CMPU 240 · Theory of Computation

Context-free Grammars and Context-free Languages

5 March 2026

Exam 1

Graded soon (hopefully today); solutions posted on Ed.

Assignment 5

Out now

Where are we?

Describing languages

We've seen two models for the regular languages:

Finite automata recognize strings in the language.

They perform a computation to determine whether a specific string is in the language.

Regular expressions match strings in the language.

They describe the general shape of all strings in the language.

The *Pumping Lemma for Regular Languages* let us establish limits on what languages are regular.

To describe more complex languages, we need a more powerful formalism – context-free grammars.

The creation of a thousand forests is in one acorn.

Ralph Waldo Emerson

The creation of a *potentially infinite number of strings* is in one *grammar*.

Ralph Waldo Emerson,
theoretician

A *context-free grammar* (CFG) describes a language by recursively describing the structure of the strings in the language.

Context-free:

$A \rightarrow$ `foo`

Context-sensitive:

`bar` A `baz` $\rightarrow$ `bar foo baz`

We won't use these!

If G is a CFG with alphabet Σ and start symbol S, then the language of G is the set

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow}_G w\}$$

That is, $L(G)$ is the set of strings derivable from the start symbol using the production rules in grammar G.

Note that w must be in Σ^*, the set of strings made up only of terminals. Strings with variables aren't in the language!

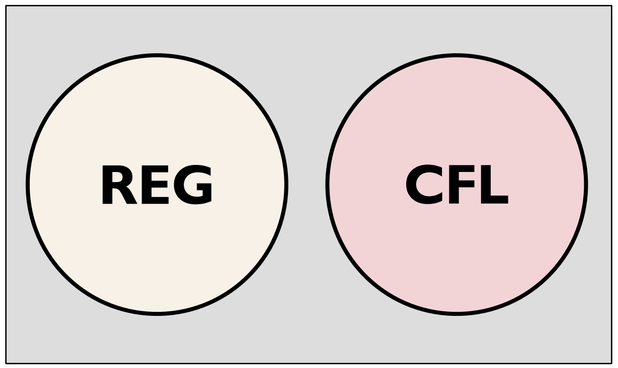
A language L is called a *context-free language* (CFL) if there is a CFG G such that $L = L(G)$.

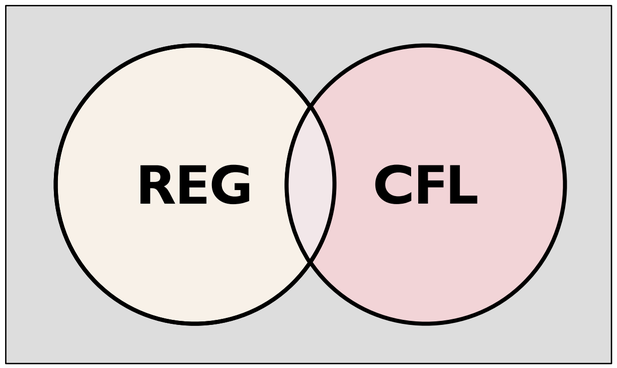
How are context-free and regular languages related?

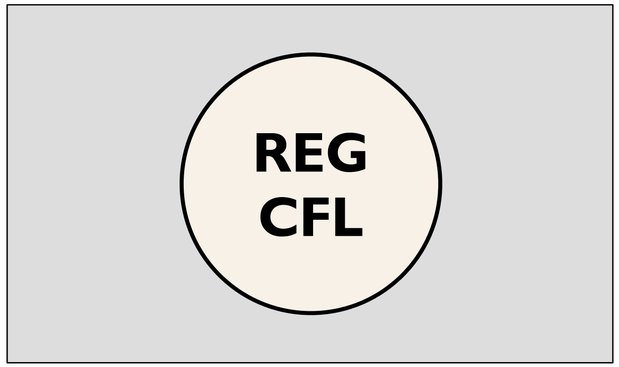
How do we design context-free grammars for context-free languages?

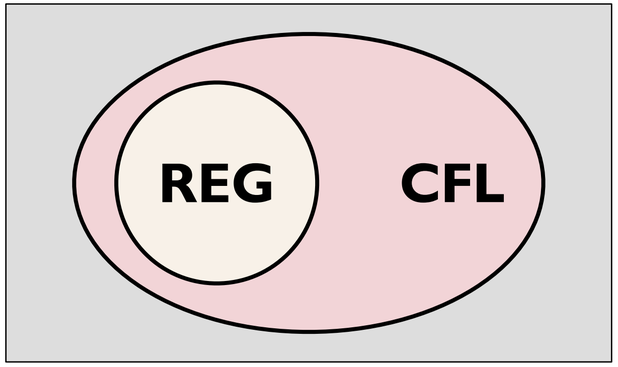
Regular and context-free languages

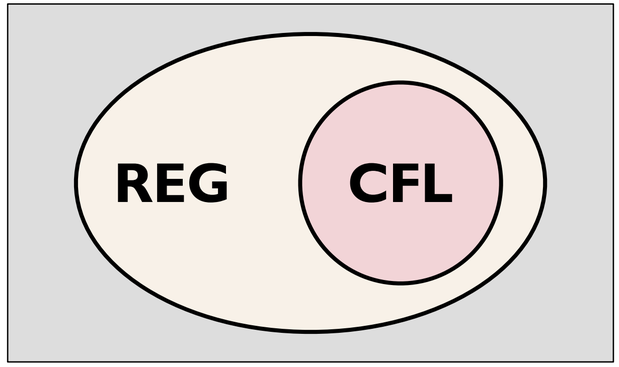
Possible relations











Consider the CFG G:

$S \rightarrow a^*b$

L(G) = …?

