Regular Expressions

CSE 311 Summer 2025
Lecture 18

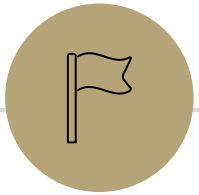
Announcements

- The HW6 due date was pushed back to this Friday, 8/8
  - **The late due date is still Saturday, 8/9, so you can use at most one late day for HW6**

- HW7 will be released on **Friday** and will be due next Friday, 8/15

- HW3, HW4, and midterm solutions are at the front of the class

- Fill out the midterm retake scheduling form that's posted on Ed by this Thursday (8/7) EOD. You will not be able to take the midterm retake exam if you don't fill out the form.



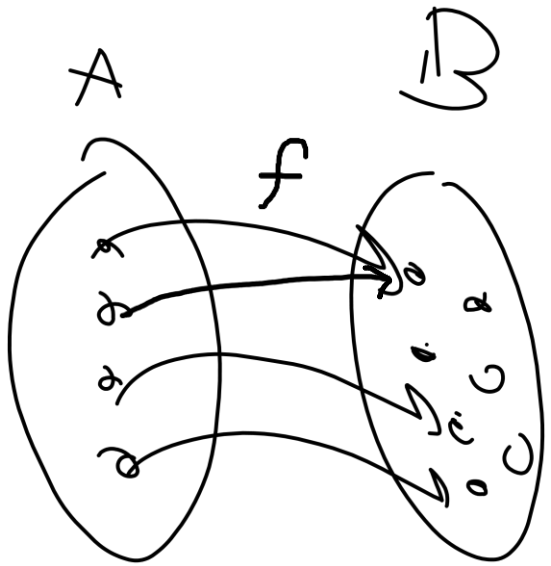
Review

Functions!

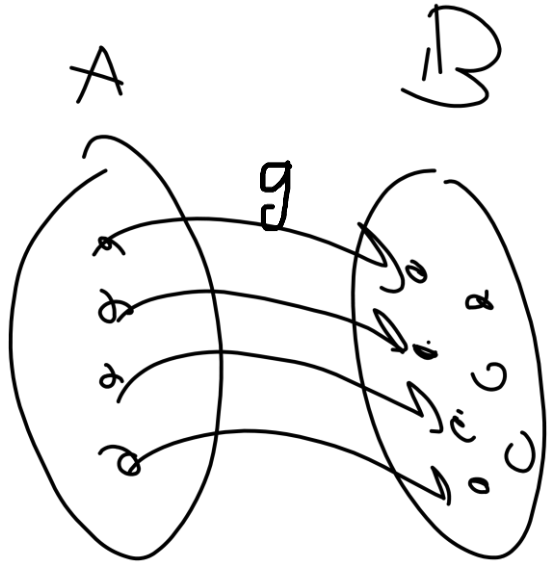
A function $f: A \to B$ maps every element of A to one element of B

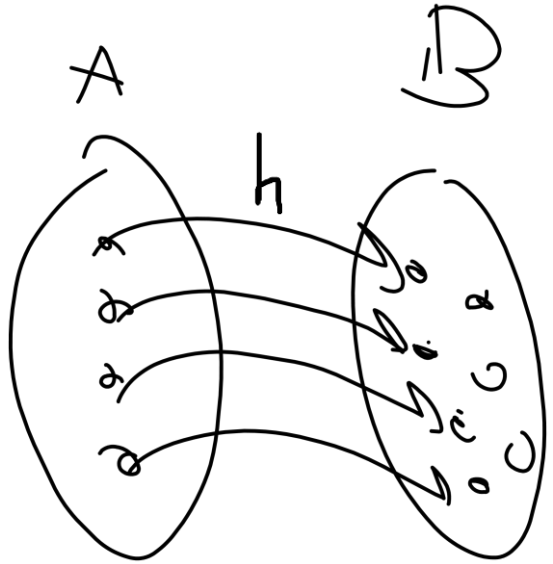
A is the "domain", B is the "co-domain" (also called the "image" or "range")

Good function



Not a function





Bijection

One-to-one (aka injection)

A function f is one-to-one iff
$\forall a \forall b (f(a) = f(b) \rightarrow a = b)$

Onto (aka surjection)

A function $f: A \rightarrow B$ is onto iff
$\forall b \in B \exists a \in A (b = f(a))$

Bijection

A function $f: A \rightarrow B$ is a bijection iff
f is one-to-one and onto

A bijection maps every element of the domain to **exactly** one element of the co-domain, and every element of the codomain to **exactly** one element of the domain.

