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:

E → int

E → E Op E

E → (E)

Op → +

Op → -

Op → *

Op → /

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E * (E Op E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int Op int)

⇒ int * (int + 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 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. ***α***, ***γ***, ***ω***

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 → **E Op E** | `int` | (**E**)
Op → `+` | `*` | `-` | `/`

E
⇒ **E Op E**
⇒ **E Op** `(`**E**`)`
⇒ **E Op** `(`**E Op E**`)`
⇒ **E** `* (`**E Op E**`)`
⇒ `int * (`**E Op E**`)`
⇒ `int * (int` **Op E**`)`
⇒ `int * (int` **Op** `int)`
⇒ `int * (int + int)`

- 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 **E** $\Rightarrow^*$ `int * (int + int)`.

The Language of a Grammar

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

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

- That is, $\mathscr{L}(G)$ is the set of strings derivable from the start symbol.
- Note: $\boldsymbol{\omega}$ must be in Σ*, the set of strings made from terminals. Strings involving nonterminals aren't in the language.

Context-Free Languages

- A language L is called a ***context-free language*** (or CFL) if there is a CFG G such that $L = \mathscr{L}(G)$.
- Questions:
  - What languages are context-free?
  - How are context-free and regular languages related?

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → **`a*b`**

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

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

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → `a(b ∪ c*)`

From Regexes to CFGs

- CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.
- However, we can convert regular expressions to CFGs as follows:

S → **aX**

X → **b** | **C**

C → **Cc** | **ε**

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

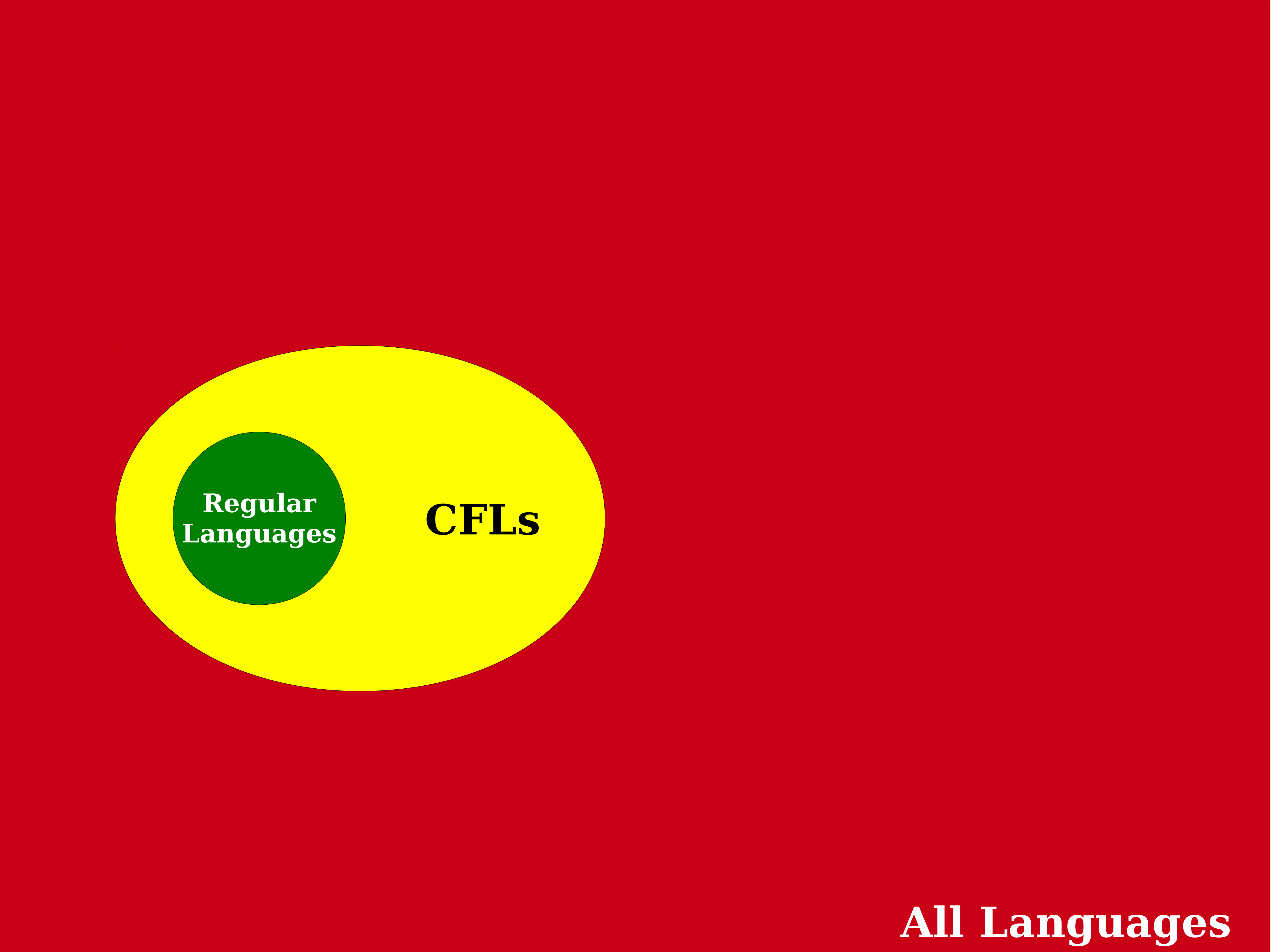
- Consider the following CFG G:

$$\mathtt{S} \to \mathtt{aSb} \mid \varepsilon$$

- 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.”

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

a	a	a	a	b	b	b	b

Time-Out for Announcements!

Problem Sets

- Problem Set Six was due at the start of class; feel free to turn it in by the start of Monday's lecture using late days.
- Problem Set Seven goes out today. It's due on Friday of next week.
  - Play around with the limits of regular languages!
  - Explore context-free grammars!
  - See the interplay of automata and CFGs!
- As always, feel free to stop by office hours or ask questions on Piazza.

A Reminder: The Honor Code

- This is the point in the quarter where we start to see a *lot* of cheating cases. Please take the following to heart:
  - ***It is not the end of the world if you can't figure out all the problems on a problem set***. Each problem set is worth something like 3% of your total grade in this course. Skipping a problem set is not going to tank your grade.
  - ***If you know someone in this class who is unhealthily stressed out, please reach out to them***. We all need to look out for each other and take care of each other. If you know someone who is really hurting, be there for them.
  - ***If you are feeling overwhelmed by this class, please come talk to us***. This class is hard, but it's not supposed to make you suffer. If things aren't going well, we'd be happy to discuss your options.
  - ***You are not your grades. You are a human being***. I mentioned this earlier, but it is not the end of the world if you get a low grade, withdraw from this class, or fail this class. It does not reflect poorly on you. It does not make you worthless. From experience, obsessing over your grades will make you miserable in the long term. Please don't make the same mistakes I made.

Your Questions!

“I am feeling overwhelmed right now with all my classes. Second wave of midterms have already started for me, and I am finding it hard to balance studying for the midterms and staying on top of the day to day material for all my classes. Any tips?”

I'm sorry to hear that. That can be really rough. There is no “one size fits all” solution here. Your goal is to make sure that not all of the following are true:

- you have too many things to do,
- you have to do all of them well,
- you don't enjoy them,
- you have limited time to do them, and
- this process repeats.

Try addressing each of these independently. If you can address the root causes of each, you will probably end up a lot happier.

Three Questions

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

"I heard rumors that there are plans to make the CS department more difficult to get into, because there's so many people trying to major in CS. Is this true? Also, what are some of the plans out there to tackle the influx of so many undergrads?"

I haven't heard anything like this – it goes against so much of what makes Stanford Stanford. We've had a lot of discussions about how to deal with logistical issues from large class sizes and are considering things like increasing the number of offerings of each class, having multiple people teach the same class each quarter, hiring more staff, etc., but nothing like this.

We'd consider it a huge failure on our part if we made it harder to get into CS. That would undo so much of the work we've done in making CS more accessible and interesting and would totally poison the culture in the department.

Back to CS103!

Designing CFGs

- Like designing DFAs, NFAs, and regular expressions, designing CFGs is a craft.
- When thinking about CFGs:
  - ***Think recursively:*** Build up bigger structures from smaller ones.
  - ***Have a construction plan:*** Know in what order you will build up the string.
  - ***Store information in nonterminals:*** Have each nonterminal correspond to some useful piece of information.

Designing CFGs

- Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{w \in \Sigma^* \mid w$ is a palindrome $\}$
- We can design a CFG for L by thinking inductively:
  - Base case: ε, **a**, and **b** are palindromes.
  - If ω is a palindrome, then **a**ω**a** and **b**ω**b** are palindromes.

