Warm up:
How would you search through a document for all email addresses inside?
(e.g. with control-f)

What if it was a document full of tweets?

Regular Expressions (for real this time)

CSE 311 Spring 2022
Lecture 21

Just The Setup

abc = cba

Define $P(n)$ "for all strings, x, of length n, len(x^R)=len(x)."

Write a strong inductive proof (not a structural one yet).

What's the first sentence of your inductive step?

BC $P(0)$

Just The Setup

Define $P(n)$ "for all strings, x, of length n, len(x^R)=len(x)."

BC: Let x be an arbitrary string of length $\mathbf{0}$. ….

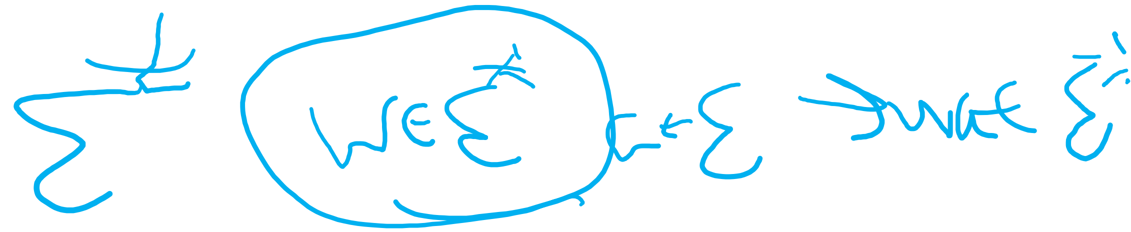
…

IH: Suppose $P(0), P(1), \ldots P(k)$ for an arbitrary $k \geq 0$.

IS: Let x be an arbitrary string of length $k + 1$.

…

Just The Setup



Define $P(n)$ "for all strings, x, of length n, len(x^R)=len(x)."

BC: Let x be an arbitrary string of length $\mathbf{0}$. The only string of length $\mathbf{0}$ is ε (since applying the recursive rule only increases the length).

ε^R is, by definition, ε. So len(ε^R)=len(ε).

IH: Suppose $P(0), P(1), \ldots P(k)$ for an arbitrary $k \geq \mathbf{0}$.

IS: Let x be an arbitrary string of length $k+1$. Since $k \geq 0$, $k+1 \geq 1$.

Since the length of x is more than $\mathbf{0}$, it must have been built with a recursive rule. So $x = wa$ for some string w and character a. From the definition of len, len(w)=k. By IH len(w^R)=len(w).

The full IS

IH: Suppose $P(0), P(1), \dots P(k)$ for an arbitrary $k \geq 0$.

IS: Let x be an arbitrary string of length $k+1$. Since $k \geq 0$, $k+1 \geq 1$.

Since the length of x is more than 0, it must have been built with a recursive rule. So $x = wa$ for some string w and character a. From the definition of len, len(w)=k. By IH len(w^R)=len(w).

len(x^R)=len($[wa]^R$)=len(aw^R)=len(a)+len(w^R), where the last step is treating a as a string and using len($x \cdot y$)=len(x)+len(y) from last lecture.

Applying IH, we have
len(x^R)=…=len(a)+len(w)=1+len(w)=len(wa)=len(x).

What if we did structural induction?

If we used structural induction instead, what would we do?

Our base case would have been ε.

Our IH would have supposed $P(w)$

Our IS would have shown $P(wa)$.

That's…exactly what we did for that strong inductive proof!

There's an optional reading where we say a little more on why structural weak and strong induction are "all the same."

CAUTION

Structural induction *looks like* we're violating the rule of "introduce an arbitrary variable to prove a for-all statement"

We're not!

What structural induction really says is "consider an arbitrary element of the recursively-defined set. By the exclusion rule, it's either a basis element, or made from other elements by a rule" and then break into all possible cases.

Only when we have an explicit recursive definition of a set can we do this. You should not be "building up" elements in inductive steps.

If you don't have a recursively-defined set

You won't do structural induction.

You can do weak or strong induction though.

For example, Let $P(n)$ be "for all elements of S of "size" n <something> is true"

To prove "for all $x \in S$ of size n…" you need to start with "let x be an arbitrary element of size $k+1$ in your IS.

You CAN'T start with size k and "build up" to an arbitrary element of size $k+1$ it isn't arbitrary (we only *seem* to do that with structural induction, but we're using the exclusion rule there).

Induction: Hats!

You have n people in a line ($n \geq 2$). Each of them wears either a purple hat or a gold hat. The person at the front of the line wears a purple hat. The person at the back of the line wears a gold hat.

Show that for every arrangement of the line satisfying the rule above, there is a person with a purple hat next to someone with a gold hat.

Yes this is kinda obvious. I promise this is good induction practice.