They're all the same size.

$\mathbb{Z}$ and even integers?

$f(x) = 2x$ Is it a bijection?

One-to-one? Let $a, b \in \mathbb{Z}$ be arbitrary. Suppose $f(a) = f(b)$. By definition of $f, 2a = 2b$. Dividing by 2, $a = b$.

Onto? Let b be an arbitrary even integer. Since b is even, there must be some a $\in \mathbb{Z}$ such that $b = 2a$. By definition of f, $f(a) = b$.

Definition

Two sets A, B have the same size (same cardinality) if and only if there is a bijection $f: A \to B$

This matches our intuition on finite sets.

But it also works for infinite sets!

Let's see just how infinite these sets are.

Countable

Countable

The set A is countable iff there's a one-to-one function from A to $\mathbb{N}$, Equivalently, A is countable iff it is finite or there is a bijection from A to $\mathbb{N}$

$\mathbb{N}, \mathbb{Z}, \{x: x \text{ is an even integer}\}$ are all countable.

To build a bijection from A to $\mathbb{N}$, just list all the elements!

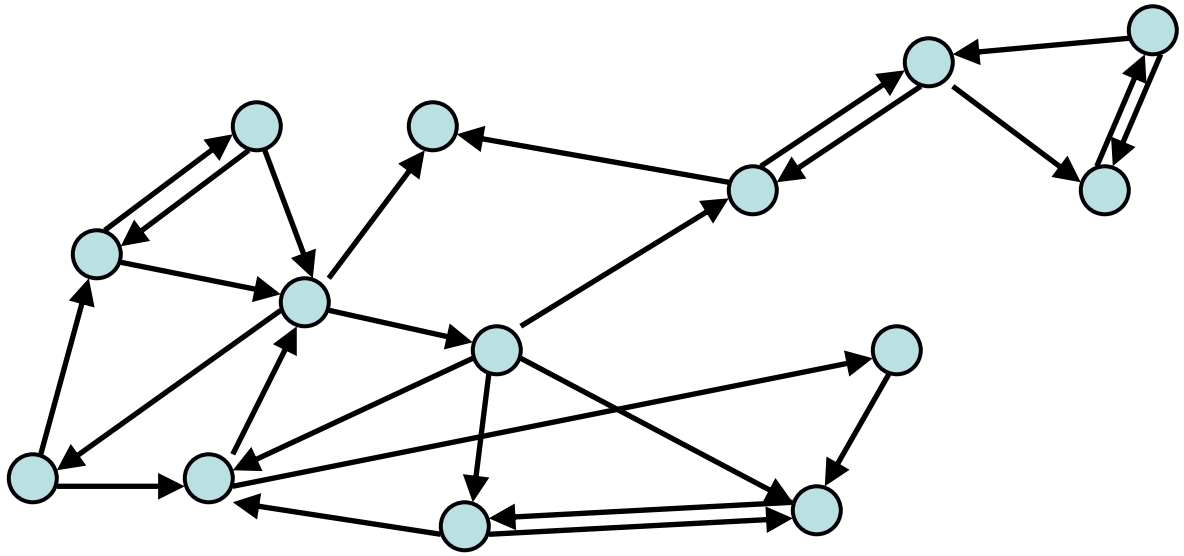
Directed Graphs

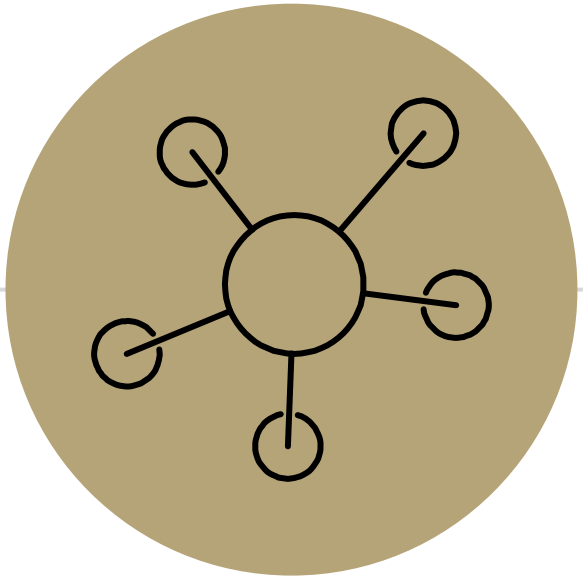
$G = (V, E)$

V is a set of vertices (an underlying set of elements)

E is a set of edges (ordered pairs of vertices; i.e. connections from one to the next).