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

Designing CFGs

- Let Σ = { (,) } and let L = {w ∈ Σ* | w is a string of balanced parentheses }
- Some sample strings in L:

((()))

(())()

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

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

ε

()()

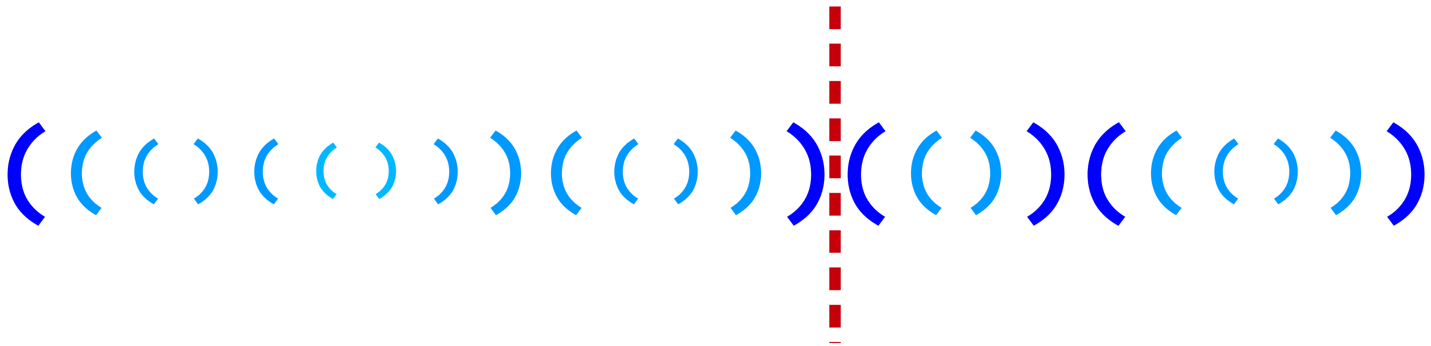
Designing CFGs

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

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

Designing CFGs

- Let Σ = { (,) } and let L = {w ∈ Σ* | w is a string of balanced parentheses }
- Let's think about this recursively.
  - Base case: the empty string is a string of balanced parentheses.
  - Recursive step: Look at the closing parenthesis that matches the first open parenthesis.



Designing CFGs

- Let Σ = { (,) } and let L = {w ∈ Σ* | w is a string of balanced parentheses }
- Let's think about this recursively.
  - Base case: the empty string is a string of balanced parentheses.
  - Recursive step: Look at the closing parenthesis that matches the first open parenthesis. Removing the first parenthesis and the matching parenthesis forms two new strings of balanced parentheses.

$$\mathbf{S} \rightarrow \mathbf{(S)S} \mid \boldsymbol{\varepsilon}$$

Designing CFGs: A Caveat

- Let Σ = {**a**, **b**} and let L = {w ∈ Σ* | w has the same number of **a**'s and **b**'s }
- Is this a CFG for L?

S → **aSb** | **bSa** | **ε**

- Can you derive the string **abba**?

Designing CFGs: A Caveat

- When designing a CFG for a language, make sure that it
  - generates all the strings in the language and
  - never generates a string outside the language.
- The first of these can be tricky – make sure to test your grammars!
- You'll design your own CFG for this language on the next problem set.

CFG Caveats II

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

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

- 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?

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

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

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

Finding a Build Order

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

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

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

Function Prototypes

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

Function Prototypes

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

Summary of CFG Design Tips

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

Applications of Context-Free Grammars

CFGs for Programming Languages

```
BLOCK   →  STMT
        |  { STMTS }

STMTS   →  ε
        |  STMT STMTS

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

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

Grammars in Compilers

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

Natural Language Processing

- By building context-free grammars for actual languages and applying statistical inference, it's possible for a computer to recover the likely meaning of a sentence.
    - In fact, CFGs were first called ***phrase-structure grammars*** and were introduced by Noam Chomsky in his seminal work *Syntactic Structures*.
    - They were then adapted for use in the context of programming languages, where they were called ***Backus-Naur forms***.
- Stanford's CoreNLP project is one place to look for an example of this.
- Want to learn more? Take CS124 or CS224N!

Next Time

- **Turing Machines**
  - What does a computer with unbounded memory look like?
  - How do you program them?