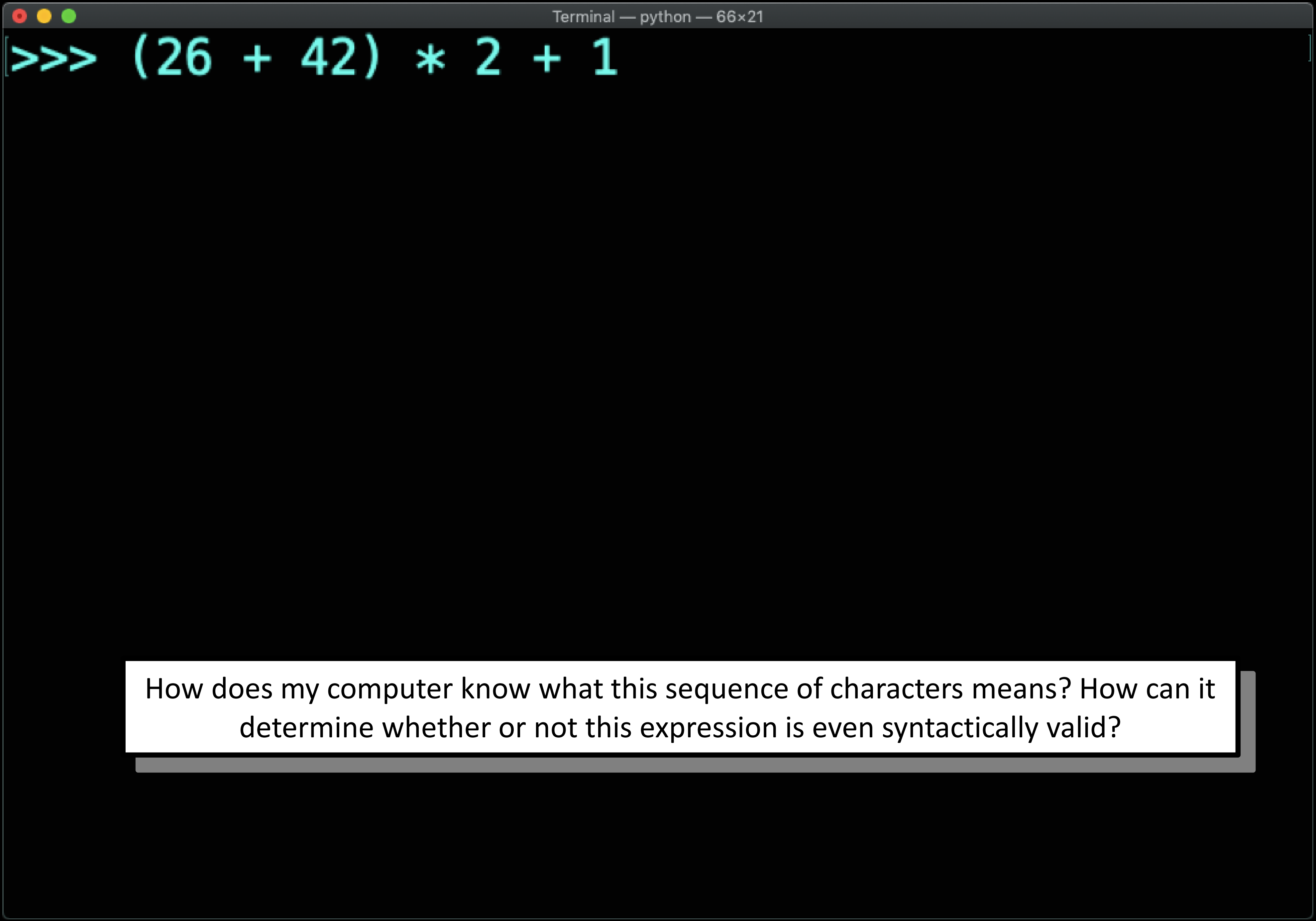
Context-Free Grammars

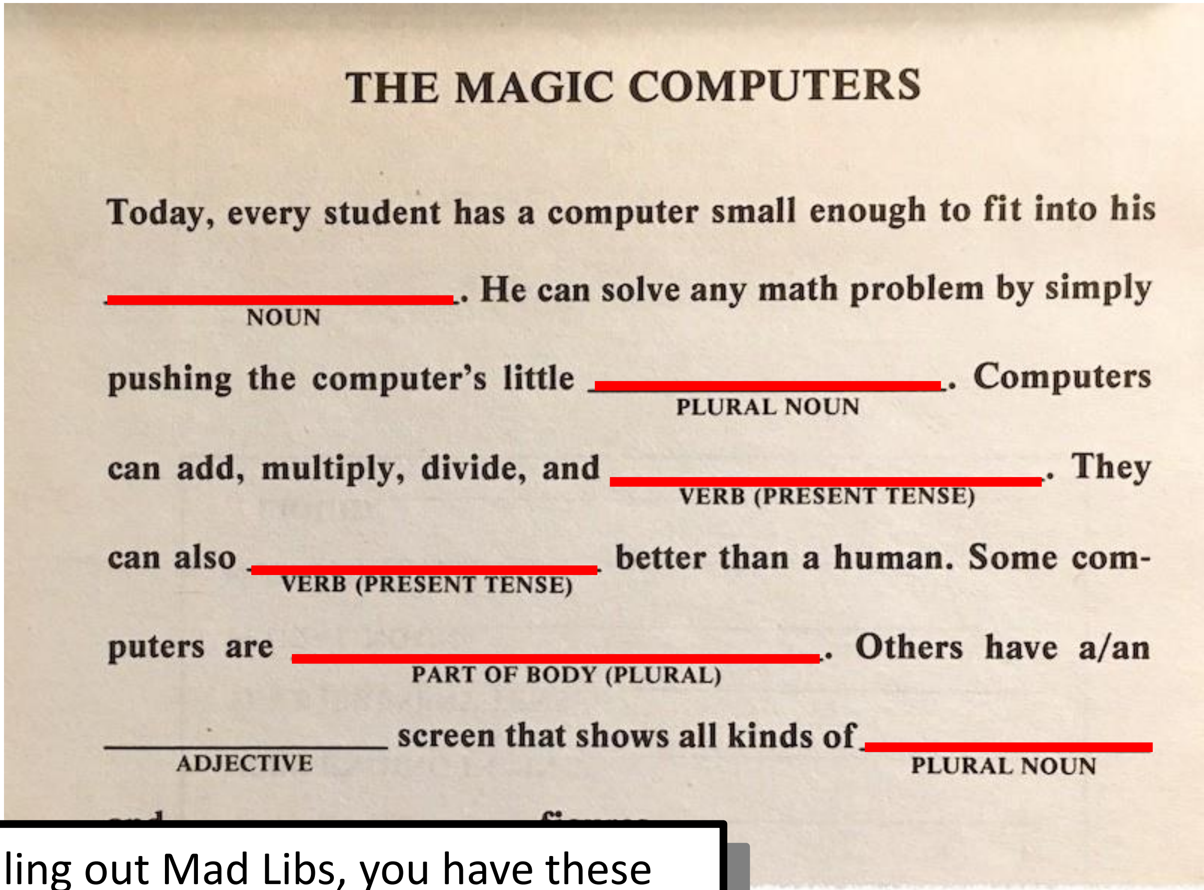
A Motivating Question



```
>>> (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
________________. He can solve any math problem by simply
NOUN
pushing the computer's little ________________. Computers
PLURAL NOUN
can add, multiply, divide, and ________________. They
VERB (PRESENT TENSE)
can also ________________ better than a human. Some com-
VERB (PRESENT TENSE)
puters are ________________. Others have a/an
PART OF BODY (PLURAL)
________________ screen that shows all kinds of ________________
ADJECTIVE
PLURAL NOUN
and ________________ figures.

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:

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 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

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 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**$\}$:

S → **Sa** | **dT**

T → **bTb** | **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 $L = \mathscr{L}(G)$.

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:

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

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:

S → **`a*b`**

A → **A`a`** | **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:

S → **A**`b`

A → **A**`a` **|** `ε`

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 ∪ c*)`

From Regexes to CFGs

CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.

However, we can convert regular expressions to CFGs as follows:

S → `a(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 → `a(b ∪ c*)`

X → `b` | `c*`

From Regexes to CFGs

CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.

However, we can convert regular expressions to CFGs as follows:

S → `a(b ∪ c*)`

X → `b` | `c*`

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`**X**

X → `b` | `c*`

From Regexes to CFGs

CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.