Beware!

Consider the CFG G:

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

$L(G) \neq \{\mathtt{a}^n\mathtt{b} \mid n \in \mathbb{N}_0\}$.

$L(G) = \{\mathtt{a}^*\mathtt{b}\}$. It contains exactly one string, $\mathtt{a}^*\mathtt{b}$.

Context-free grammars just consist of production rules with a concatenation of terminals and variables.

They don't have the regular expression operators Kleene star, union, or parenthesized expressions.

Context-free grammars just consist of production rules with a concatenation of terminals and variables.

They don't have the regular expression operators Kleene star, union, or parenthesized expressions.

But we can convert regular expressions to CFGs!

THEOREM Every regular language is context-free.

PROOF IDEA Show how to convert an arbitrary regular expression into a context-free grammar for the same language.

In regular expressions, we could write * for zero-or-more repetitions, e.g.,

```
a*b
```

In regular expressions, we could write * for zero-or-more repetitions, e.g.,

$S \rightarrow \texttt{a*b}$

In regular expressions, we could write * for zero-or-more repetitions, e.g.,

$S \rightarrow \mathbf{a^*}b$

In regular expressions, we could write * for zero-or-more repetitions, e.g.,

$S \rightarrow \mathtt{a}^*\mathtt{b}$

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

In regular expressions, we could write * for zero-or-more repetitions, e.g.,

$S \rightarrow \texttt{a*b}$

$A \rightarrow Aa \mid \varepsilon$

In regular expressions, we could write * for zero-or-more repetitions, e.g.,

$S \rightarrow Ab$

$A \rightarrow Aa \mid \varepsilon$

In regular expressions, we could write parentheses to group expressions and `u` for alternatives, e.g.,

```
a(b u c*)
```

In regular expressions, we could write parentheses to group expressions and `u` for alternatives, e.g.,

$S \rightarrow$ `a(b u c*)`

In regular expressions, we could write parentheses to group expressions and **u** for alternatives, e.g.,

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

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

In regular expressions, we could write parentheses to group expressions and `u` for alternatives, e.g.,

$S \rightarrow$ `a(b u c*)`

$X \rightarrow$ `b` | `c*`

In regular expressions, we could write parentheses to group expressions and **u** for alternatives, e.g.,

$S \rightarrow aX$

$X \rightarrow b \mid c^*$

In regular expressions, we could write parentheses to group expressions and **u** for alternatives, e.g.,

$S \rightarrow \mathbf{a}X$

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

In regular expressions, we could write parentheses to group expressions and **u** for alternatives, e.g.,

$S \rightarrow \mathbf{a}X$

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

$C \rightarrow Cc \mid \varepsilon$

In regular expressions, we could write parentheses to group expressions and **u** for alternatives, e.g.,

$S \rightarrow aX$

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

$C \rightarrow Cc \mid \varepsilon$

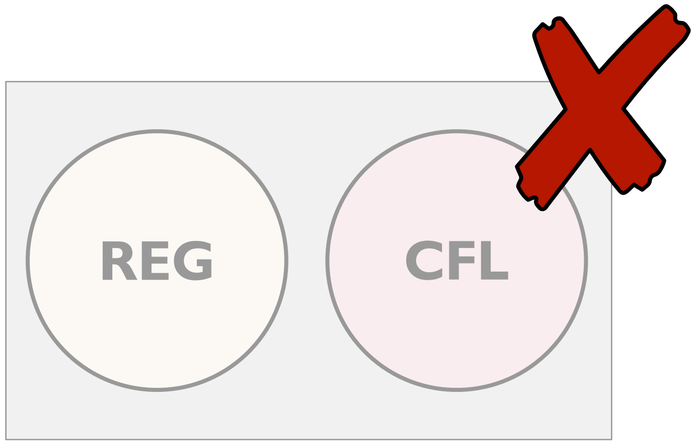
In regular expressions, we could write parentheses to group expressions and **u** for alternatives, e.g.,

