))**, **(Int Op Int) Op (Int Op Int)** ...

Describing Languages

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

Context-Free Grammars

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

Arithmetic Expressions

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

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

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

Arithmetic Expressions

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

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

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

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. By convention, the start symbol is the one on the left-hand side of the first production.

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

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

A Notational Shorthand

$$\mathbf{E} \rightarrow \texttt{int} \mid \mathbf{E}\ \mathbf{Op}\ \mathbf{E} \mid \texttt{(}\mathbf{E}\texttt{)}$$

$$\mathbf{Op} \rightarrow \texttt{+} \mid \texttt{-} \mid \times \mid \texttt{/}$$

Derivations

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

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

- 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

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

- 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

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

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

$$\mathbf{S} \to \mathbf{Sa} \mid \mathbf{dT}$$
$$\mathbf{T} \to \mathbf{bTb} \mid \mathbf{c}$$

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

dca
cad
bcb
dTaa

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

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

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

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:

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:

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

From Regexes to CFGs

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

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

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:

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

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:

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

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

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:

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

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

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:

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

From Regexes to CFGs

- CFGs consist purely of production rules of the form $\mathbf{A} \rightarrow \boldsymbol{\omega}$. 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:

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

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:

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

From Regexes to CFGs

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

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

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. ■
- ***Great 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:

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

- What strings can this generate?

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

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:

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

- 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:

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

- What strings can this generate?

a	a	a	a	b	b	b	b

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


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."

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

a	**S**	**b**

Why the Extra Power?

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

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

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

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

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	**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."

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

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."

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

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."

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

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."

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

a	a	a	a	b	b	b	b

Let’s take a five minute break!

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 $\Sigma = \{\texttt{a}, \texttt{b}\}$ 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 $\texttt{a}\omega\texttt{a}$ and $\texttt{b}\omega\texttt{b}$ are palindromes.
  - No other strings are palindromes.

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

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:

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

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:

a	**b**	**b**	**a**

S → **ε** | **a** | **b** | **aSa** | **bSb**

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:

b	**a**	**b**	**b**	**a**	**b**

S → **ε** | **a** | **b** | **aSa** | **bSb**

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:

a	b	a	b	b	a	b	a

S → **ε** | **a** | **b** | **aSa** | **bSb**

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:

Inductive (building up) perspective: you can take any palindrome and build a larger one by adding the same character to both ends.

S → **ε** | **a** | **b** | **aSa** | **bSb**

Designing CFGs

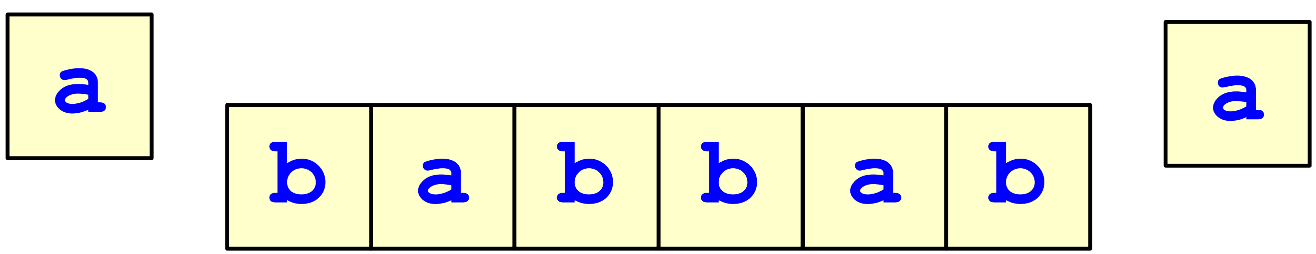
- Let $\Sigma = \{\texttt{a}, \texttt{b}\}$ and let $L = \{w \in \Sigma^* \mid w \text{ is a palindrome }\}$
- We can design a CFG for L by thinking recursively:

a	b	a	b	b	a	b	a

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

Designing CFGs

- Let Σ = {**a**, **b**} and let L = {$w \in \Sigma^*$ | w is a palindrome }
- We can design a CFG for L by thinking recursively:



S → **ε** | **a** | **b** | **aSa** | **bSb**

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 recursively:

b	a	b	b	a	b

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

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 recursively:

b | **a** | **b** | **b** | **a** | **b**

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

Designing CFGs

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{w \in \Sigma^* \mid w$ is a palindrome $\}$
- We can design a CFG for L by thinking recursively:

a	b	b	a

S → **ε** | **a** | **b** | **aSa** | **bSb**

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 recursively:

Recursive (building down) perspective: you can take any palindrome and repeatedly remove the same character from both ends, leaving behind a palindrome.