However, we can convert regular expressions to CFGs as follows:

S → `a`**X**
X → `b` | `c*`

From Regexes to CFGs

CFGs consist purely of production rules of the form **A** → ***ω***. They do not have the regular expression operators * or ∪.

However, we can convert regular expressions to CFGs as follows:

S → **`a`X**
X → `b` | **`c*`**
C → **C`c`** | **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:

S → aX
X → b | c*
C → Cc | 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:

S → `a`**X**
X → `b` | **C**
C → **C**`c` | 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. ■

Great Exercise: Instead, show how to convert a DFA/NFA into a CFG.

The Language of a Grammar

Consider the following CFG *G*:

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

What strings can this generate?

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?

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

The Language of a Grammar

Consider the following CFG *G*:

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

What strings can this generate?

a **S** **b**

The Language of a Grammar

Consider the following CFG *G*:

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

What strings can this generate?

a S b

The Language of a Grammar

Consider the following CFG *G*:

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

What strings can this generate?

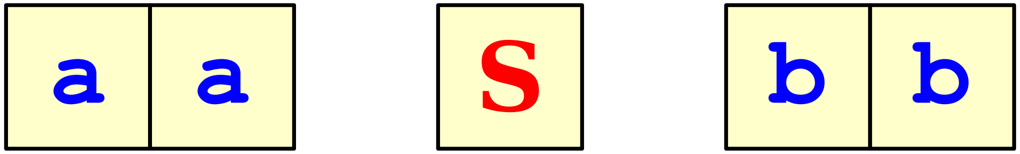
a	a	S	b	b

The Language of a Grammar

Consider the following CFG *G*:

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

What strings can this generate?



a a S b b

The Language of a Grammar

Consider the following CFG *G*:

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

What strings can this generate?

a	a	a	S	b	b	b

The Language of a Grammar

Consider the following CFG *G*:

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

What strings can this generate?

a	a	a	S	b	b	b

The Language of a Grammar

Consider the following CFG *G*:

S → **aSb** | **ε**

What strings can this generate?

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

The Language of a Grammar

Consider the following CFG *G*:

S → **`a`S`b`** | **ε**

What strings can this generate?

`a`	**`a`**	**`a`**	**`a`**

`b`	**`b`**	**`b`**	**`b`**

The Language of a Grammar

Consider the following CFG *G*:

S → **aSb** | **ε**

What strings can this generate?

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

The Language of a Grammar

Consider the following CFG G:

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

What strings can this generate?

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

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

Regular Languages

CFLs

All Languages

Why the Extra Power?

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory."

Why the Extra Power?

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory."

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

Why the Extra Power?

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory."

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

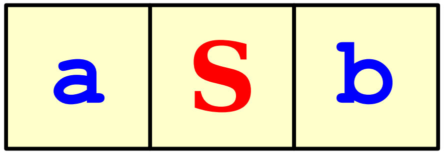
$$\boxed{S}$$

Why the Extra Power?

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded “memory.”

S → **aSb** | **ε**



a **S** **b**

Why the Extra Power?

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory."

$S \to aSb \mid \varepsilon$

a S b

Why the Extra Power?

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory."

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

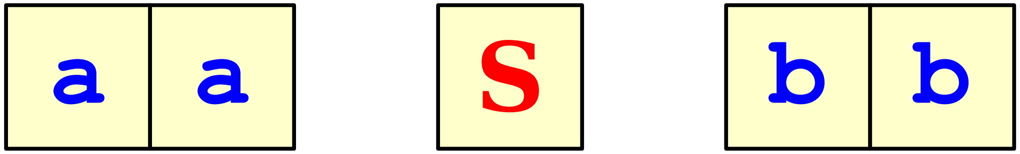
a	**a**	**S**	**b**	**b**

Why the Extra Power?

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory."