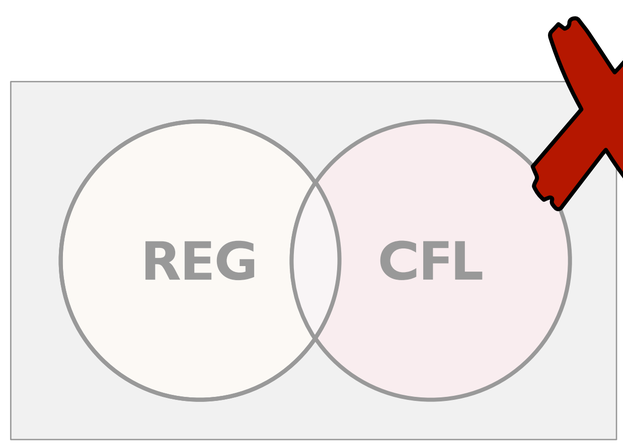
$S \rightarrow aX$

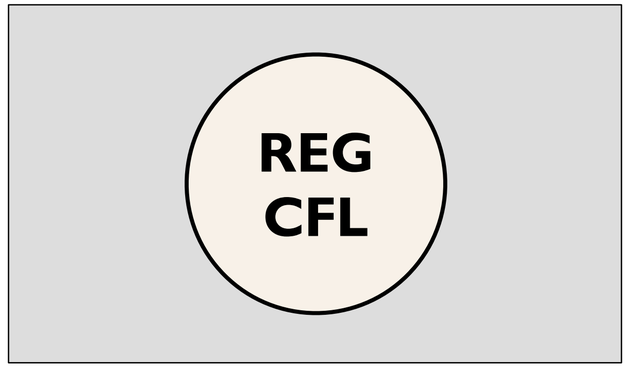
$X \rightarrow b \mid C$

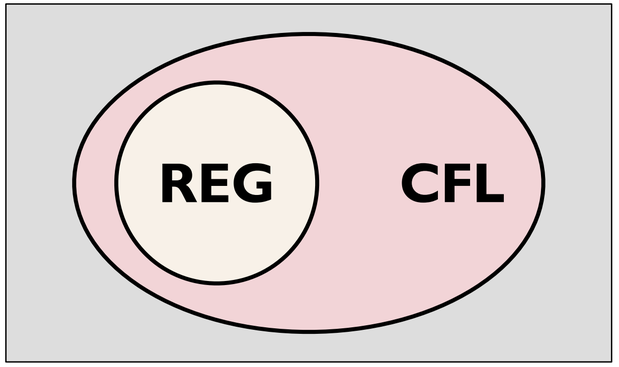
$C \rightarrow Cc \mid \varepsilon$

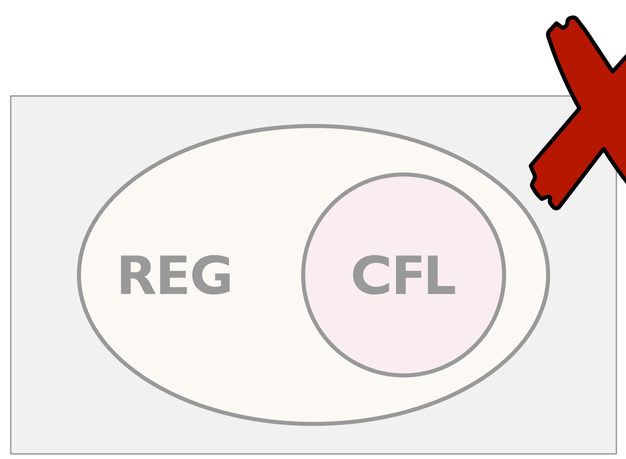
Possible relations











Consider the following CFG *G*:

$S \rightarrow aSb \mid \varepsilon$

What strings can this generate?

Consider the following CFG *G*:

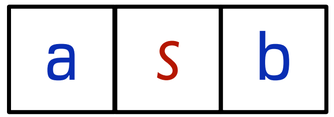
$S \rightarrow aSb \mid \varepsilon$

What strings can this generate?

Consider the following CFG *G*:

$S \rightarrow aSb \mid \varepsilon$

What strings can this generate?



Consider the following CFG *G*:

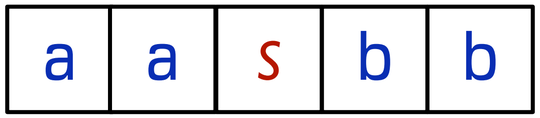
$S \rightarrow aSb \mid \varepsilon$

What strings can this generate?

Consider the following CFG *G*:

$S \rightarrow aSb \mid \varepsilon$

What strings can this generate?



Consider the following CFG *G*:

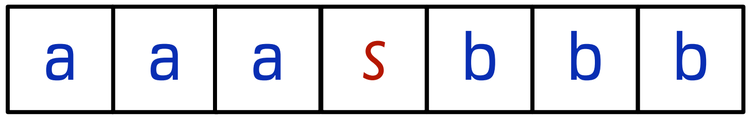
$S \rightarrow aSb \mid \varepsilon$

What strings can this generate?

Consider the following CFG *G*:

$S \rightarrow aSb \mid \varepsilon$

What strings can this generate?



