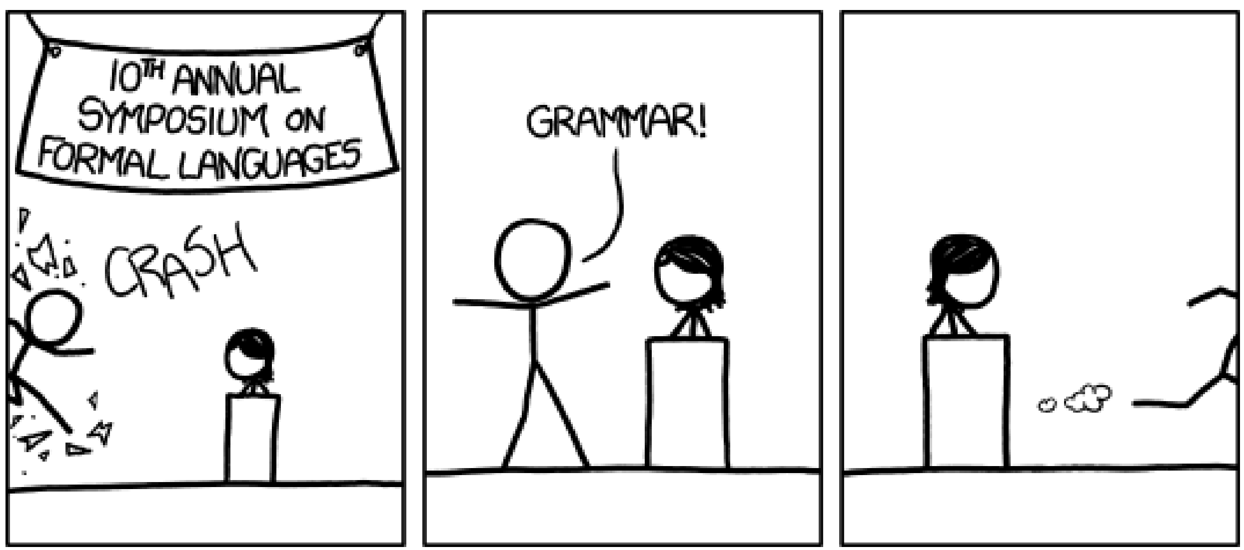
cse 311: foundations of computing

Fall 2015

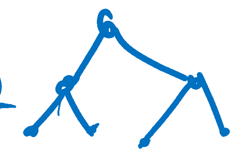
Lecture 20: Regular expressions and context-free grammars



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

$P(T) = $ "$\text{size}(T) \leq 2^{\text{height}(T)+1} - 1$"

size = 7
height = 2



Base Case: $P(\cdot)$ $\quad \text{size}(\cdot) = 1 \leq 2^{0+1} - 1 = 2^{\text{height}(\cdot)+1} - 1$ ✓

IH: For T_1, T_2, $P(T_1), P(T_2)$ hold.

IS: $P($  $)$ holds

T_3

$\text{size}(T_3) = 1 + \text{size}(T_1) + \text{size}(T_2) \overset{IH}{\leq} 1 + 2^{\text{height}(T_1)+1} - 1 + 2^{\text{height}(T_2)+1} - 1$

$\leq 2 \cdot \left(2^{\text{height}(T_1)} + 2^{\text{height}(T_2)}\right) - 1$

$\leq 2 \cdot 2 \cdot \left(2^{\max\{\text{height}(T_1), \text{height}(T_2)\}}\right) - 1$

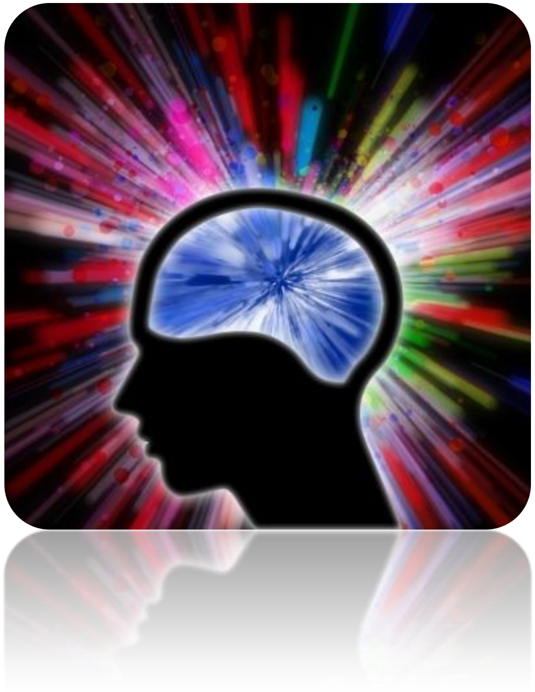
$= 2 \cdot 2^{\max\{\text{height}(T_1), \text{height}(T_2)\}+1} - 1$

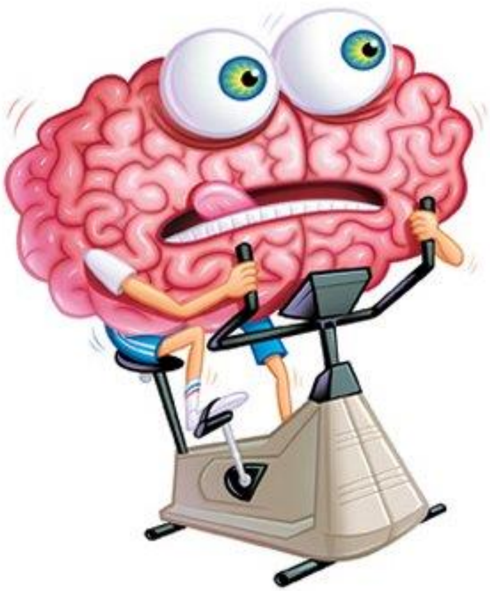
$= 2 \cdot 2^{\text{height}(T_3)} - 1 = 2^{\text{height}(T_3)+1} - 1$

$\Rightarrow P(T_3)$.