Yes you could argue this by contradiction. I promise this is good induction practice.

Induction: Hats!

Define $P(n)$ to be “in every line of n people with gold and purple hats, with a purple hat at one end and a gold hat at the other, there is a person with a purple hat next to someone with a gold hat”

We show $P(n)$ for all integers $n \geq 2$ by induction on n.

Base Case: $n = 2$

Inductive Hypothesis:

Inductive Step:

By the principle of induction, we have $P(n)$ for all $n \geq 2$

Induction: Hats!

Define $P(n)$ to be "in every line of n people with gold and purple hats, with a purple hat at one end and a gold hat at the other, there is a person with a purple hat next to someone with a gold hat"

We show $P(n)$ for all integers $n \geq 2$ by induction on n.

Base Case: $n = 2$ The line must be just a person with a purple hat and a person with a gold hat, who are next to each other.

Inductive Hypothesis: Suppose $P(k)$ holds for an arbitrary $k \geq 2$.

Inductive Step: Consider an arbitrary line with $k + 1$ people in purple and gold hats, with a gold hat at one end and a purple hat at the other.

Target: there is someone in a purple hat next to someone in a gold hat.

By the principle of induction, we have $P(n)$ for all $n \geq 2$

Induction: Hats!

Define $P(n)$ to be "in every line of n people with gold and purple hats, with a purple hat at one end and a gold hat at the other, there is a person with a purple hat next to someone with a gold hat"

We show $P(n)$ for all integers $n \geq 2$ by induction on n.

Base Case: $n = 2$ The line must be just a person with a purple hat and a person with a gold hat, who are next to each other.

Inductive Hypothesis: Suppose $P(k)$ holds for an arbitrary $k \geq 2$.

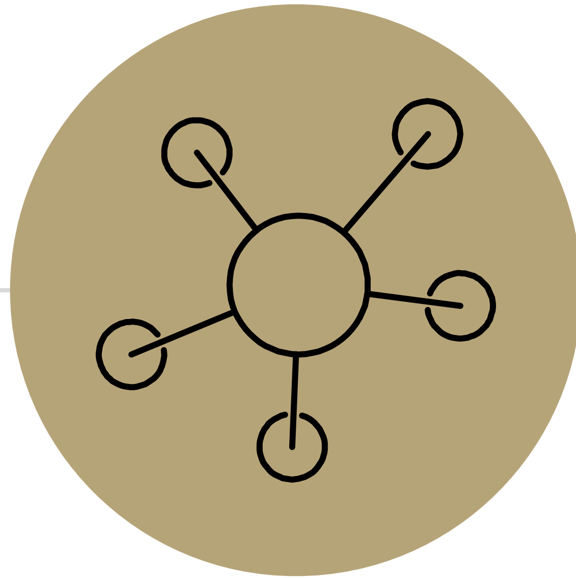
Inductive Step: Consider an arbitrary line with $k + 1$ people in purple and gold hats, with a gold hat at one end and a purple hat at the other.

Case 1: There is someone with a purple hat next to the person in the gold hat at one end. Then those people are the required adjacent opposite hats.

Case 2:. There is a person with a gold hat next to the person in the gold hat at the end. Then the line from the second person to the end is length k, has a gold hat at one end and a purple hat at the other. Applying the inductive hypothesis, there is an adjacent, opposite-hat wearing pair.

In either case we have $P(k + 1)$.

By the principle of induction, we have $P(n)$ for all $n \geq 2$



Part 3 of the course!

Course Outline

Symbolic Logic (training wheels; lectures 1-8)

Just make arguments in mechanical ways.

Set Theory/Arithmetic (bike in your backyard; lectures 9-19)

Models of computation (biking in your neighborhood; lectures 19-30)

Still make and communicate rigorous arguments

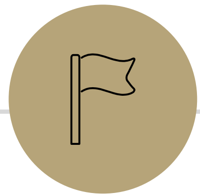
But now with objects you haven't used before.

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

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

What these simple computers can do

Last week of class: what simple computers (and normal ones) can't 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

Every pattern automatically gives you a language .

The set of all strings that match that pattern.

We'll formalize "patterns" via "regular expressions"

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

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

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

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.