Consider the following CFG *G*:

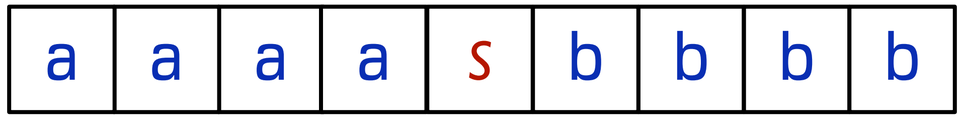
$S \rightarrow aSb \mid \varepsilon$

What strings can this generate?

Consider the following CFG *G*:

$S \rightarrow aSb \mid \varepsilon$

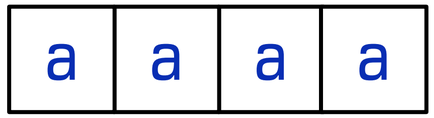
What strings can this generate?

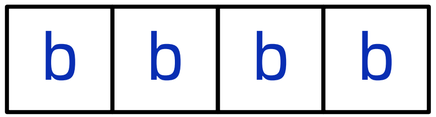


Consider the following CFG *G*:

$S \rightarrow aSb \mid \varepsilon$

What strings can this generate?

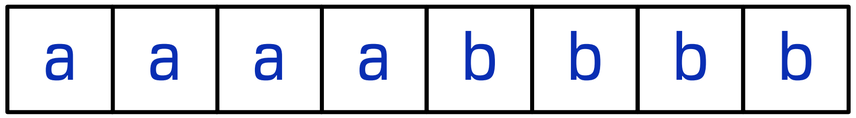




Consider the following CFG *G*:

$S \rightarrow aSb \mid \varepsilon$

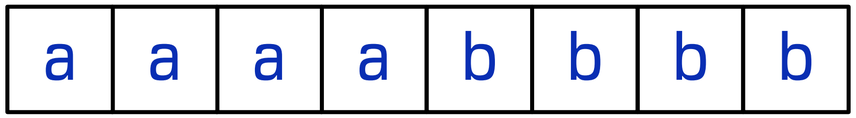
What strings can this generate?



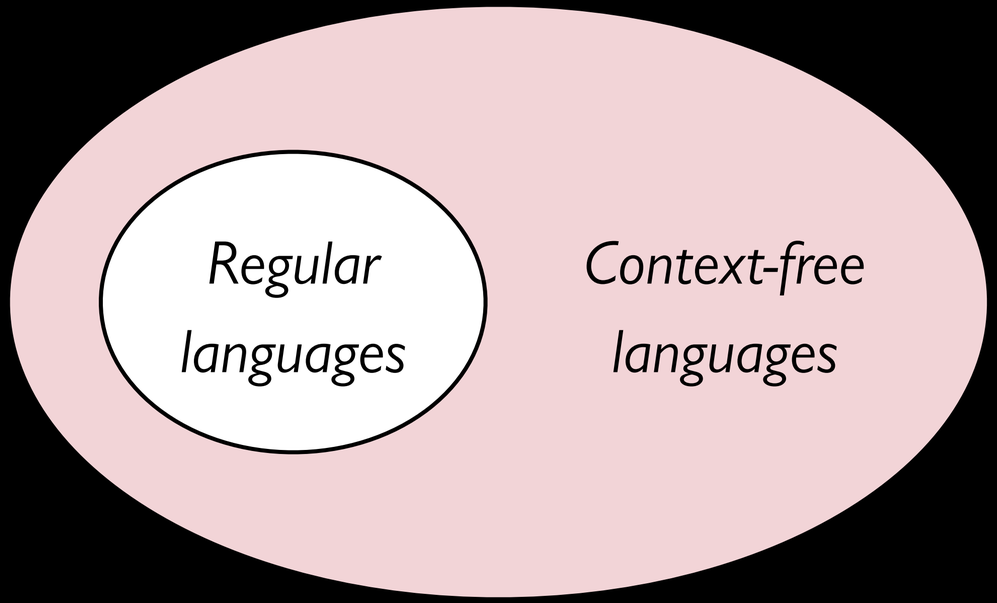
Consider the following CFG G:

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

What strings can this generate?



$$L(G) = \{a^n b^n \mid n \in \mathbb{N}_0\}$$



All languages

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded “memory”.

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory".

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

Why do CFGs have more power than regular expressions?

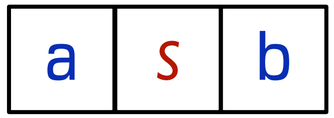
Intuition: Derivations of strings have unbounded "memory".

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

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory".

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



Why do CFGs have more power than regular expressions?

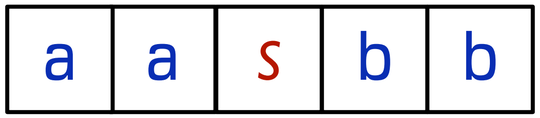
Intuition: Derivations of strings have unbounded "memory".

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

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory".

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



Why do CFGs have more power than regular expressions?