Path $v_0, v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$
Simple Path: path with all v_i distinct
Cycle: path with $v_0 = v_k$ (and $k > 0$)
Simple Cycle: simple path plus edge (v_k, v_0) with $k > 0$





Part 3 of the course!

Course Outline

Symbolic Logic (training wheels)

Just make arguments in mechanical ways.

Proof Strategies/Set Theory/Number Theory (bike in your backyard)

Models of computation (biking in your neighborhood)

Still make and communicate rigorous arguments

But now with objects you haven't used before.

- A first taste of how we can argue rigorously about computers.

First: regular expressions and context free grammars – understand these "simpler computers"

Soon: what these simple computers can do

Then: what simple computers can't do.

Last week: A problem our computers cannot solve.

The next two weeks

What are we learning?

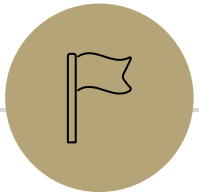
Today: some theory that's useful for computer scientists

The topics for this lecture are key parts of building compilers, we'll use that as motivation—no details on compilers, but hopefully enough that you'll find it interesting.

They ALSO can be thought of as "underpowered computers."

We'll use them to build up to proving what computers can and can't do.

Fri/Mon: Tiny computers! (what can they do/what can't they do?)



Regular Expressions

Regular Expressions

I have a giant text document. And I want to find all the email addresses inside. What does an email address look like?

[some letters and numbers] @ [more letters] . [com, net, or edu]

We want to ctrl-f for a **pattern of strings** rather than a single string

Languages

A set of strings is called a language.

Σ^* is a language

“the set of all binary strings of even length” is a language.

“the set of all palindromes” is a language.

“the set of all English words” is a language.

“the set of all strings matching a given **pattern**” is a language.

Regular Expressions

Basis:

ε is a regular expression. The empty string itself matches the pattern (and nothing else does).

$\emptyset$ is a regular expression. No strings match this pattern.

a is a regular expression, for any $a \in \Sigma$ (i.e. any character). The character itself matching this pattern.

Recursive

If A, B are regular expressions then $(A \cup B)$ is a regular expression
matched by any string that matches A or that matches B [or both]).

If A, B are regular expressions then AB is a regular expression.
matched by any string x such that $x = yz$, y matches A and z matches B.

If A is a regular expression, then A^* is a regular expression.
matched by any string that can be divided into 0 or more strings that match A.

Regular Expressions

$(a \cup bc)$

$0(0 \cup 1)1$

0^*

$(0 \cup 1)^*$

Regular Expressions

$(a \cup bc)$

Corresponds to $\{a, bc\}$

$0(0 \cup 1)1$

Corresponds to $\{001, 011\}$

all length three strings that start with a 0 and end in a 1.

0^*

Corresponds to $\{\varepsilon, 0{,}00{,}000{,}0000, \ldots\}$

$(0 \cup 1)^*$

Corresponds to the set of all binary strings.

More Examples

$(0^*1^*)^*$

0^*1^*

$(0 \cup 1)^*(00 \cup 11)^*(0 \cup 1)^*$

$(00 \cup 11)^*$

More Examples

$(0^*1^*)^*$

All binary strings

0^*1^*

All binary strings with any 0's coming before all 1's

$(0 \cup 1)^*(00 \cup 11)^*(0 \cup 1)^*$

This is all binary strings again. Not a "good" representation, but valid.

$(00 \cup 11)^*$

All binary strings where 0s and 1s come in pairs

More Practice

You can also go the other way

Write a regular expression for “the set of all binary strings of odd length”

Write a regular expression for “the set of all binary strings with at most two ones”

Write a regular expression for “strings that don’t contain 00”

More Practice

You can also go the other way

Write a regular expression for “the set of all binary strings of odd length”

$(0 \cup 1)(00 \cup 01 \cup 10 \cup 11)^*$

Write a regular expression for “the set of all binary strings with at most two ones”

$0^*(1 \cup \epsilon)0^*(1 \cup \epsilon)0^*$

Write a regular expression for “strings that don’t contain 00”

$(01 \cup 1)^*(0 \cup \epsilon)$ (key idea: all 0s followed by 1 or end of the string)

Practical Advice

Check ε and 1 character strings to make sure they're excluded or included (easy to miss those edge cases).

If you can break into pieces, that usually helps.