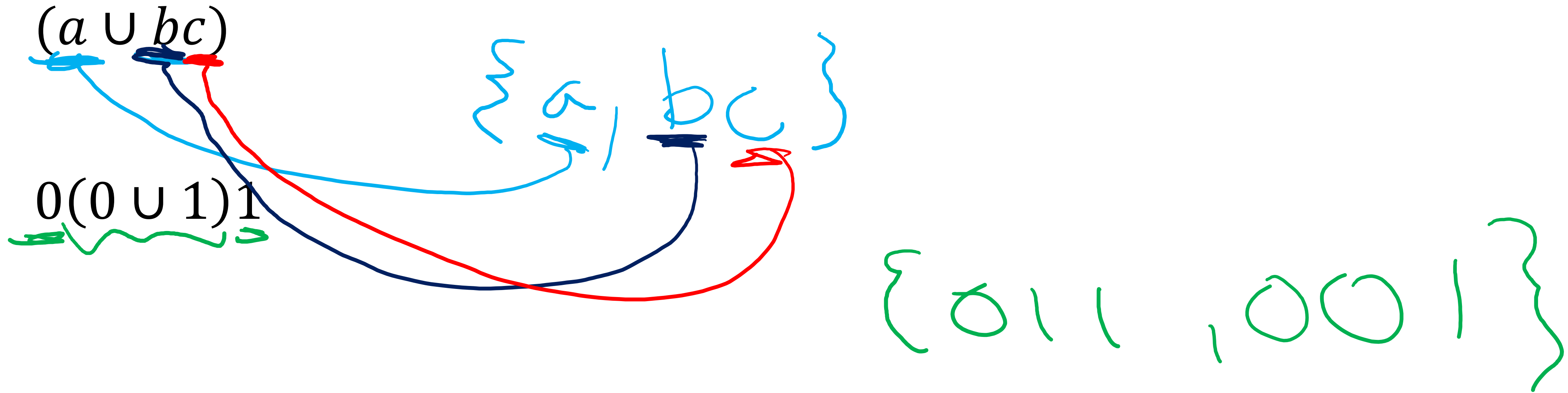
Recursive

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^*

{ε, 0, 00, 000, 0000, ...}

$(0 \cup 1)^*$

all binary strings

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 $\{\varepsilon, 0,00,000,0000, \dots\}$

$(0 \cup 1)^*$

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

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

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”

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

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

ε

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)

Regular Expressions In Practice

When you only have ASCII characters (say in a programming language)

| usually takes the place of ∪

? (and perhaps creative rewriting) take the place of ε.

E.g. $(0 \cup \varepsilon)(1 \cup 10)^*$ is `0?(1|10)*`

A Final Vocabulary Note

Not everything can be represented as a regular expression.

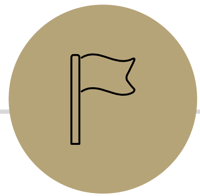
E.g. “the set of all palindromes” is not the language of any regular expression.

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.



Context Free Grammars

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

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

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

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

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

Pollev.com/uwcse311

Arithmetic

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

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

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

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

Parse Trees

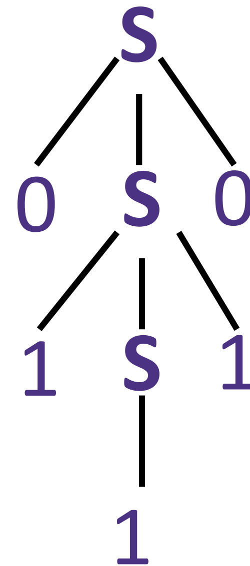
Suppose a context free grammar G generates a string x

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.

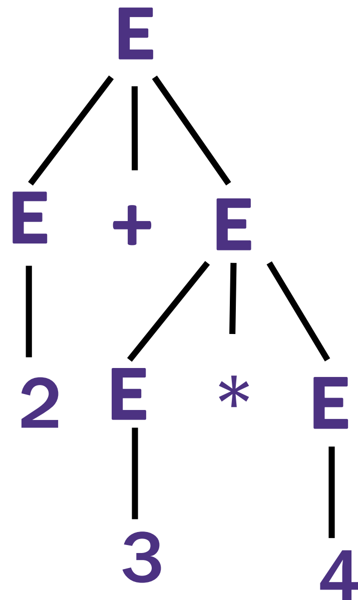


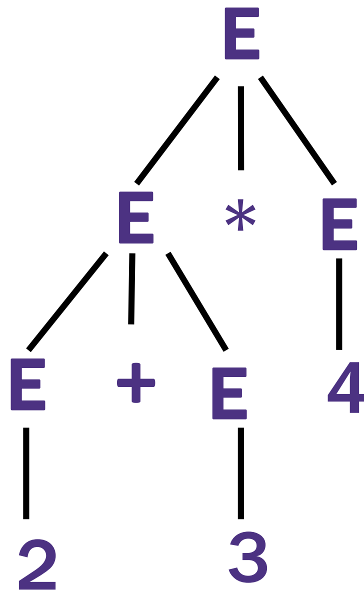
$S \to 0S0|1S1|0|1|\varepsilon$

Back to the arithmetic

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

Two parse trees for $2 + 3 * 4$





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.

$E \rightarrow T | E + T$

$T \rightarrow F | T * F$

$F \rightarrow (E) | N$

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