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



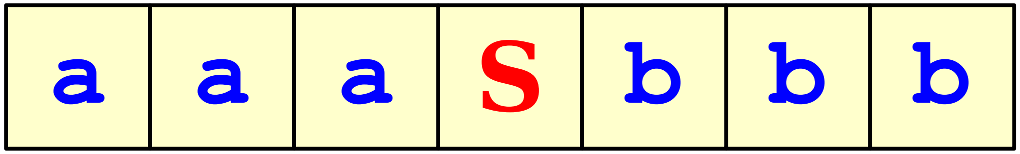
a a S b b

Why the Extra Power?

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory."

S → **aSb** | **ε**



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."

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

`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 → **aSb** | **ε**

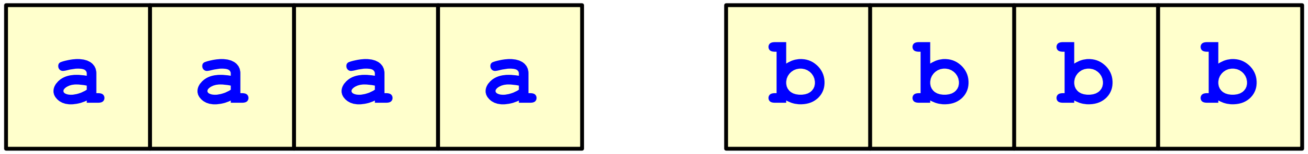
a	**a**	**a**	**a**	**S**	**b**	**b**	**b**	**b**

Why the Extra Power?

Why do CFGs have more power than regular expressions?

Intuition: Derivations of strings have unbounded "memory."

S → **aSb** | **ε**



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	a	a	b	b	b	b

Your Questions

What is the next hot thing in CS/software (fad or otherwise)? In the past decade, things like AI/ML, IOT, and blockchain have become buzzwords – what’s next?

Staff recommendations for favorite CS or math books?

Favorite video game?

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 Σ = {**a**, **b**} and let *L* = {*w* ∈ Σ* | *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.

No other strings are palindromes.

S → ε | a | b | aSa | bSb

Designing CFGs

Let $\Sigma = \{$**a**, **b**$\}$ and let $L = \{w \in \Sigma^* \mid w$ is a palindrome $\}$

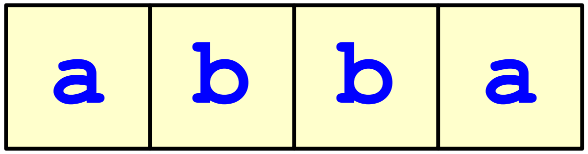
We can design a CFG for L by thinking inductively:

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

Designing CFGs

Let Σ = {**a**, **b**} and let *L* = {*w* ∈ Σ* | *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 $\Sigma = \{\texttt{a}, \texttt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w \text{ is a palindrome } \}$

We can design a CFG for L by thinking inductively:

a	b	a	b	b	a	b	a

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