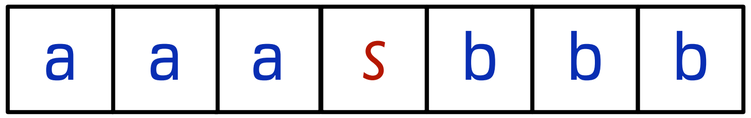
Intuition: Derivations of strings have unbounded "memory".

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

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory".

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



Why do CFGs have more power than regular expressions?

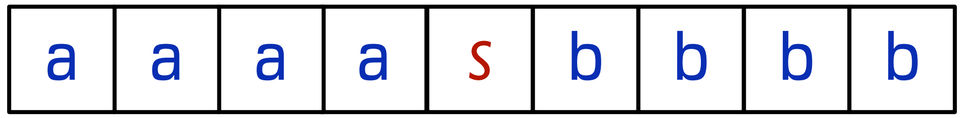
Intuition: Derivations of strings have unbounded "memory".

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

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory".

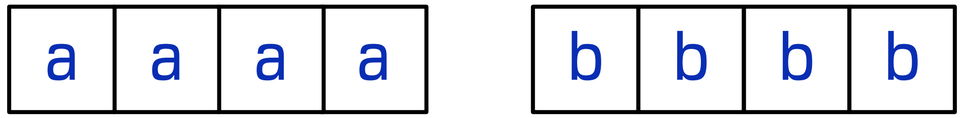
$$S \rightarrow aSb \mid \varepsilon$$



Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory".

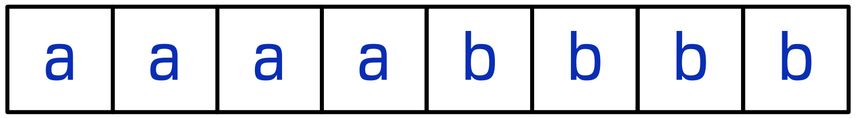
$$S \rightarrow aSb \mid \varepsilon$$



Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory".

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



Designing CFGs

Like writing code, designing a grammar is a craft.

Helpful strategies:

Think recursively.

Build bigger structures from smaller ones.

Have a construction plan.

Know what order you'll build up the string.

Store information in variables:

Have each variable correspond to some useful piece of information.

Example 1

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

Let $L = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}$.

Example 1

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

Let $L = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}$.

We can design a CFG for L by thinking inductively:

Base case: ε, a, and b are palindromes

Example 1

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

Let $L = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}$.

We can design a CFG for L by thinking inductively:

Base case: ε, a, and b are palindromes

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

Example 1

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

Let $L = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}$.

We can design a CFG for L by thinking inductively:

Base case: ε, a, and b are palindromes

Recursive case: If ω is a palindrome, then $a\omega a$ and $b\omega b$ are palindromes.

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

Example 1

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

Let $L = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}$.

We can design a CFG for L by thinking inductively:

Base case: ε, a, and b are palindromes

Recursive case: If ω is a palindrome, then $a\omega a$ and $b\omega b$ are palindromes.

No other strings are palindromes.

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

Example 1

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

Let $L = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}$.

We can design a CFG for L by thinking inductively:

Base case: ε, a, and b are palindromes

Recursive case: If ω is a palindrome, then $a\omega a$ and $b\omega b$ are palindromes.

No other strings are palindromes.

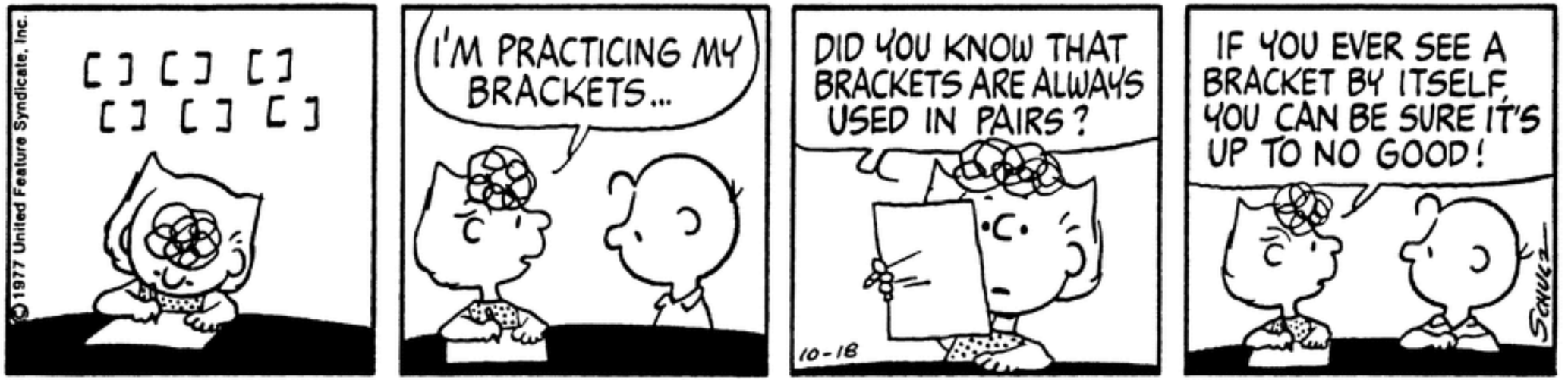
$S \rightarrow \varepsilon \mid a \mid b$

$S \rightarrow aSa \mid bSb$

Example 2

Let $\Sigma = \{$[,]$\}$.