S → ε | a | b | aSa | bSb

Designing CFGs

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

{{{}}}

{{}}{}

{{}{}}{{}{}}

{{{{{}}}{{}}}}

ε

{}{}

Designing CFGs

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

`{{{}{{}}}{{}}}{{}}{{{}}}`

Designing CFGs

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

{{{}{{}}}{{}}}{{}}{{{}}}

Designing CFGs

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

{{{}{{}}}{{}}}{{}}{{{}}}

Designing CFGs

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

{ { } { { } } } { { } } { { } } { { { } } }

Designing CFGs

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

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

Designing CFGs

- Here's the derivation from class today:

  $\mathbf{S}$

  $\Rightarrow \{\mathbf{S}\}\mathbf{S}$

  $\Rightarrow \{\{\mathbf{S}\}\mathbf{S}\}\mathbf{S}$

  $\Rightarrow \{\{\{\mathbf{S}\}\mathbf{S}\}\mathbf{S}\}\mathbf{S}$

  $\Rightarrow \{\{\{\mathbf{S}\}\{\mathbf{S}\}\mathbf{S}\}\mathbf{S}\}\mathbf{S}$

  $\Rightarrow \{\{\{\varepsilon\}\{\mathbf{S}\}\mathbf{S}\}\mathbf{S}\}\mathbf{S}$

  $\Rightarrow \{\{\{\varepsilon\}\{\varepsilon\}\mathbf{S}\}\mathbf{S}\}\mathbf{S}$

  $\Rightarrow \{\{\{\varepsilon\}\{\varepsilon\}\varepsilon\}\mathbf{S}\}\mathbf{S}$

  $\Rightarrow \{\{\{\varepsilon\}\{\varepsilon\}\varepsilon\}\varepsilon\}\mathbf{S}$

  $\Rightarrow \{\{\{\varepsilon\}\{\varepsilon\}\varepsilon\}\varepsilon\}\varepsilon$

Designing CFGs

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

How many of the following CFGs have language L?

S → aSb | bSa | ε

S → abS | baS | ε

S → abSba | baSab | ε

S → SbaS | SabS | ε

Designing CFGs

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

How many of the following CFGs have language L?

S → aSb | bSa | ε	**S → abS | baS | ε**
S → abSba | baSab | ε	**S → SbaS | SabS | ε**

Designing CFGs

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

How many of the following CFGs have language L?

S → **aSb** | **bSa** | **ε**	**S** → **abS** | **baS** | **ε**
S → **abSba** | **baSab** | **ε**	**S** → **SbaS** | **SabS** | **ε**

Designing CFGs

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

How many of the following CFGs have language L?

S → **aSb** | **bSa** | **ε**

S → **abS** | **baS** | **ε**

S → **abSba** | **baSab** | **ε**

S → **SbaS** | **SabS** | **ε**

Designing CFGs

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

How many of the following CFGs have language L?

S → aSb | bSa | ε

S → abS | baS | ε

S → abSba | baSab | ε

S → SbaS | SabS | ε

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

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

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

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

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

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

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:

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

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

Storing Information in Nonterminals

- ***Key idea:*** Different non-terminals should represent different states or different types of strings.
  - For example, different phases of the build, or different possible structures for the string.
  - Think like the same ideas from DFA/NFA design where states in your automata represent pieces of information.

Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- Examples:

ε $\in L$	**a** $\notin L$
abb $\in L$	**b** $\notin L$
bab $\in L$	**ababab** $\notin L$
aababa $\in L$	**aabaaaaaa** $\notin L$
bbbbbb $\in L$	**bbbb** $\notin L$

Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- Examples:

ε $\in L$	**a** $\notin L$
a bb $\in L$	**b** $\notin L$
b ab $\in L$	**ab abab** $\notin L$
aa baba $\in L$	**aab aaaaaa** $\notin L$
bb bbbb $\in L$	**bbbb** $\notin L$

Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- One approach:

aaa	**bab**
abb	**bbb**
aaabab	**bbabbb**
aababa	**bbbaaaaaa**
aaaaaaaaa	**bbbbbabaa**

Observation 1:

Strings in this language are either: the first third is **a**s or the first third is **b**s.

Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- One approach:

aaa	**bab**
abb	**bbb**
aaabab	**bbabbb**
aababa	**bbbaaaaaa**
aaaaaaaaa	**bbbbbabaa**

Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- One approach:

aaa	bab
abb	bbb
aaabab	bbabbb
aababa	bbbaaaaaa
aaaaaaaaa	bbbbbabaa

Observation 2:

Amongst these strings, for every **a** I have in the first third, I need two other characters in the last two thirds.

Storing Information in Nonterminals

- Let $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- One approach:

aaa	bab
abb	bbb
aaabab	bbabbb
	aaaaa
	babaa

Observation 2:

Amongst these strings, for every **a** I have in the first third, I need two other characters in the last two thirds.

This pattern of "for every x I see here, I need a y somewhere else in the string" is very common in CFGs!

