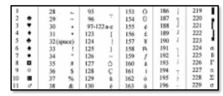
cse 311: foundations of computing

Spring 2015
Lecture 19: Structural induction and regular expressions

review: strings

- An *alphabet* Σ is any finite set of characters.

  e.g. $\Sigma = \{0,1\}$ or $\Sigma = \{A, B, C, \ldots X, Y, Z\}$ or

  $\Sigma =$ 

- The set Σ^* of *strings* over the alphabet Σ is defined by
  - **Basis:** $\varepsilon \in \Sigma^*$ (ε is the empty string)
  - **Recursive:** if $w \in \Sigma^*$, $a \in \Sigma$, then $wa \in \Sigma^*$

function definitions on recursively defined sets

Length:
- len (ε) = 0;
- len (wa) = 1 + len(w); for $w \in \Sigma^*, a \in \Sigma$

Reversal:
- $\varepsilon^{\mathrm{R}} = \varepsilon$
- $(wa)^{\mathrm{R}} = aw^{\mathrm{R}}$ for $w \in \Sigma^*, a \in \Sigma$

Concatenation:
- $x \bullet \varepsilon = x$ for $x \in \Sigma^*$
- $x \bullet wa = (x \bullet w)a$ for $x, w \in \Sigma^*, a \in \Sigma$

review: structural induction

How to prove $\forall x \in S, P(x)$ is true:

Base Case: Show that $P(u)$ is true for all specific elements u of S mentioned in the *Basis step*

Inductive Hypothesis: Assume that P is true for some arbitrary values of *each* of the existing named elements mentioned in the *Recursive step*

Inductive Step: Prove that $P(w)$ holds for each of the new elements w constructed in the *Recursive step* using the named elements mentioned in the Inductive Hypothesis

Conclude that $\forall x \in S, P(x)$

structural induction for strings

Let S be a set of strings over $\Sigma = \{a, b\}$ defined by

Basis: $a \in S$

Recursive:

If $w \in S$ then $wa \in S$ and $wba \in S$

If $u, v \in S$ then $uv \in S$

Claim: If $w \in S$ then w has more a's than b's.

proof continued?

prove: $\text{len}(x \cdot y) = \text{len}(x) + \text{len}(y)$ for all $x, y \in \Sigma^*$

Let $P(y)$ be "$\text{len}(x \cdot y) = \text{len}(x) + \text{len}(y)$ for all $x \in \Sigma^*$

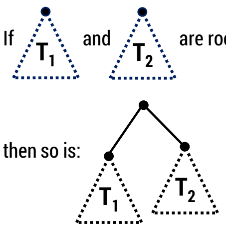
Length:
len $(\varepsilon) = 0$;
len $(wa) = 1 +$ len(w); for $w \in \Sigma^*, a \in \Sigma$

review: rooted binary trees

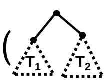
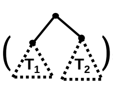
- Basis: • is a rooted binary tree
- Recursive step:

If T_1 and T_2 are rooted binary trees,

then so is:



defining a function on rooted binary trees

- size(•) = 1
- size () = 1 + size(T_1) + size(T_2)
- height(•) = 0
- height ()=1 + max{height(T_1), height(T_2)}

size vs. height

Claim: For every rooted binary tree T, $\text{size}(T) \leq 2^{\text{height}(T)+1} - 1$

languages: sets of strings

Sets of strings that satisfy special properties are called languages.

Examples:
- English sentences
- Syntactically correct Java/C/C++ programs
- Σ^* = All strings over alphabet Σ
- Palindromes over Σ
- Binary strings that don't have a 0 after a 1
- Legal variable names, keywords in Java/C/C++
- Binary strings with an equal # of 0's and 1's

regular expressions

Regular expressions over Σ

- Basis:
  $\varnothing$, ε are regular expressions
  a is a regular expression for any $a \in \Sigma$
- Recursive step:
  - If **A** and **B** are regular expressions then so are:
    (A ∪ B)
    (AB)
    A*

each regular expression is a "pattern"

ε matches the empty string
a matches the one character string *a*

(A ∪ B)
matches all strings that either **A** matches or **B** matches (or both)

(AB)
matches all strings that have a first part that **A** matches followed by a second part that **B** matches

A*
matches all strings that have any number of strings (even 0) that **A** matches, one after another

examples

- ***001****
- ***0*1****
- *(**0** ∪ **1**)**0**(**0** ∪ **1**)**0***
- *(**0*1***)**
- *(**0** ∪ **1**)** **0110** (**0** ∪ **1**)**
- *(**00** ∪ **11**)** (**01010** ∪ **10001**)(**0** ∪ **1**)**

regular expressions in practice

- Used to define the "tokens": e.g., legal variable names, keywords in programming languages and compilers
- Used in **`grep`**, a program that does pattern matching searches in UNIX/LINUX
- Pattern matching using regular expressions is an essential feature of PHP
- We can use regular expressions in programs to process strings!

regular expressions in java

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

`[01]`	a 0 or a 1 `^` start of string `$` end of string	
`[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)

matching email addresses: RFC 822

What?! No nested comments?

more examples

- All binary strings that have an even # of 1's
- All binary strings that *don't* contain 101

limitations of regular expressions

- Not all languages can be specified by regular expressions
- Even some easy things like
  - Palindromes
  - Strings with equal number of 0's and 1's
- But also more complicated structures in programming languages
  - Matched parentheses
  - Properly formed arithmetic expressions
  - etc.