

**CSE
31F**

Foundations of Computing I

* All slides are a combined effort between
previous instructors of the course

CSE 311: Foundations of Computing

Lecture 19: Regular Expressions and Context-Free Grammars



Regular Expression Examples

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

~~$(0^*) (1(0^*)1)^* (0^*)$~~ 11011
 $\rightarrow 0^* (1(0^*)1)^* 0^*$ 11

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

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

$0^* (1^* 0)^*$ $0^* (1^* 000^* 1^*)^* 1^* 0^*$
 $0^* 1^* 0^*$ 111001

- Let $\Sigma = \{a, b, e\}$. All strings with no two consecutive vowels.

$0^* (1^* 000^* 1^*)^* 0^*$ $0^* (1^* 00)^* (001)^* (10^* 1)^*$
 $1^* 001$

Regular Expression Examples

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

$$0^*(10^*10^*)^*$$

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

$$0^*(1 \cup 000^*)^*0^*$$

- Let $\Sigma = \{a, b, e\}$. All strings with no two consecutive vowels.

$$b^* \cup (b^*(b^*(a \cup e)b)^*(a \cup e)b^*)$$

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 \mathbf{V} of *variables* that can be replaced
 - Alphabet Σ of *terminal symbols* that can't be replaced
 - One variable, usually \mathbf{S} , is called the *start symbol*
- The rules involving a variable \mathbf{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 $\mathbf{xAy} \Rightarrow \mathbf{xwy}$
 - Repeat until no variables left
- The set of strings the CFG generates are all strings produced in this way that have no variables

Context-Free Grammar Example

CFG: $S \rightarrow OS \mid 1S \mid \epsilon$

$S \Rightarrow OS \Rightarrow 01S \Rightarrow 01$

$(01)^*$

$S \rightarrow OS \mid 1S \mid \epsilon$

Context-Free Grammar Example

CFG: $S \rightarrow 0S \mid 1S \mid \varepsilon$

$S \Rightarrow 0S \Rightarrow 00S \Rightarrow 000S \Rightarrow 000\varepsilon \Rightarrow 000$

$S \Rightarrow 0S \Rightarrow 01S \Rightarrow 010S \Rightarrow 010\varepsilon \Rightarrow 010$

All binary strings!

Equivalent Regular Expression: $(0 \cup 1)^*$

5

Context-Free Grammar Example

Regular Expression: $(0 \cup 1)^*1$

Context-Free Grammar Example

Regular Expression: $(0 \cup 1)^*1$

CFG: $S \rightarrow 0S \mid 1S \mid 1$

These accept the same sets of strings!

Regular Expressions vs. CFGs

There is no regular expression for **palindromes** (with $\Sigma=\{0,1\}$). (We'll prove this later.)

Is there a CFG for it?

$$S \rightarrow \varepsilon \mid 1 \mid 0 \mid 1s1 \mid 0s0$$

Regular Expressions vs. CFGs

There is no regular expression for **palindomes** (with $\Sigma=\{0,1\}$). (We'll prove this later.)

Is there a CFG for it?

Yes: $S \rightarrow OSO \mid 1S1 \mid \varepsilon \mid 1 \mid 0$

Is there a CFG for every regular expression?

There is! We won't prove this, though.