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 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 $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w \text{ is a palindrome } \}$

We can design a CFG for L by thinking recursively:

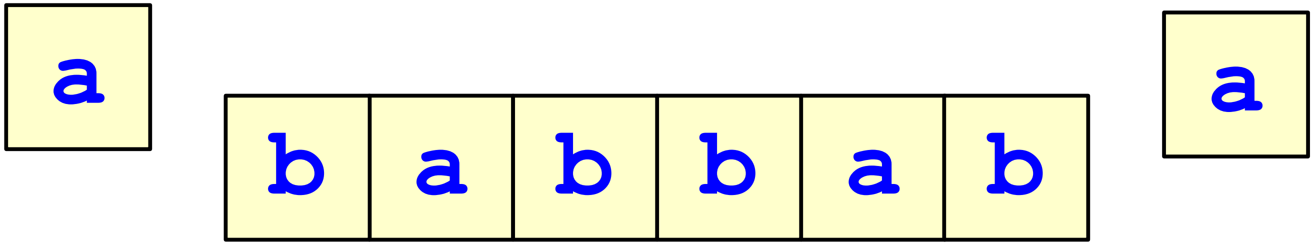
a	b	a	b	b	a	b	a

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

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:



S → ε | `a` | `b` | `a`**S**`a` | `b`**S**`b`

Designing CFGs

Let Σ = {**a**, **b**} and let *L* = {*w* ∈ Σ* | *w* is a palindrome }

We can design a CFG for *L* by thinking recursively:

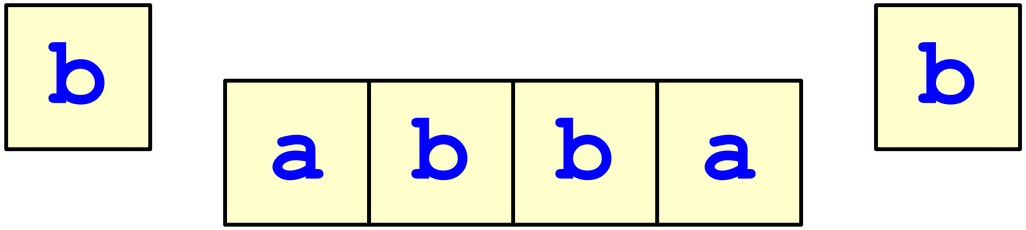
b	**a**	**b**	**b**	**a**	**b**

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:



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

Designing CFGs

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w \text{ is a palindrome } \}$

We can design a CFG for L by thinking recursively:

a	b	b	a

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

Designing CFGs

Let Σ = {**a**, **b**} and let L = {w ∈ Σ* | 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 Σ = {**{**, **}**} and let *L* = {*w* ∈ Σ* | *w* is a string of balanced braces }

Some sample strings in *L*:

{{{}}}

{{}}{}

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

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

ε

{}{}

Designing CFGs

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

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 Σ = {{, }} and let L = {w ∈ Σ* | 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.

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

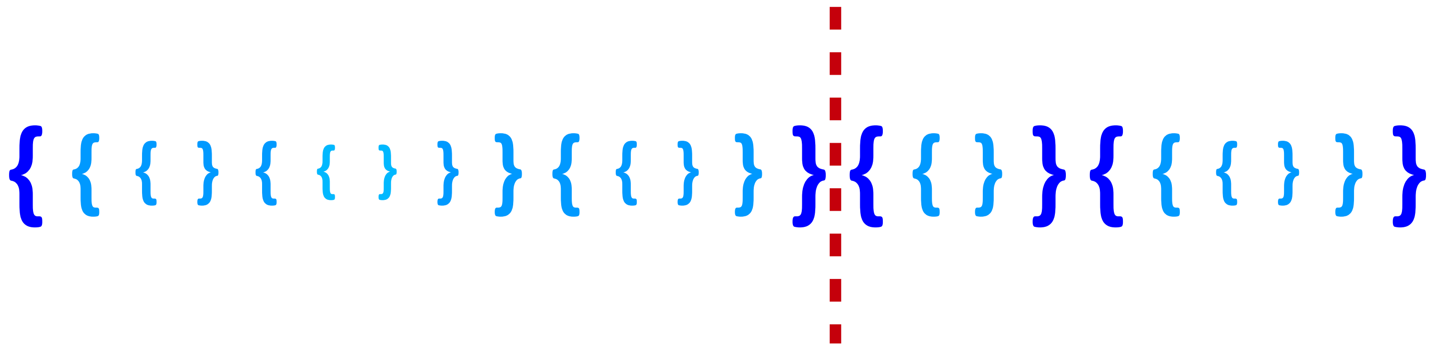
Designing CFGs

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.



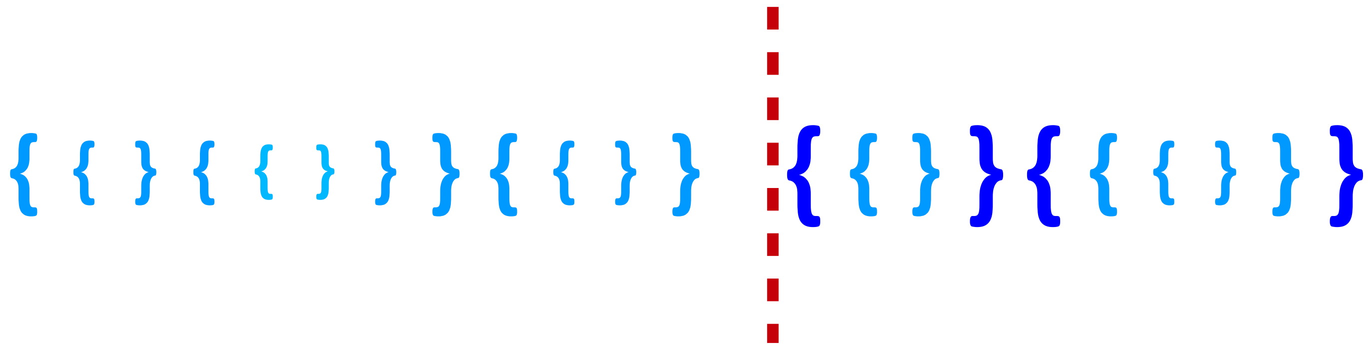
Designing CFGs

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. Removing the first brace and the matching brace forms two new strings of balanced braces.

