Warm up:
What is the following recursively-defined set?
Basis Step: $4 \in S$, $5 \in S$
Recursive Step: If $x \in S$ and $y \in S$ then $x - y \in S$

xkcd.com/208

Structural Induction and Regular Expressions

CSE 311 Autumn 20
Lecture 19

Announcements

We’ll release solutions to the (hopefully graded) midterm early next week.

Homework 6 will be released on Monday, due the following Monday.

Strings

Why these recursive definitions?

They're the basis for regular expressions, which we'll introduce next week. Answer questions like "how do you search for anything that looks like an email address"

First, we need to talk about strings.

Σ will be an **alphabet** the set of all the letters you can use in words.

Σ^* is the set of all **words** all the strings you can build off of the letters.

Strings

ε is "the empty string"

The string with 0 characters – `""` in Java (not `null!`)

Σ^*:

Basis: $\varepsilon \in \Sigma^*$.

Recursive: If $w \in \Sigma^*$ and $a \in \Sigma$ then $wa \in \Sigma^*$

wa means the string of w with the character a appended.

You'll also see $w \cdot a$ (a $\cdot$ to mean "concatenate" i.e. + in Java)

Functions on Strings

Since strings are defined recursively, most functions on strings are as well.

Length:

len(ε)=0;

len(wa)=len(w)+1 for $w \in \Sigma^*$, $a \in \Sigma$

Reversal:

$\varepsilon^R = \varepsilon$;
$(wa)^R = aw^R$ for $w \in \Sigma^*$, $a \in \Sigma$

Concatenation

$x \cdot \varepsilon = x$ for all $x \in \Sigma^*$;
$x \cdot (wa) = (x \cdot w)a$ for $w \in \Sigma^*, a \in \Sigma$

Number of c's in a string

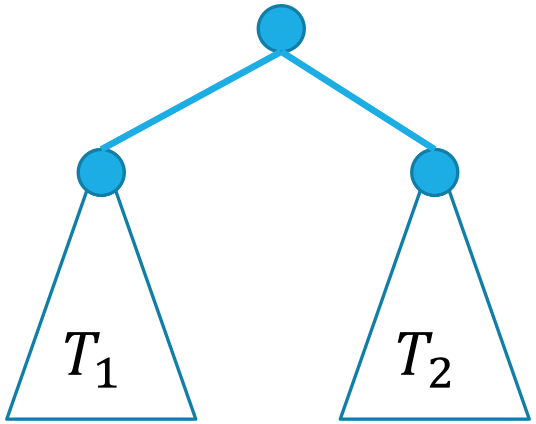
$\#_c(\varepsilon) = 0$
$\#_c(wc) = \#_c(w) + 1$ for $w \in \Sigma^*$;
$\#_c(wa) = \#_c(w)$ for $w \in \Sigma^*, a \in \Sigma \setminus \{c\}$.

More Structural Sets

Binary Trees are another common source of structural induction.

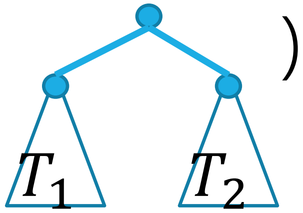
Basis: A single node is a rooted binary tree. 

Recursive Step: If T_1 and T_2 are rooted binary trees with roots r_1 and r_2, then a tree rooted at a new node, with children r_1, r_2 is a binary tree.

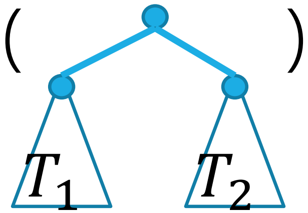


Functions on Binary Trees

size()=1

size() = size(T_1) + size(T_2) + 1

height() = 0

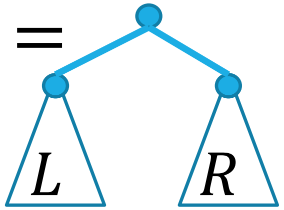
height() = 1+max(height(T_1),height(T_2))

Structural Induction on Binary Trees

Let $P(T)$ be "size(T) $\leq 2^{height(T)+1} - 1$". We show $P(T)$ for all binary trees T by structural induction.

Base Case: Let $T =$ . size(T)=1 and height(T) = 0, so size(T)=1$\leq 2 - 1 = 2^{0+1} - 1 = 2^{height(T)+1} - 1$.

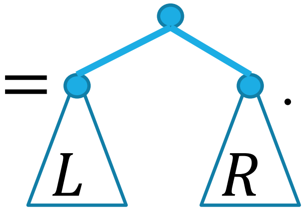
Inductive Hypothesis: Suppose $\mathrm{P}(L)$ and $\mathrm{P}(R)$ for arbitrary binary trees L, R.

Inductive Step: Let $T =$ 

.

Structural Induction on Binary Trees (cont.)

Let $P(T)$ be "size(T) $\leq 2^{height(T)+1} - 1$". We show $P(T)$ for all binary trees T by structural induction.

Inductive Step: Let $T =$  .

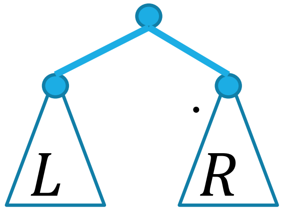
height(T)=$1 + \max\{height(L), height(R)\}$

size(T)= 1 +size(L)+size(R)

So $P(T)$ holds, and we have $P(T)$ for all binary trees T by the principle of induction.

Structural Induction on Binary Trees (cont.)

Let $P(T)$ be "size(T) $\leq 2^{height(T)+1} - 1$". We show $P(T)$ for all binary trees T by structural induction.

Inductive Step: Let $T =$  .

height(T)=$1 + \max\{height(L), height(R)\}$

size(T)= 1 +size(L)+size(R)

size(T)=1+size(L)+size(R) $\leq 1 + 2^{height(L)+1} - 1 + 2^{height(R)+1} - 1$ (by IH)

$\leq 2^{height(L)+1} + 2^{height(R)+1} - 1$ (cancel 1's)

$\leq 2^{height(T)} + 2^{height(T)} - 1 = 2^{height(T)+1} - 1$ (T taller than subtrees)

So $P(T)$ holds, and we have $P(T)$ for all binary trees T by the principle of induction.

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.

Regular expressions and context free grammars – understand these "simpler computers"

Then: 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).

$\emptyset$ 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.

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 mattern.

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

$(0 \cup 1)^*$

Corresponds to the set of all binary strings.

More Examples

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

0^*1^*

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

$(00 \cup 11)^*$

Fill out the poll everywhere for Activity Credit!

Go to pollev.com/cse311 and login with your UW identity

More Examples

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

All binary strings

0^*1^*

All binary strings with any 0's coming before any 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”

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

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

A Remark on languages

We said regular expressions were patterns, we often asked the question "does this string match this pattern?"

We could also ask "what is the set of all strings that match this pattern?

That set is called the "language of the regular expression"

Our next object answers both of these questions, but it's

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

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

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

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