"nots" are hard (there's no "not" in standard regular expressions)

But you can negate things, usually by negating at a low-level. E.g. to have binary strings without 00, your building blocks are 1's and 0's followed by a 1

$(01 \cup 1)^*(0 \cup \varepsilon)$ then make adjustments for edge cases (like ending in 0)

Remember $*$ allows for 0 copies! To say "at least one copy" use AA^*.

Regular Expressions In Practice

EXTREMELY useful. Used to define valid “tokens” (like legal variable names or all known keywords when writing compilers/languages)

Used in `grep` to actually search through documents.

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

^	start of string	
$	end of string	
[01]	a 0 or a 1	
[0-9]	any single digit	
\.	period **\,** comma **\-** minus	
.	any single character	
ab	a followed by b	(**AB**)
(a**|**b**)**	a or b	(**A** ∪ **B**)
a**?**	zero or one of a	(**A** ∪ **ε**)
a*****	zero or more of a	**A***
a**+**	one or more of a	**AA***

e.g. **^[\-+]?[0-9]*(\.|\,)?[0-9]+$**

General form of decimal number e.g. 9.12 or -9,8 (Europe)

What **can't** regular expressions do?

Can you write a regular expression for all strings of the form $0^k 1^k$?
i.e., same number of 0's and 1's, all 0's coming first

Can you write a regular expression for $\{0^k 1^k : k \leq 100\}$
i.e., same restrictions as above, but also only at most 200 characters total.

What **can't** regular expressions do?

Can you write a regular expression for all strings of the form $0^k 1^k$?
i.e., same number of 0's and 1's, all 0's coming first

No! There is no such regular expression (we'll prove it in a few weeks).

Can you write a regular expression for $\{0^k 1^k : k \leq 100\}$
i.e., same restrictions as above, but also only at most 200 characters total.

Yes! It'll probably take you a while to write it though…
…there are a finite number of strings satisfying this description, just list them all and ∪ them together.

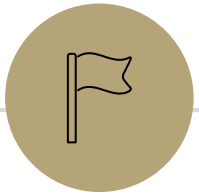
A Vocabulary Note

Not everything can be represented as a regular expression.
E.g. “the set of all palindromes” is not the language of any regular expression.

Some programming languages define features in their “regexes” that can’t be represented by our definition of regular expressions.
Things like “match this pattern, then have exactly that **substring** appear later.

So before you say “ah, you can’t do that with regular expressions, I learned it in 311!” you should make sure you know whether your language is calling a more powerful object “regular expressions”.

But the more “fancy features” beyond regular expressions you use, the slower the checking algorithms run, (and the harder it is to force the expressions to fit into the framework) so this is still very useful theory.



More Practice: Regular Expressions

More Practice

All binary strings with a 1 in the third position. (index from 1)

All binary strings with a 1 in the third position from the end (with length at least three).

All binary strings with an even number of 0s or exactly one 1.

More Practice

All binary strings with a 1 in the third position. (index from 1)

$$(0 \cup 1)(0 \cup 1)1(0 \cup 1)^*$$

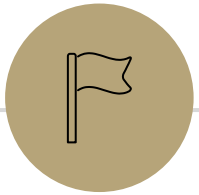
All binary strings with a 1 in the third position from the end (with length at least three).

$$(0 \cup 1)^*1(0 \cup 1)(0 \cup 1)$$

All binary strings with an even number of 0s or exactly one 1.

$$(1^*01^*01^*)^* \cup (0^*10^*)$$

Sometimes you can write two regular expressions and just ∪ together.



Context Free Grammars

Another way of defining a language

What Can't Regular Expressions Do?

Some easy things
Things where you could say whether a string matches with just a loop
$\{0^k 1^k : k \geq 0\}$
The set of all palindromes.

And some harder things
Expressions with matched parentheses
Properly formed arithmetic expressions

Context Free Grammars can solve all of these problems!

Context Free Grammars

A context free grammar (CFG) is a finite set of production rules over:

An alphabet Σ of "terminal symbols"

A finite set V of "nonterminal symbols"

A start symbol (one of the elements of V) usually denoted S.

A production rule for a nonterminal $A \in V$ takes the form

$A \to w_1 | w_2 | \cdots | w_k$

Where each $w_i \in (V \cup \Sigma)^*$ is a string of nonterminals and terminals.

Context Free Grammars

We think of context free grammars as **generating** strings.

1. Start from the start symbol S.

2. Choose a nonterminal in the string, and a production rule $A \rightarrow w_1|w_2|\ldots|w_k$ replace that copy of the nonterminal with w_i.

