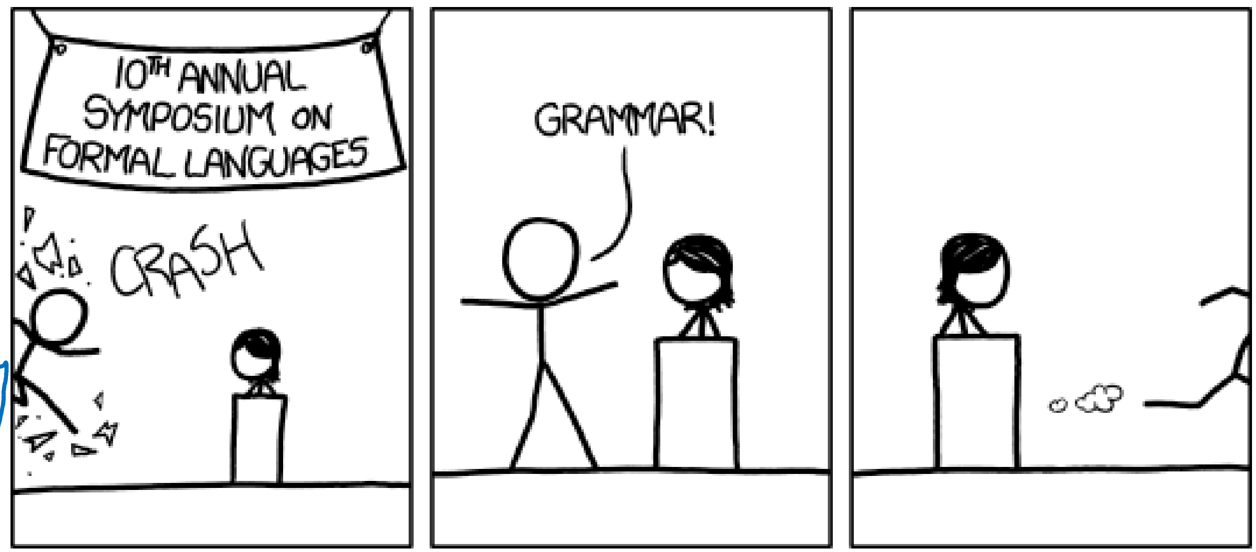
CSE 311: Foundations of Computing

Lecture 19: Regular Expressions & Context-Free Grammars

1. See updated HW6 online with clarifications (not changed in Canvas)

2. If you got less than 5 points on midterm 2(a) you may submit your midterm for a regrade of Q2 only: Mark "Regrade #2" on upper right corner of front of exam and bring it to class/section by This Friday.
or my office hour



[Audience looks around]
"What is going on? There must be some context we're missing"

Review: 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)

union

concatenation

(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

(Kleene) star

Examples

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

Examples

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

  e.g., 0*(10*10*)*

- All binary strings that *don't* contain 101

  e.g., 0*(1 U 000*)* 0*

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.

Context-Free Grammars

- 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 – that is $w_i \in (\mathbf{V} \cup \Sigma)^*$

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 Context-Free Grammars

$\Sigma = \{0, 1\}$

$V = \{S\}$

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

$S \Rightarrow 0S0 \Rightarrow 01S10 \Rightarrow 01110$

$S \Rightarrow 0S0 \Rightarrow 00$

$S \Rightarrow \varepsilon$

$S \Rightarrow 0$

$S \Rightarrow 1$

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

$0^* 1^*$

Example Context-Free Grammars

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

The set of all binary palindromes

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

0*1*

Example Context-Free Grammars

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)