Let $L = \{w \in \Sigma^* \mid w$ is a string of balanced brackets$\}$.



Example 2

Let $\Sigma = \{$`[`, `]`$\}$.

Let $L = \{w \in \Sigma^* \mid w$ is a string of balanced brackets$\}$.

Some sample strings in L:

- ε
- `[[[]]]`
- `[][]`
- `[[]][]`
- `[[][]][[]]`

Example 2

Let $\Sigma = \{$`[`, `]`$\}$.

Let $L = \{w \in \Sigma^* \mid w$ is a string of balanced brackets$\}$.

Let's think about this recursively:

Base case: The empty string is a string of balanced brackets.

Recursive case: Look at the closing bracket that matches the first open bracket.

`[[][[]][[]][[]][[]]]`

Example 2

Let $\Sigma = \{$[,]$\}$.

Let $L = \{w \in \Sigma^* \mid w$ is a string of balanced brackets$\}$.

Let's think about this recursively:

Base case: The empty string is a string of balanced brackets.

Recursive case: Look at the closing bracket that matches the first open bracket.

Example 2

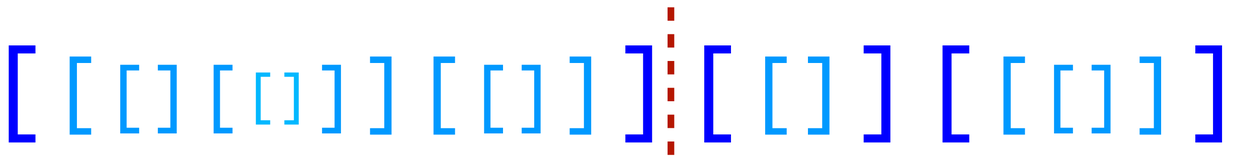
Let $\Sigma = \{$[,]$\}$.

Let $L = \{w \in \Sigma^* \mid w$ is a string of balanced brackets$\}$.

Let's think about this recursively:

Base case: The empty string is a string of balanced brackets.

Recursive case: Look at the closing bracket that matches the first open bracket.



Example 2

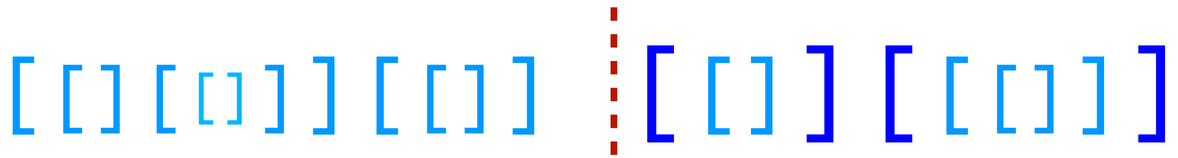
Let $\Sigma = \{$[,]$\}$.

Let $L = \{w \in \Sigma^* \mid w$ is a string of balanced brackets$\}$.

Let's think about this recursively:

Base case: The empty string is a string of balanced brackets.

Recursive case: Look at the closing bracket that matches the first open bracket.



Example 2

Let $\Sigma = \{[,]\}$.

Let $L = \{w \in \Sigma^* \mid w \text{ is a string of balanced brackets}\}$.

Let's think about this recursively:

Base case: The empty string is a string of balanced brackets.

Recursive case: Look at the closing bracket that matches the first open bracket. Removing the first bracket and the matching bracket forms two new strings of balanced brackets.

$$S \rightarrow \varepsilon \mid SS \mid [S]$$

Designing CFGs

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid n_a(w) = n_b(w)\}$

How many of the following CFGs have language L?

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

$S \to abS \mid baS \mid \varepsilon$

$S \to abSba \mid baSab \mid \varepsilon$

$S \to SbaS \mid SabS \mid \varepsilon$

Designing CFGs

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid n_a(w) = n_b(w)\}$

How many of the following CFGs have language L?

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

$S \to abS \mid baS \mid \varepsilon$

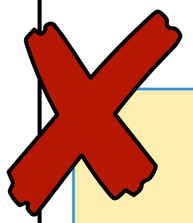
$S \to abSba \mid baSab \mid \varepsilon$

$S \to SbaS \mid SabS \mid \varepsilon$

Designing CFGs

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid n_a(w) = n_b(w)\}$

How many of the following CFGs have language L?



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

$S \to abS \mid baS \mid \varepsilon$

$S \to abSba \mid baSab \mid \varepsilon$

$S \to SbaS \mid SabS \mid \varepsilon$

Designing CFGs

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid n_a(w) = n_b(w)\}$

How many of the following CFGs have language L?