3. If no nonterminals remain, you're done! Otherwise, goto step 2.

A string is in the language of the CFG iff it can be generated starting from S.

Examples

$S \rightarrow 0S0|1S1|0|1|\varepsilon$

$S \rightarrow 0S|S1|\varepsilon$

$S \rightarrow (S)|SS|\varepsilon$

The alphabet here is {(,)} i.e. parentheses are the characters.

$S \rightarrow AB$

$A \rightarrow 0A1|\varepsilon$

$B \rightarrow 1B0|\varepsilon$

Examples

$S \to 0S0|1S1|0|1|\varepsilon$

The set of all binary palindromes

$S \to 0S|S1|\varepsilon$

The set of all strings with any 0's coming before any 1's (i.e. 0^*1^*)

$S \to (S)|SS|\varepsilon$

Balanced parentheses

$S \to AB$

$A \to 0A1|\varepsilon$

$B \to 1B0|\varepsilon$ $\quad \{0^j 1^{j+k} 0^k : j, k \geq 0\}$

Arithmetic

$E \to E + E | E * E | (E) | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

Generate $(2 * x) + y$

Generate $2 + 3 * 4$ in two different ways

Arithmetic

$E \to E + E | E * E | (E) | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

Generate $(2 * x) + y$

$E \Rightarrow E + E \Rightarrow (E) + E \Rightarrow (E * E) + E \Rightarrow (2 * E) + E \Rightarrow (2 * x) + E \Rightarrow (2 * x) + y$

Generate $2 + 3 * 4$in two different ways

$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow 2 + E * E \Rightarrow 2 + 3 * E \Rightarrow 2 + 3 * 4$

$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow 2 + E * E \Rightarrow 2 + 3 * E \Rightarrow 2 + 3 * 4$

Multiple ways of generating strings

Generate $2 + 3 * 4$in two different ways

$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow 2 + E * E \Rightarrow 2 + 3 * E \Rightarrow 2 + 3 * 4$

$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow 2 + E * E \Rightarrow 2 + 3 * E \Rightarrow 2 + 3 * 4$

What did we mean by these being different? They represent different meanings mathematically.

One says “you’re adding together two numbers: 2 and (whatever 3*4 is)”

The other says “you’re multiplying two numbers: (whatever 2+3 is) and 4”

Those have different meanings!

Parse Trees—remember where parentheses go

Suppose a context free grammar G generates a string x

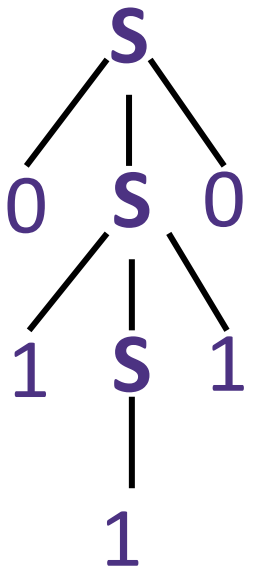
A parse tree of x for G has

Rooted at S (start symbol)

Children of every A node are labeled with the characters of w for some $A \to w$

Reading the leaves from left to right gives x.

