

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 description of a language by recursively describing the structure of the strings in the language.
- CFGs are best explained by example...

Arithmetic Expressions

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

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

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

Arithmetic Expressions

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

E → int

E → **E Op E**

E → (**E**)

Op → +

Op → -

Op → ×

Op → /

⇒ **E**

⇒ **E Op E**

⇒ **E Op int**

⇒ int **Op** int

⇒ int / int

Context-Free Grammars

- Formally, a context-free grammar is a collection of four items:
 - a set of **nonterminal symbols** (also called **variables**),
 - 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 → **x**

Op → **/**

Some CFG Notation

- In today's slides, capital letters in **Bold Red Uppercase** will represent nonterminals.
 - e.g. **A, B, C, D**
- Lowercase letters in **blue monospace** will represent terminals.
 - e.g. **t, u, v, w**
- Lowercase Greek letters in *gray italics* will represent arbitrary strings of terminals and nonterminals.
 - e.g. *α, γ, ω*
- You don't need to use these conventions on your own; just make sure whatever you do is readable. ☺

A Notational Shorthand

E → int

E → **E Op E**

E → (**E**)

Op → +

Op → -

Op → ×

Op → /

A Notational Shorthand

E → int | **E Op E** | (**E**)

Op → + | - | × | /

Derivations

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

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

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

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

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

- That is, $\mathcal{L}(G)$ is the set of strings of terminals derivable from the start symbol.

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

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

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

$$\begin{aligned} S &\rightarrow Sa \mid dT \\ T &\rightarrow bTb \mid c \end{aligned}$$

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

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

$$S \rightarrow a^*b$$

From Regexes to CFGs

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

$$S \rightarrow Ab$$

$$A \rightarrow Aa \mid \epsilon$$

From Regexes to CFGs

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

$$S \rightarrow a (b \cup c^*)$$

From Regexes to CFGs

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

$$S \rightarrow aX$$

$$X \rightarrow b \mid C$$

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

Regular Languages and CFLs

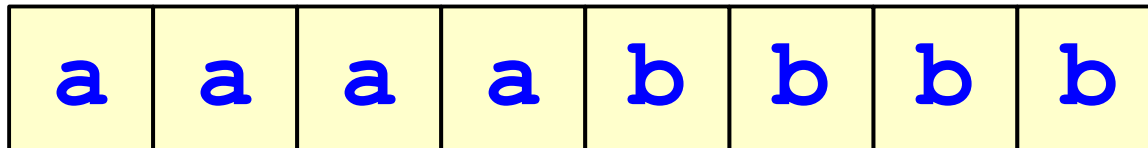
- ***Theorem:*** Every regular language is context-free.
- ***Proof Idea:*** Use the construction from the previous slides to convert a regular expression for L into a CFG for L . ■
- ***Great Exercise:*** Instead, show how to convert a DFA/NFA into a CFG.

The Language of a Grammar

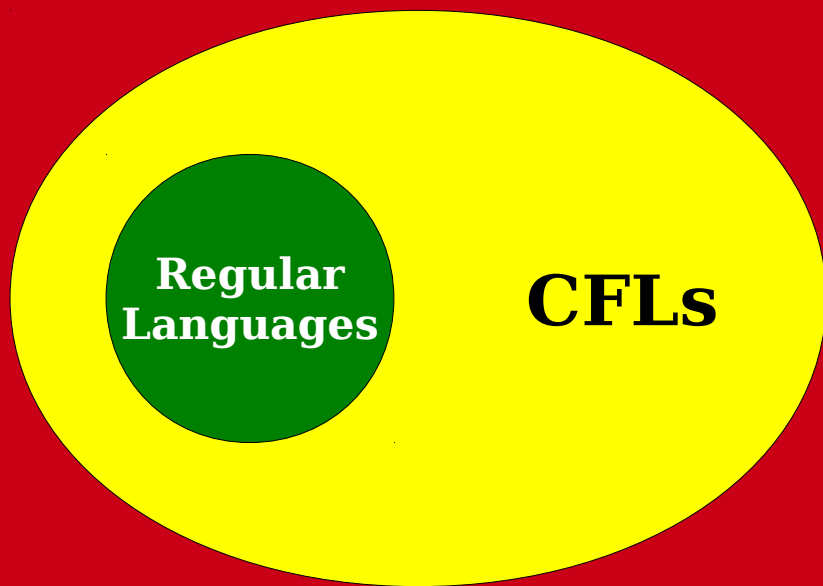
- Consider the following CFG G :

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

- What strings can this generate?