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

$S \to abS \mid baS \mid \varepsilon$

$S \to abSba \mid baSab \mid \varepsilon$

$S \to SbaS \mid SabS \mid \varepsilon$

Designing CFGs

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid n_a(w) = n_b(w)\}$

How many of the following CFGs have language L?

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

$S \to abS \mid baS \mid \varepsilon$

$S \to abSba \mid baSab \mid \varepsilon$

$S \to SbaS \mid SabS \mid \varepsilon$

Designing CFGs

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid n_a(w) = n_b(w)\}$

How many of the following CFGs have language L?

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

$S \to abS \mid baS \mid \varepsilon$

$S \to abSba \mid baSab \mid \varepsilon$

$S \to SbaS \mid SabS \mid \varepsilon$

Designing CFGs

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid n_a(w) = n_b(w)\}$

How many of the following CFGs have language L?

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

$S \to abS \mid baS \mid \varepsilon$

$S \to abSba \mid baSab \mid \varepsilon$

$S \to SbaS \mid SabS \mid \varepsilon$

Designing CFGs

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid n_a(w) = n_b(w)\}$

How many of the following CFGs have language L?

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

$S \to abS \mid baS \mid \varepsilon$

$S \to abSba \mid baSab \mid \varepsilon$

$S \to SbaS \mid SabS \mid \varepsilon$

Designing CFGs

Let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^* \mid n_a(w) = n_b(w)\}$

How many of the following CFGs have language L?

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

$S \to abS \mid baS \mid \varepsilon$

$S \to abSba \mid baSab \mid \varepsilon$

$S \to SbaS \mid SabS \mid \varepsilon$

Designing CFGs: Caveat 1

You need to make sure your grammar

- can generate *all* of the strings in the language
- can *never* generate a string outside the language

This can be tricky; you'll need to try test cases!

Designing CFGs: Caveat 2

Is this a CFG for the language $\{a^n b^n \mid n \in \mathbb{N}_0\}$?

$$S \to aSb$$

Designing CFGs: Caveat 2

Is this a CFG for the language $\{a^n b^n \mid n \in \mathbb{N}_0\}$?

$$S \to aSb$$

What strings can you derive?

Designing CFGs: Caveat 2

Is this a CFG for the language $\{a^n b^n \mid n \in \mathbb{N}_0\}$?

$$S \to aSb$$

What strings can you derive?

None!

Designing CFGs: Caveat 2

Is this a CFG for the language $\{a^n b^n \mid n \in \mathbb{N}_0\}$?

$$S \rightarrow aSb$$

What strings can you derive?

None!

What is the language of the grammar?

Designing CFGs: Caveat 2

Is this a CFG for the language $\{a^n b^n \mid n \in \mathbb{N}_0\}$?

$$S \to aSb$$

What strings can you derive?

None!

What is the language of the grammar?

$\varnothing$

Designing CFGs: Caveat 2

Is this a CFG for the language $\{a^n b^n \mid n \in \mathbb{N}_0\}$?

$$S \to aSb$$

What strings can you derive?

None!

What is the language of the grammar?

$\varnothing$

When designing CFGs, make sure your recursion actually terminates!

Designing CFGs: Caveat 3

Remember that each variable can be expanded out independently of the others.

E.g., let $\Sigma = \{\mathtt{a}, \mathtt{=}\}$ and $L = \{\mathtt{a}^n\mathtt{=}\mathtt{a}^n \mid n \in \mathbb{N}_0\}$.