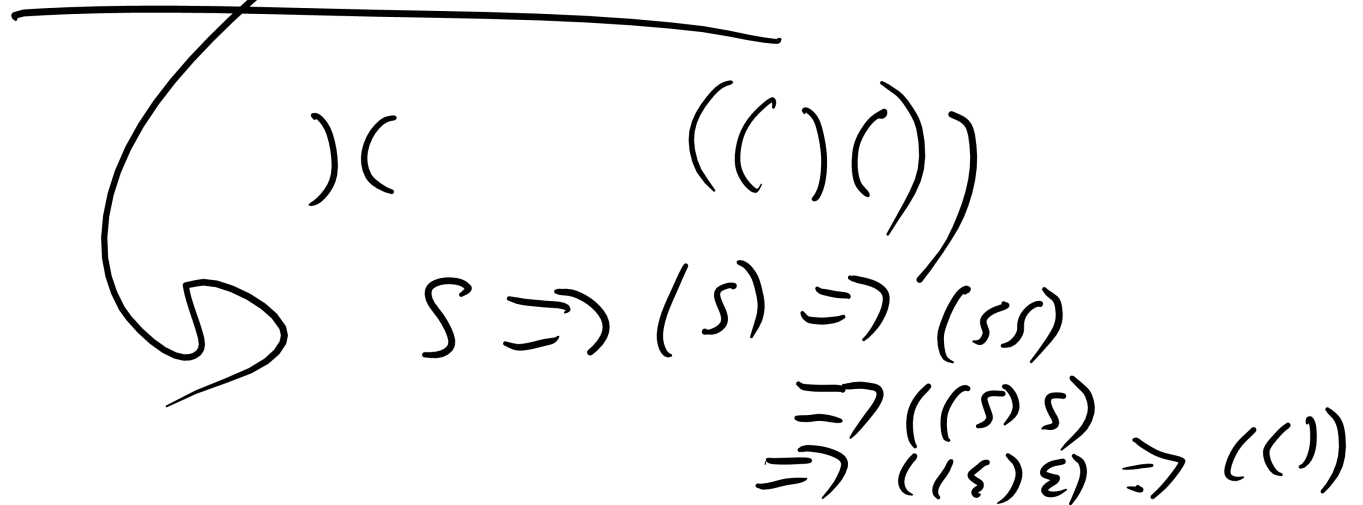
Example Context-Free Grammars

Find a CFG for $\{0^n 1^n : n \geq 0\}$.

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

$\underbrace{\quad \quad \quad}_\varepsilon$
 $0\varepsilon 1 \rightarrow 01$

What strings does $S \rightarrow (S) \mid SS \mid \varepsilon$ generate?



Example Context-Free Grammars

Find a CFG for $\{0^n 1^n : n \geq 0\}$.

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

What strings does $S \rightarrow (S) \mid SS \mid \varepsilon$ generate?

Balanced Parentheses!

Simple Arithmetic Expressions

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

$E \Rightarrow 1$

$\Rightarrow (2 * 3) * 4$

Is there more than one “meaning” of “ $x + y * z$ ”?

Yes: $(x + y) * z, x + (y * z)$

$(1 + 2) * 3 = 7$
 $1 + 2 * 3 = 9$

Generate it once for each meaning.

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

Parse Trees

Suppose that grammar **G** generates a string **x**

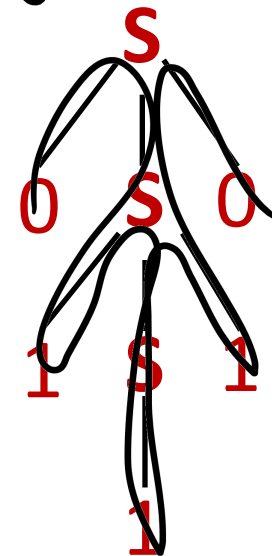
- A *parse tree* of **x** for **G** has
 - Root labeled **S** (start symbol of **G**)
 - The children of any node labeled **A** are labeled by symbols of **w** left-to-right for some rule **A** \rightarrow **w**
 - The symbols of **x** label the leaves ordered left-to-right

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

S \rightarrow **0S0** | **1S1** | **0** | **1** | ϵ

9

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 Arithmetic Expressions

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

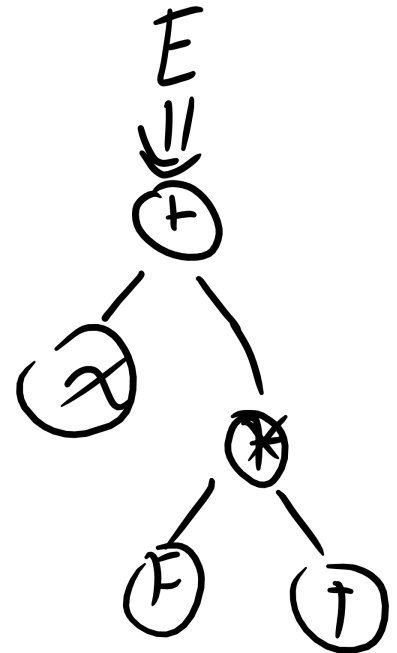
$E \rightarrow T \mid E+T$

$T \rightarrow F \mid F*T$

$F \rightarrow (E) \mid I \mid N$

$I \rightarrow x \mid y \mid z$

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



$x + (y * 5)$

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