Storing Information in Nonterminals

- Let $\Sigma = \{$a, b$\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- One approach:

aaa	bab	***Observation 2:***
abb	bbb	Amongst these strings, for every a I have in the first third, I need two other characters in the last two thirds.
aaabab	bbabbb	
aababa	bbbaaaaaa	
aaaaaaaaa	bbbbbabaa	

$\mathbf{A} \rightarrow \texttt{a}\mathbf{AXX} \mid \varepsilon \qquad \mathbf{X} \rightarrow \texttt{a} \mid \texttt{b}$

Storing Information in Nonterminals

- Let Σ = {**a**, **b**} and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- One approach:

aaa

abb

aaabab

aababa

aaaaaaaaa

bab

bbbbbabaa

Here the nonterminal **A** represents "a string where the first third is **a**'s" and the nonterminal **X** represents "any character"

$A \rightarrow$ **a**AXX | ε $\qquad$ $X \rightarrow$ **a** | **b**

Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- One approach:

aaa	bab
abb	bbb
aaabab	bbabbb
aababa	bbbaaaaaa
aaaaaaaaa	bbbbbabaa

$A \rightarrow$ **a**$AXX \mid \varepsilon$ $\quad X \rightarrow$ **a** $\mid$ **b**

Storing Information in Nonterminals

- Let $\Sigma = \{$a, b$\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- One approach:

aaa	bab
abb	bbb
aaabab	bbabbb
aababa	bbbaaaaaa
aaaaaaaaa	bbbbbabaa

B → b**BXX** | ε **X** → a | b

Storing Information in Nonterminals

- Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- Tying everything together:

$$\mathbf{S} \rightarrow \mathbf{A} \mid \mathbf{B}$$

$$\mathbf{A} \rightarrow \mathtt{a}\mathbf{AXX} \mid \varepsilon$$

$$\mathbf{B} \rightarrow \mathtt{b}\mathbf{BXX} \mid \varepsilon$$

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

Storing Information in Nonterminals

- Let Σ = {**a**, **b**} and let $L = \{w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same }.
- Tying everything together:

S → **A** | **B**

A → a**AXX** | ε

B → b**BXX** | ε

X → a | b

Overall strings in this language either follow the pattern of **A** or **B**.

Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- Tying everything together:

S → **A** | **B**

A → **aAXX** | **ε**

B → **bBXX** | **ε**

X → **a** | **b**

A represents "strings where the first third is **a**'s"

Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- Tying everything together:

S → **A** | **B**

A → **aAXX** | **ε**

B → **bBXX** | **ε**

X → **a** | **b**

B represents "strings where the first third is **b**'s"

Storing Information in Nonterminals

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{ w \in \Sigma^* \mid |w| \equiv_3 0$ and all the characters in the first third of w are the same $\}$.
- Tying everything together:

S → **A** | **B**

A → **a**AXX | ε

B → **b**BXX | ε

X → **a** | **b**

X represents "either an **a** or a **b**"

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`

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

Terminal — python — 66×21

```
>>> (26 + 42) * 2 + 1
```

How does my computer know what this sequence of characters means? How can it determine whether or not this expression is even syntactically valid?

Applications of CFGs

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

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

Given a set of production rules and an expression,

If I can somehow reverse engineer the derivation, I can ascribe meaning to the pieces of my string.

Exact details of how to do this are beyond the scope of this class – ***Take CS143!***

CFGs for Programming Languages

$$
\begin{aligned}
\textbf{BLOCK} &\rightarrow \textbf{STMT} \\
&\mid\ \texttt{\{}\ \textbf{STMTS}\ \texttt{\}} \\[1em]
\textbf{STMTS} &\rightarrow \varepsilon \\
&\mid\ \textbf{STMT STMTS} \\[1em]
\textbf{STMT} &\rightarrow \textbf{EXPR}\texttt{;} \\
&\mid\ \texttt{if (}\textbf{EXPR}\texttt{)}\ \textbf{BLOCK} \\
&\mid\ \texttt{while (}\textbf{EXPR}\texttt{)}\ \textbf{BLOCK} \\
&\mid\ \texttt{do}\ \textbf{BLOCK}\ \texttt{while (}\textbf{EXPR}\texttt{);} \\
&\mid\ \textbf{BLOCK} \\
&\mid\ \ldots \\[1em]
\textbf{EXPR} &\rightarrow \texttt{identifier} \\
&\mid\ \texttt{constant} \\
&\mid\ \textbf{EXPR}\ \texttt{+}\ \textbf{EXPR} \\
&\mid\ \textbf{EXPR}\ \texttt{-}\ \textbf{EXPR} \\
&\mid\ \textbf{EXPR}\ \texttt{*}\ \textbf{EXPR} \\
&\mid\ \ldots
\end{aligned}
$$

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 would you program it?