S → {S}S | ε

Designing CFGs

Here’s the derivation from class today:

S

⇒ **{S}S**

⇒ **{{S}S}S**

⇒ **{{{S}S}S}S**

⇒ **{{{S}{S}S}S}S**

⇒ **{{{ε}{S}S}S}S**

⇒ **{{{ε}{ε}S}S}S**

⇒ **{{{ε}{ε}ε}S}S**

⇒ **{{{ε}{ε}ε}ε}S**

⇒ **{{{ε}{ε}ε}ε}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 Σ = {**a**, **b**} and let L = {w ∈ Σ* | 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 Σ = {**a**, **b**} and let L = {w ∈ Σ* | 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 Σ = {**a**, **b**} and let L = {w ∈ Σ* | 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 5.

CFG Caveats II

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

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

What strings in {**a**, **b**}* can you derive?

Answer: ***None!***

What is the language of the grammar?

Answer: **Ø**

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

Is the following a CFG for L?

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

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

$\mathbf{S} \Rightarrow \mathbf{X} \stackrel{?}{=} \mathbf{X}$

$\Rightarrow \mathtt{a}\mathbf{X} \stackrel{?}{=} \mathbf{X}$

$\Rightarrow \mathtt{aa}\mathbf{X} \stackrel{?}{=} \mathbf{X}$

$\Rightarrow \mathtt{aa} \stackrel{?}{=} \mathbf{X}$

$\Rightarrow \mathtt{aa} \stackrel{?}{=} \mathtt{a}\mathbf{X}$

$\Rightarrow \mathtt{aa} \stackrel{?}{=} \mathtt{a}$

Finding a Build Order

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

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:

$$\mathbf{S} \to \overset{?}{=} \mid \mathbf{aSa}$$

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

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 $\in L$

abb $\in L$

bab $\in L$

aababa $\in L$

bbbbbb $\in L$

a $\notin L$

b $\notin L$

ababab $\notin L$

aabaaaaa $\notin L$

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 bb $\in L$

b ab $\in L$

aa baba $\in L$

bb bbbb $\in L$

a $\notin L$

b $\notin L$

ab abab $\notin L$

aab aaaaaa $\notin L$

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

abb

aaabab

aababa

aaaaaaaaa

bab

bbb

bbabbb

bbbaaaaaa

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

abb

aaabab

aababa

aaaaaaaaa

bab

bbb

bbabbb

bbbaaaaaa

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

abb

aaabab

aababa

aaaaaaaaa

bab

bbb

bbabbb

bbbaaaaaa

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 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

bab

bbb

bbabbb

bbbaaaaaa

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

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.

A → **aAXX** | **ε**

X → **a** | **b**

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	
aababa	
aaaaaaaaa	**bbbbbabaa**

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

A → **aAXX** | **ε**

X → **a** | **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	**bab**
abb	**bbb**
aaabab	**bbabbb**
aababa	**bbbaaaaaa**
aaaaaaaaa	**bbbbbabaa**
A → **aAXX** | **ε**	**X** → **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

abb

aaabab

aababa

aaaaaaaaa

bab

bbb

bbabbb

bbbaaaaa

bbbbbabaa

B → **bBXX** | **ε**

X → **a** | **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 → **aAXX** | **ε**

B → **bBXX** | **ε**

X → **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 $\}$.