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

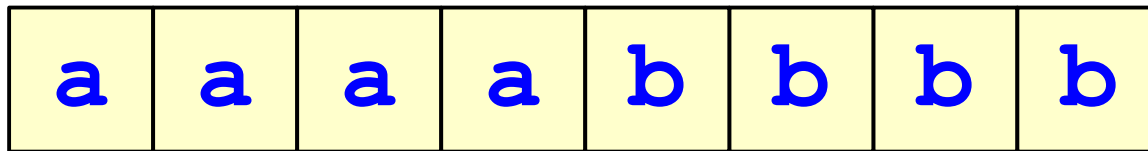


All Languages

Why the Extra Power?

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

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



Time-Out for Announcements!

Midterm Exam Logistics

- The next midterm is **Monday, November 12th** from **7:00PM - 10:00PM**. Locations are divvied up by last (family) name:
 - A-L: Go to **Bishop Auditorium**.
 - M-Z: Go to **Cubberley Auditorium**.
- The exam focuses on Lecture 06 – 13 (binary relations through induction, inclusive) and PS3 – PS5. Finite automata onward is *not* tested.
 - Topics from earlier in the quarter (proofwriting, first-order logic, set theory, etc.) are also fair game, but that's primarily because the later material builds on this earlier material.
- 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.

Our Advice

- ***Eat dinner the night of the exam.*** You are not a brain in a jar. You are a rich, complex, beautiful biological system. Please take care of yourself.
- ***Read all the questions before diving into them.*** Tunnel vision can hurt you on an exam. There's evidence that spreading your time out leads to better outcomes.
- ***Reflect on how far you've come.*** How many of these questions would you have been able to *understand* two months ago? That's the mark that you're learning something!

Your Questions

“What do you like most about teaching CS103?”

The sense that everyone gets when they turn around and see how high they've climbed.

Three Questions

- What is something you know now that, at the start of the quarter, you knew you didn't know?
- What is something you know now that, at the start of the quarter, you *didn't* know that you didn't know?
- What is something you *don't* know that, at the start of the quarter, you *didn't* know that you didn't know?

“Have you ever been in love before?”

Yep!

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 $L = \{w \in \Sigma^* \mid w \text{ is a palindrome}\}$
- We can design a CFG for L by thinking inductively:
 - Base case: ε , a , and b are palindromes.
 - If w is a palindrome, then $aw a$ and $bw b$ are palindromes.
 - No other strings are palindromes.

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

Designing CFGs

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

$\{\{\}\}$

$\{\}\{\}$

$\{\}\{\}\{\}\{\}$

$\{\{\{\}\}\}\{\}\{\}$

ϵ

$\{\}\{\}$

Designing CFGs

- Let $\Sigma = \{\{, \}\}$ and let $L = \{w \in \Sigma^* \mid w \text{ is a string of balanced braces}\}$
- 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 $L = \{ w \in \Sigma^* \mid w \text{ is a string of balanced braces} \}$
- 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 $L = \{ w \in \Sigma^* \mid w \text{ is a string of balanced braces} \}$
- 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.

$$S \rightarrow \{S\}S \mid \epsilon$$

Designing CFGs

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

How many of the following CFGs have language L ?

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

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

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

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

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

$$S \rightarrow aSb$$

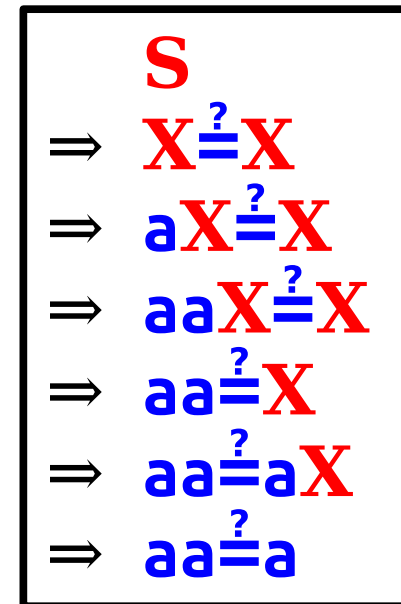
- What strings in $\{a, b\}^*$ can you derive?
 - Answer: ***None!***
- What is the language of the grammar?
 - Answer: \emptyset
- When designing CFGs, make sure your recursion actually terminates!

Designing CFGs

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

$$S \rightarrow X \underline{a} X$$

$$X \rightarrow aX \mid \epsilon$$



A box containing a derivation sequence for the string 'aaa' using the given grammar rules. The sequence is as follows:

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

Finding a Build Order

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

$$S \rightarrow \stackrel{?}{=} \mid aSa$$

	S
\Rightarrow	aSa
\Rightarrow	aaSaa
\Rightarrow	aaaSaaa
\Rightarrow	aaa[?]aaa

Function Prototypes

- Let $\Sigma = \{\text{void, int, double, name, (,), ,, ;}\}$.
- 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 →  $\epsilon$   
        | 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 would you program it?