$S \to X\mathtt{=}X$

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

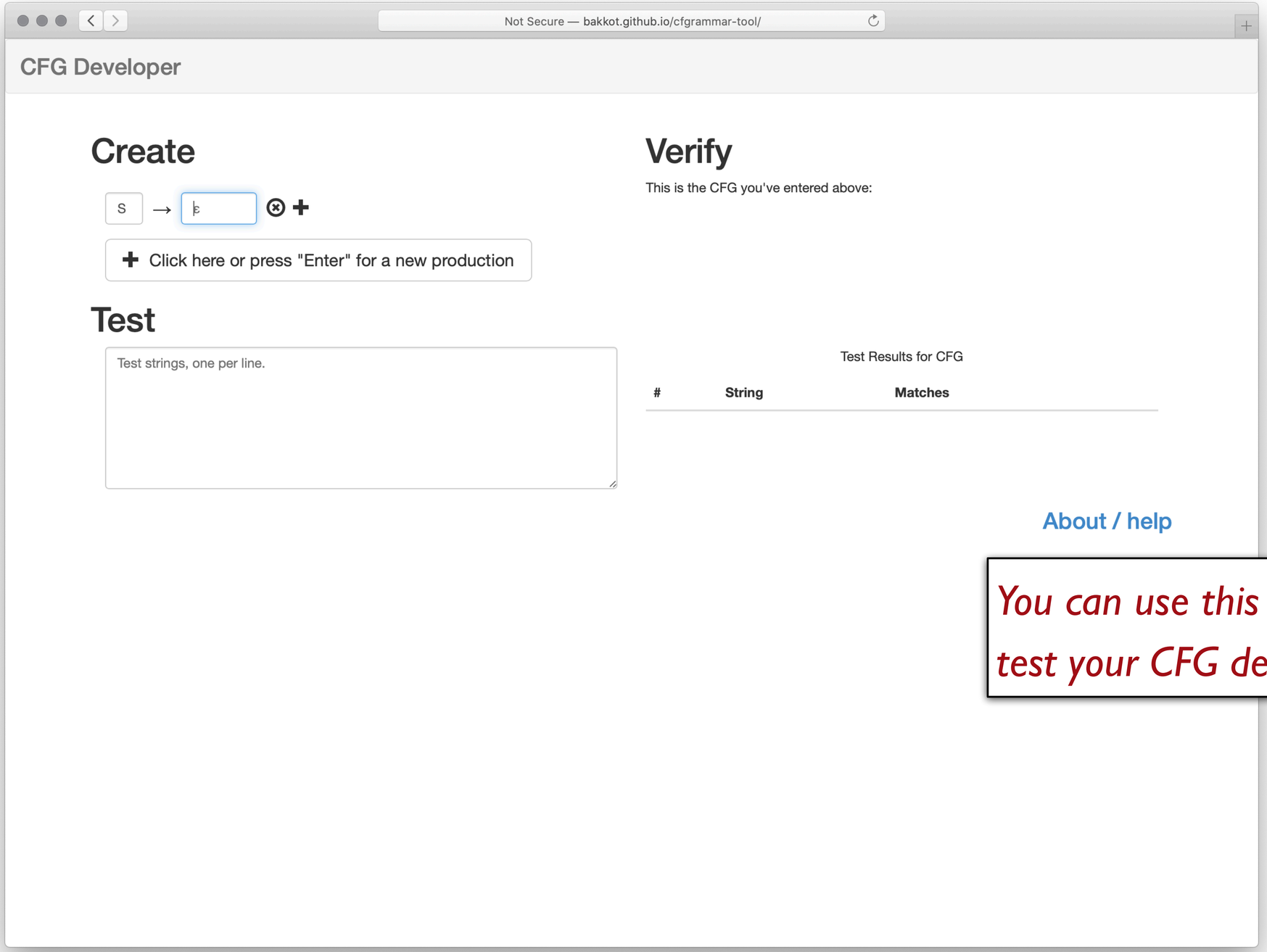
$$
\begin{aligned}
& S \\
\Rightarrow\ & X\mathtt{=}X \\
\Rightarrow\ & \mathtt{a}X\mathtt{=}X \\
\Rightarrow\ & \mathtt{aa}X\mathtt{=}X \\
\Rightarrow\ & \mathtt{aa=}X \\
\Rightarrow\ & \mathtt{aa=a}X \\
\Rightarrow\ & \mathtt{aa=a}
\end{aligned}
$$

Designing CFGs: Caveat 3

We can force a build order, so the string is built from the ends inward, balancing the sides of the equality:

$S \rightarrow aSa \mid =$

$$
\begin{aligned}
& S \\
\Rightarrow\ & aSa \\
\Rightarrow\ & aaSaa \\
\Rightarrow\ & aaaSaaa \\
\Rightarrow\ & aaa=aaa
\end{aligned}
$$


Not Secure — bakkot.github.io/cfgrammar-tool/
CFG Developer
Create
S
→
ε
Click here or press "Enter" for a new production
Test
Test strings, one per line.
Verify
This is the CFG you've entered above:
Test Results for CFG
String Matches
About / help
You can use this tool to test your CFG designs!

Language: Function prototypes

Let Σ = {`void`, `int`, `double`, `name`, `(`, `)`, `,`, `;`}

Let's write a CFG for C-style function prototypes!

Language: 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);
```

$$
\begin{aligned}
S &\rightarrow \mathit{Ret}\ \texttt{name (}\mathit{Args}\texttt{);} \\
\mathit{Ret} &\rightarrow \mathit{Type} \mid \texttt{void} \\
\mathit{Type} &\rightarrow \texttt{int} \mid \texttt{double} \\
\mathit{Args} &\rightarrow \varepsilon \mid \texttt{void} \mid \mathit{ArgList} \\
\mathit{ArgList} &\rightarrow \mathit{OneArg} \mid \mathit{ArgList}\texttt{,}\ \mathit{OneArg} \\
\mathit{OneArg} &\rightarrow \mathit{Type} \mid \mathit{Type}\ \texttt{name}
\end{aligned}
$$

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 variables to represent different structures.

Acknowledgments

This lecture incorporates material from:

Nancy Ide, Vassar College

Keith Schwarz, Stanford University

Michael Sipser, *Introduction to the Theory of Computation*

Jeffrey Ullman, Stanford University