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

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 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

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

Storing Information in Nonterminals

Let Σ = {**a**, **b**} and let L = {w ∈ Σ* | $|w| \equiv_3 0$ 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

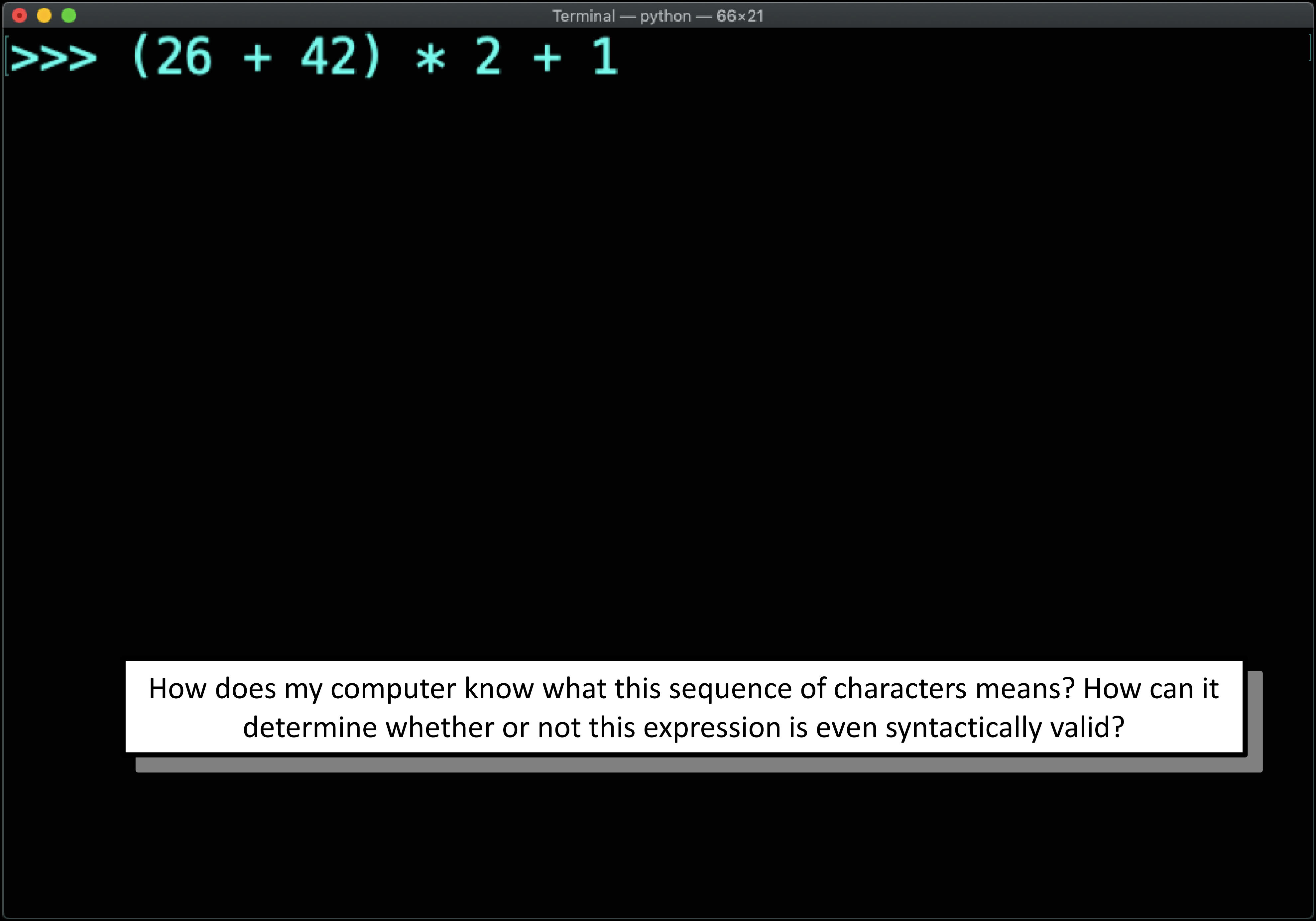
Look for recursive structures where they exist: they can help guide you toward a solution.

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



```
>>> (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

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)`

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

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 would you program it?

Thought for the Weekend:

Being right is not enough