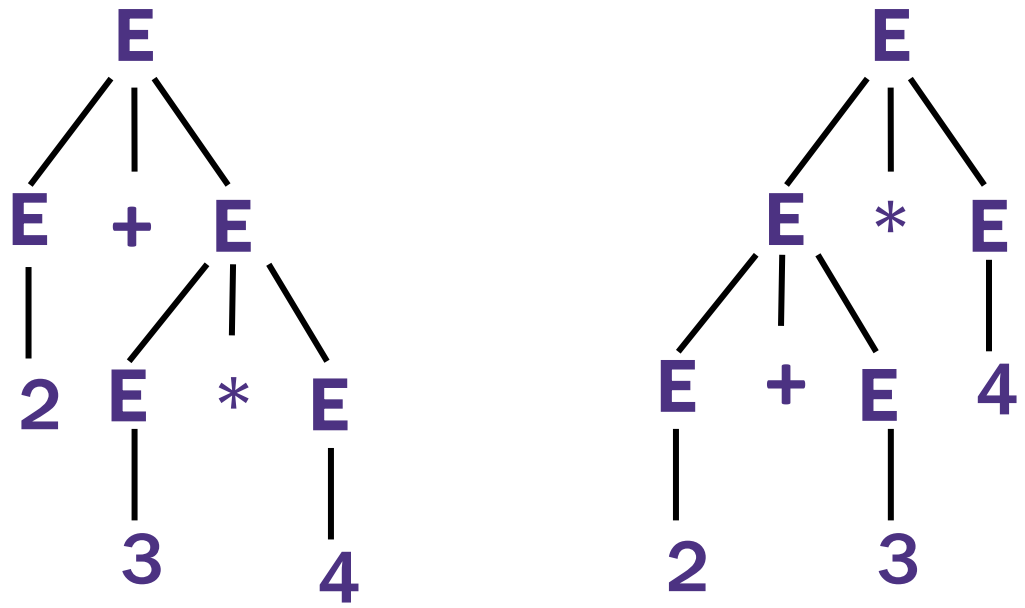
$S \to 0S0|1S1|0|1|\varepsilon$



Back to the arithmetic

$E \to E + E | E * E | (E) | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

Two parse trees for $2 + 3 * 4$



Why do we care about parsing?

$2 + 3 * 4$ can only mean one thing!

If I write these symbols in a program, we need to make sure we know which one to do.

The first grammar we saw was “ambiguous” it allows the same string to “mean” two different things.

Sometimes you can fix that!

How do we encode order of operations

If we want to keep “in order” we want there to be only one possible parse tree.

Differentiate between “things to add” and “things to multiply”

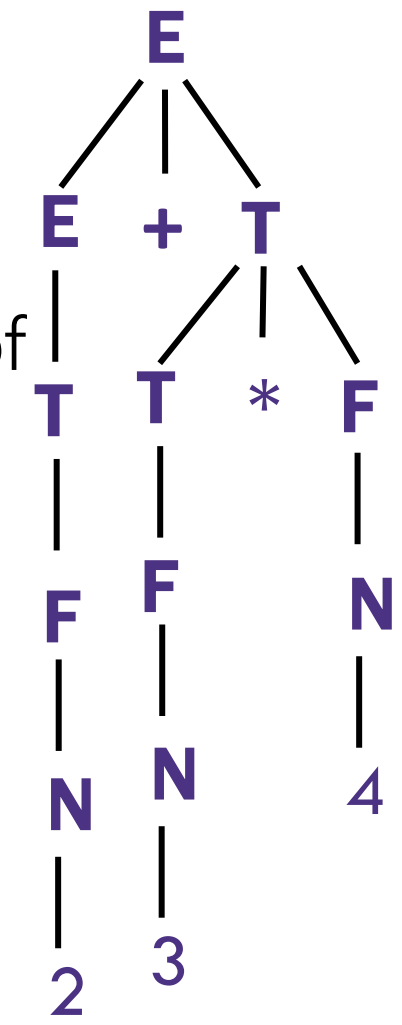
Only introduce a * sign after you’ve eliminated the possibility of introducing another + sign in that area.

$E \rightarrow T|E + T$

$T \rightarrow F|T * F$

$F \rightarrow (E)|N$

$N \rightarrow x|y|z|0|1|2|3|4|5|6|7|8|9$



How do Computer Scientists use CFGs?

Most programming languages define valid programs as "strings that fit a CFG" That makes sure Java breaks down math expressions correctly! And also code like this:

```
if(i>0)
   if(j>0)
      System.out.println("hi");
   else
      System.out.println("bye");
```

```
if(i>0)
   if(j>0)
      System.out.println("hi");
else
   System.out.println("bye");
```

The else could be attached to either "if"! Java needs a rule to decide which it goes with. Java's convention makes the one on the left the intuitive whitespace.
(You as a programmer should put braces so the humans reading your code don't have to wonder!)

CFGs in practice

Used to define programming languages.

Often written in Backus-Naur Form – just different notation

Variables are <names-in-brackets> (or sometimes without)

like <if-then-else-statement>, <condition>, <identifier>

→ is replaced with ::= or :

BNF for C (no <...> and uses : instead of ::=)

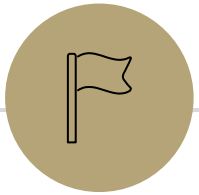
```
statement:
  ((identifier | "case" constant-expression | "default") ":")*
  (expression? ";" |
   block |
   "if" "(" expression ")" statement |
   "if" "(" expression ")" statement "else" statement |
   "switch" "(" expression ")" statement |
   "while" "(" expression ")" statement |
   "do" statement "while" "(" expression ")" ";" |
   "for" "(" expression? ";" expression? ";" expression? ")" statement |
   "goto" identifier ";" |
   "continue" ";" |
   "break" ";" |
   "return" expression? ";"
  )

block: "{" declaration* statement* "}"

expression:
  assignment-expression%

assignment-expression: (
    unary-expression (
      "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | "&=" |
      "^=" | "|="
    )
  )* conditional-expression

conditional-expression:
  logical-OR-expression ( "?" expression ":" conditional-expression )?
```