$P \rightarrow 0P1 \mid \varepsilon$

$S \rightarrow 0S1 \mid \varepsilon$

$S \rightarrow 0S \mid S1 \mid \varepsilon$

Example: $\quad S \rightarrow (S) \mid SS \mid \varepsilon$

$S \Rightarrow SS \Rightarrow (S)S \Rightarrow (SS)S$

$\Rightarrow ((S)S)S$

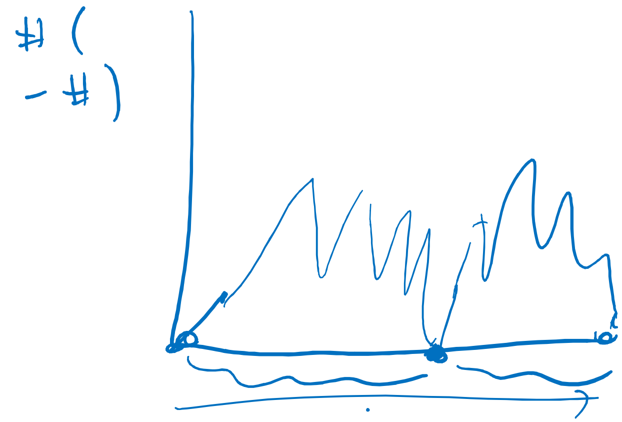
$\Rightarrow (())(S)$

Example Context-Free Grammars

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

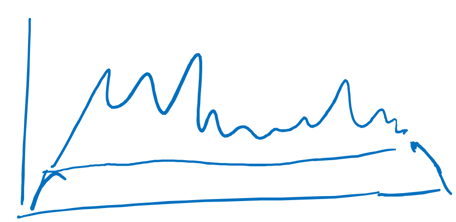
(all strings with same # of 0's and 1's with all 0's before 1's)

$$S \rightarrow 0S1 \mid \varepsilon$$



Example: $S \rightarrow (S) \mid SS \mid \varepsilon$

The set of all strings of matched parentheses



Simple Arithmetic Expressions

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

Generate (2∗x) + y

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

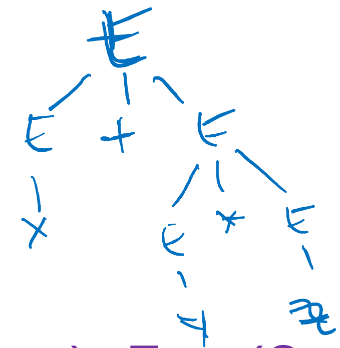
Generate x+y∗z in two fundamentally different ways

$E \Rightarrow E+E \Rightarrow x+E \Rightarrow x+E*E \Rightarrow x+y*E \Rightarrow x+y*z$

$E \Rightarrow E*E \Rightarrow E*z \Rightarrow E+E*z$
$\Rightarrow x+E*z \Rightarrow x+y*z$

Simple Arithmetic Expressions

E⟶ E+E | E∗E | (E) | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



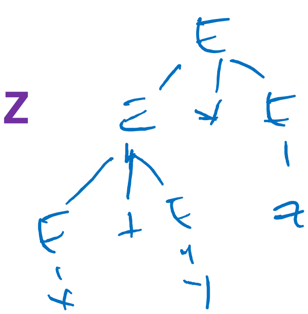
Generate (2∗x) + y

E ⇒ E+E ⇒ (E)+E ⇒ (E∗E)+E ⇒ (2∗E)+E ⇒ (2∗x)+E ⇒ (2∗x)+y

Generate x+y∗z in two fundamentally different ways

E ⇒ E+E ⇒ x+E ⇒ x+E∗E ⇒ x+y∗E ⇒ x+y∗z

E ⇒ E∗E ⇒ E+E∗E ⇒ x+E∗E ⇒ x+y∗E ⇒ x+y∗z

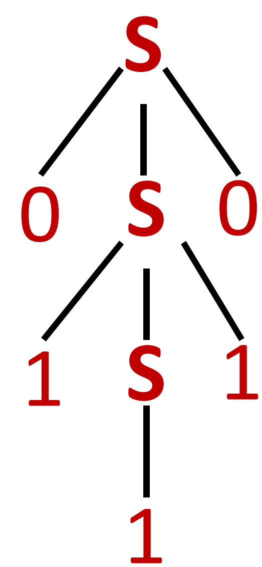


Parse Trees

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 $A \to w$
  - The symbols of x label the leaves ordered left-to-right

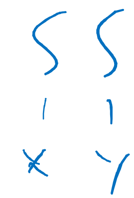
$$S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$$



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

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

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

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

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

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

Backus-Naur Form (The same thing…)

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