Fall 2015

Lecture 20: Regular expressions and context-free grammars





languages: sets of strings

Sets of strings that satisfy special properties are called **languages**.

$L \subseteq \Sigma^*$

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

$L = \{ 0 a_1 a_2 \ldots a_k 0 : a_1, \ldots, a_k \in \{0,1\} \}$

regular expressions

Regular expressions over Σ

- Basis:
  - $\emptyset$, 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***

$L_1 = \emptyset$

$L_2 = \{\varepsilon\}$

$\varepsilon \in L_2$

$\varepsilon \notin L_1$

$L = \{a, \varepsilon\}$

each regular expression is a “pattern”

ε matches the empty string

a matches the one character string *a*

a $\{a\}$

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

$\{a \cup b\}$ $\{a, b\}$

(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

$\{\varepsilon, A, AA, AAA, \ldots\}$

- 001* = {00, 001, 0011, 00111, ---}
- 0*1* = {ε, 0, 1, 01, 001, 011
  ones need to follow zeros
  10
- (0 ∪ 1)0(0 ∪ 1)0 = {0000, 0010, 1000, 1010}
- (0*1*)* = Σ^*
  (0 ∪ 1)*
- (0 ∪ 1)* 0110 (0 ∪ 1)* any string containing 0110
- (00 ∪ 11)* = {a,b}*
  a = 00
  b = 11

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** $\cup$ **B**)
a**?**	zero or one of a	(**A** $\cup$ $\mathcal{E}$)
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

shayan (good)@uw.edu

shayan@uw.edu

(a(b))

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
\t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)
?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t
]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031
]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](
?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?
:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?
:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?
:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|
\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>
@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"
(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?
:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[
\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(
```

What?! No nested comments?

- All binary strings that have at least one 1.

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

- All binary strings that have an even # of 1's

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

  ~~$(0 \cup 11)^*$~~

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

- 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.

- A Context-Free Grammar (CFG) is given by a finite set of substitution rules involving
  - A finite set **V** of *variables* that can be replaced
  - Alphabet Σ of *terminal symbols* that can't be replaced
  - One variable, usually **S**, is called the *start symbol*

- The rules involving a variable **A** are written as

$$\mathbf{A} \rightarrow w_1 \mid w_2 \mid \cdots \mid w_k$$

where each w_i is a string of variables and terminals:

$$w_i \in (\mathbf{V} \cup \Sigma)^*$$

how CFGs generate strings

- Begin with start symbol **S**

- If there is some variable **A** in the current string you can replace it by one of the w's in the rules for **A**
  - $\mathbf{A} \rightarrow w_1 \mid w_2 \mid \cdots \mid w_k$
  - Write this as $x\mathbf{A}y \Rightarrow xwy$
  - Repeat until no variables left

- The set of strings the CFG generates are all strings produced in this way that have no variables

Example: $\mathbf{S} \rightarrow 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \varepsilon$

Example: $\mathbf{S} \rightarrow 0\mathbf{S} \mid \mathbf{S}1 \mid \varepsilon$

Grammar for $\{0^n 1^n : n \geq 0\}$

(all strings with same # of 0's and 1's with all 0's before 1's)

Example: **S** $\rightarrow$ **(S)** | **SS** | ε

simple arithmetic expressions

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

Generate $(2*x) + y$

Generate $x+y*z$ in two fundamentally different ways

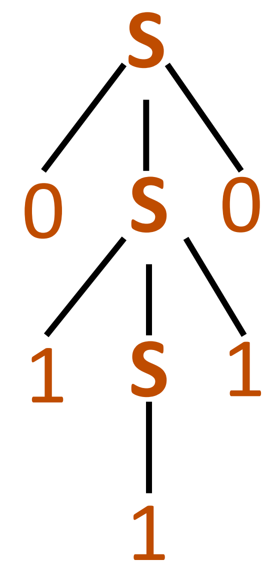
Suppose that grammar G generates a string x

A **parse tree** of x for G has

- Root labeled S (start symbol of G)
- The children of any node labeled **A** are labeled by symbols of w left-to-right for some rule **A** $\rightarrow$ w
- The symbols of x label the leaves ordered left-to-right

$$\mathbf{S} \rightarrow 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \varepsilon$$

Parse tree of 01110:



CFGs and recursively-defined sets of strings

- A CFG with the start symbol **S** as its only variable recursively defines the set of strings of terminals that **S** can generate

- A CFG with more than one variable is a simultaneous recursive definition of the sets of strings generated by *each* of its variables
  - Sometimes necessary to use more than one

building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term   **F** – factor   **I** – identifier **N** - number

$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E}+\mathbf{T}$

$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F} * \mathbf{T}$

$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$

$\mathbf{I} \rightarrow x \mid y \mid z$

$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Backus-Naur form (same as CFG)

BNF (Backus-Naur Form) grammars

- Originally used to define programming languages
- Variables denoted by long names in angle brackets, e.g.

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

  <assignment-statement>, <condition>

  $::=$ used instead of $\rightarrow$

BNF for C

```
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 )?
```

parse trees

Back to middle school:

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

<noun phrase>::==<article><adjective><noun>

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

<object>::=<noun phrase>

Parse:

The yellow duck squeaked loudly

The red truck hit a parked car