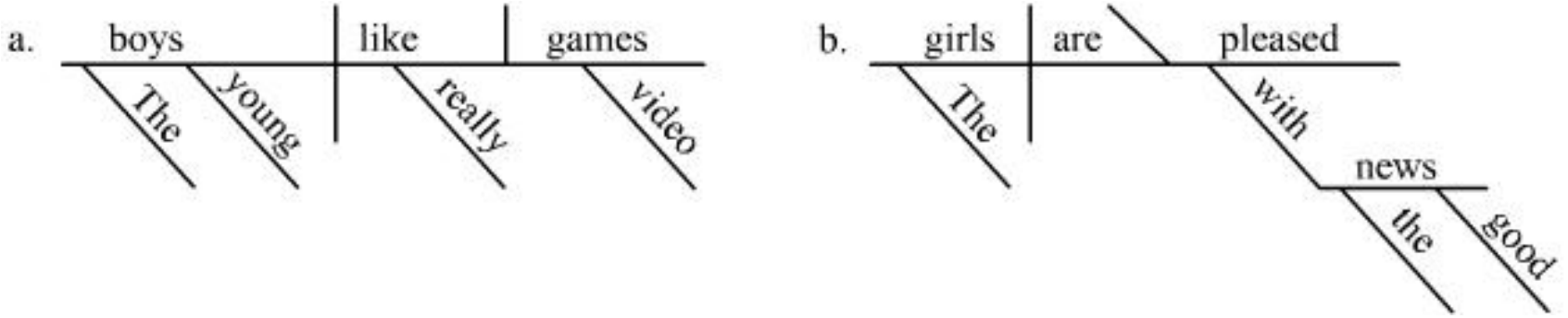


Some fun CFG applications

If we have time

Parse Trees

Remember diagramming sentences in middle school?



<sentence>::=<noun phrase><verb phrase>

<noun phrase>::=<determiner><adjective><noun>

<verb phrase>::=<verb><adverb>|<verb><object>

<object>::=<noun phrase>

Parse Trees

<sentence>::=<noun phrase><verb phrase>

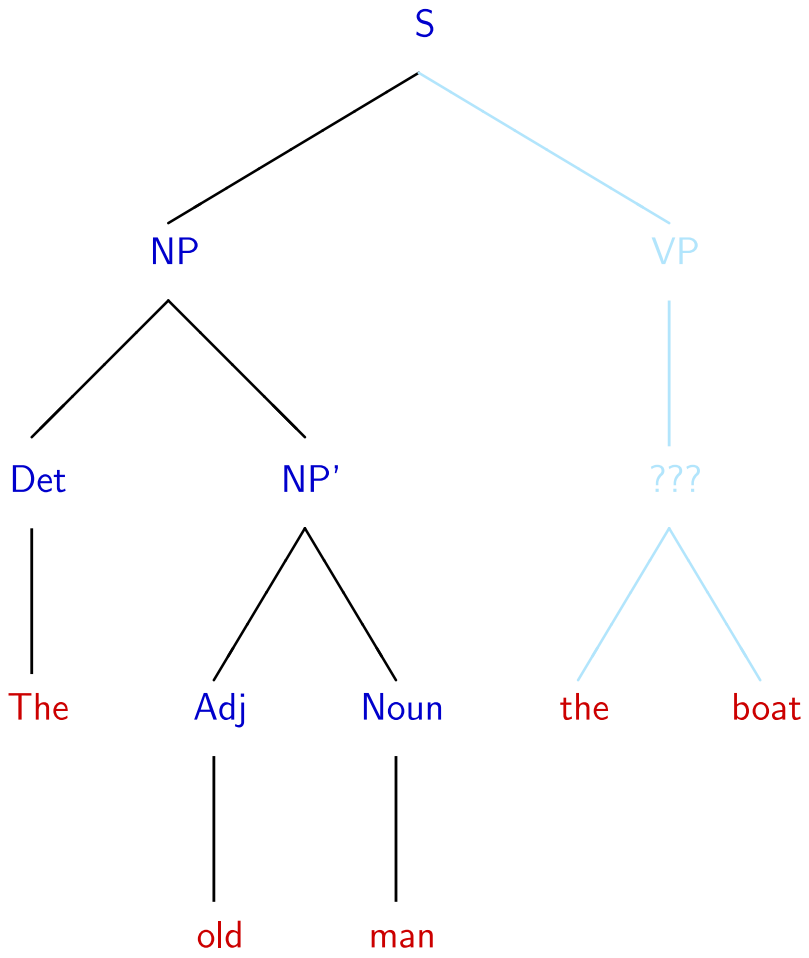
<noun phrase>::=<determiner><adjective><noun>

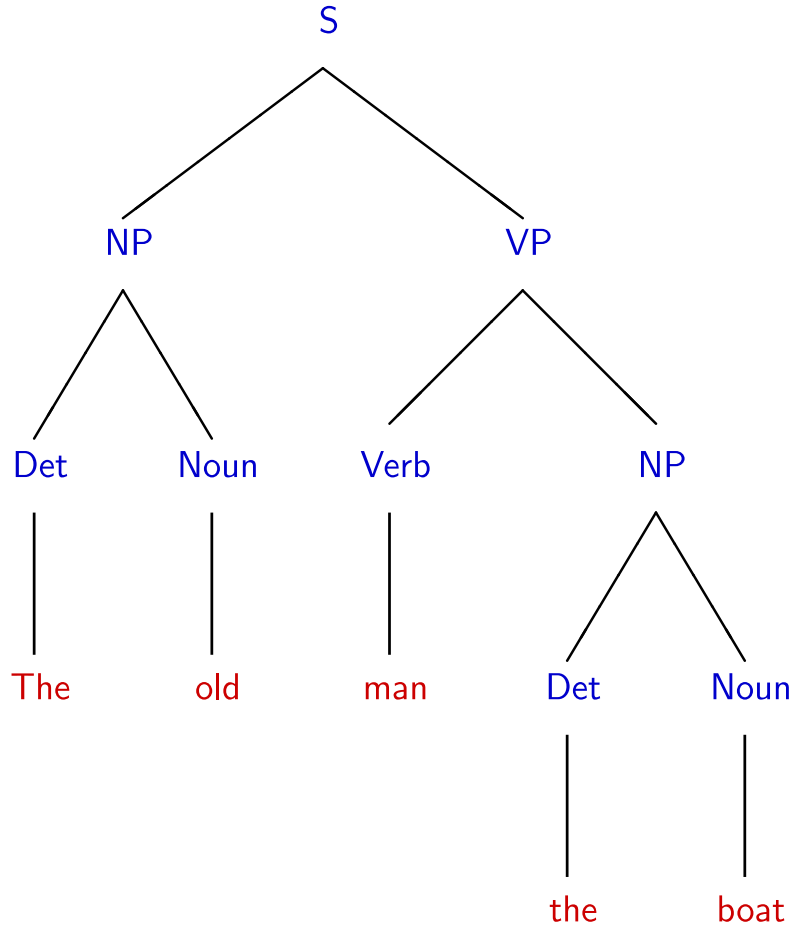
<verb phrase>::=<verb><adverb>|<verb><object>

<object>::=<noun phrase>

The old man the boat.

The old man the boat





English is ambiguous

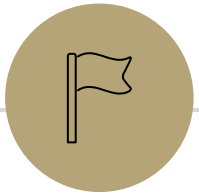
(Most of 'standard') English can be represented as a context free grammar.

It's not perfect.

The grammar is ambiguous! That is, there are sentences which have multiple valid parsings (multiple meanings).

Can you find multiple meanings of this sentence:
"Place these 3 exercise balls on the mat at the top of the hill."

See this video



The Important Takeaways

Power of Context Free Languages

There are languages CFGs can express that regular expressions can't

e.g. palindromes

What about vice versa – is there a language that a regular expression can represent that a CFG can't?

No!

Are there languages even CFGs cannot represent?

Yes!

$\{0^k 1^j 2^k 3^j | j, k \geq 0\}$ cannot be written with a context free grammar.

Takeaways

CFGs and regular expressions gave us ways of succinctly representing sets of strings

- Regular expressions super useful for representing things you need to search for
- CFGs represent complicated languages like “java code with valid syntax”

Next: (mathematical representations of) Tiny computers! And how they relate to regular expressions and CFGs.

Todo

Tonight:

- CC 18 is out and due Friday at noon
- Fill out the midterm retake scheduling form by Thursday EOD
- Start working on HW6 after lecture if you haven't already