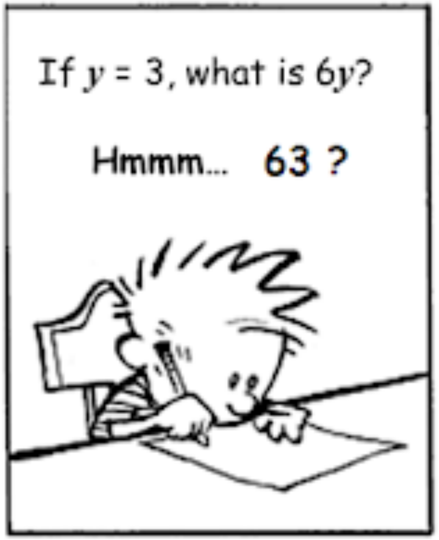
CSE 311: Foundations of Computing

Lecture 20: CFGs, Relations



Last class: Context-Free Grammars

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

- The substitution rules involving a variable **A**, 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 $w_i \in (\mathbf{V} \cup \Sigma)^*$

Last class: How CFGs generate strings

- Begin with "**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 describes are all strings, containing no variables, that can be *generated* in this manner after a finite number of steps

Last class: Examples

$S \Rightarrow 0S \Rightarrow 0S1 \Rightarrow 00S1$
$\Rightarrow 001$

Grammar	Language
S → 0S | S1 | ε	*0*1**
S → 0S0 | 1S1 | 0 | 1 | ε	The set of all binary palindromes
S → 0S1 | ε	$\{0^n 1^n : n \geq 0\}$
S → 0S11 | ε	$\{0^n 1^{2n} : n \geq 0\}$
S → A10 A → 0A1 | ε	$\{0^n 1^{n+1} 0 : n \geq 0\}$
S → (S) | SS | ε	The set of all strings of matched parentheses

$S \to (S) \mid \varepsilon \quad (()(()))(())$

Example Context-Free Grammars

Binary strings with equal numbers of 0s and 1s

(not just 0^n1^n, also 0101, 0110, etc.)

$S \rightarrow 0S1 \mid 1S0 \mid \varepsilon \mid SS$

1001

$S \Rightarrow \ldots \Rightarrow \ldots \Rightarrow w$

$\#0 = \#1$

$\#0 = \#1$

Example Context-Free Grammars

Binary strings with equal numbers of 0s and 1s

(not just 0^n1^n, also 0101, 0110, etc.)

$$S \rightarrow SS \mid 0S1 \mid 1S0 \mid \varepsilon$$

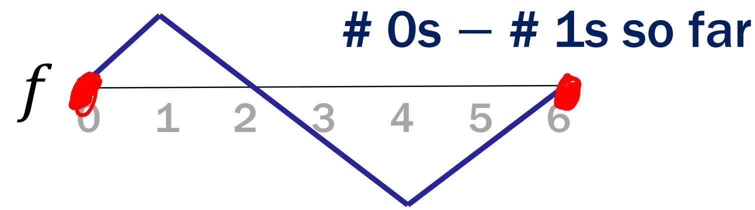
A standard structural induction can show that everything generated by S has an equal # of 0s and 1s

Intuitively, why does this generate all such strings?

Example Context-Free Grammars

Let $x \in \{0,1\}^*$. Define $f_x(k)$ to be the # of 0s minus # of 1s in the first k characters of x.

E.g., for x = 011100



$f_x(k) = 0$ when first k characters have #0s = #1s

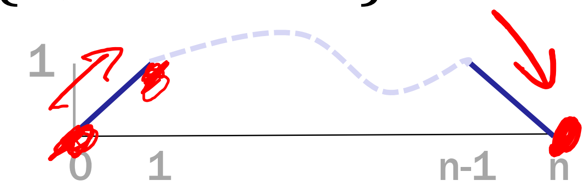
– starts out at 0 $\qquad f_x(0) = 0$

– ends at 0 $\qquad f_x(n) = 0$

Example Context-Free Grammars

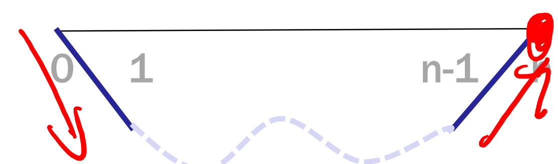
Three possibilities for $f_x(k)$ for $k \in \{1, \dots, n-1\}$

- $f_x(k) > 0$ for all such k



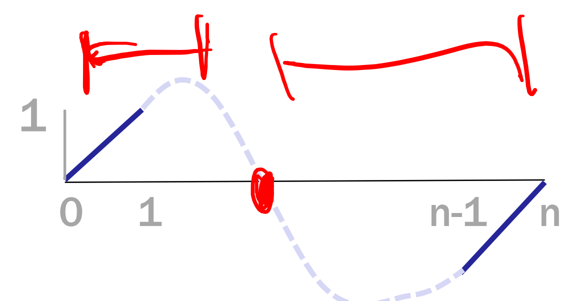
$$\mathbf{S \rightarrow 0S1}$$

- $f_x(k) < 0$ for all such k

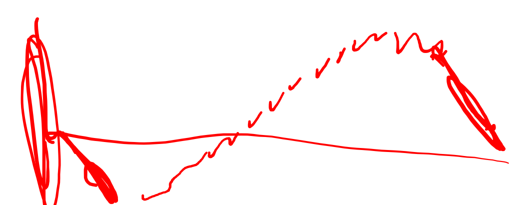


$$\mathbf{S \rightarrow 1S0}$$

- $f_x(k) = 0$ for some such k



$$\mathbf{S \rightarrow SS}$$



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 $(2*x) + y$

$$E \Rightarrow E+E \Rightarrow E+y \Rightarrow (E)+y$$

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

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

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 $(2*x) + y$

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

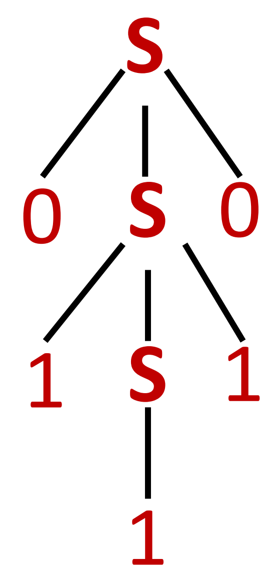
Parse Trees

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

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

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



Parse tree of 01110

Simple Arithmetic Expressions

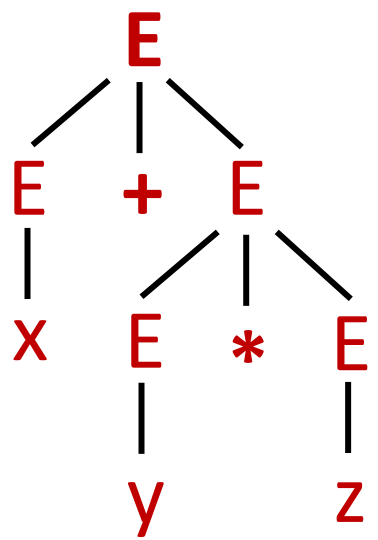
$$\mathtt{E} \rightarrow \mathtt{E+E} \mid \mathtt{E*E} \mid (\mathtt{E}) \mid \mathtt{x} \mid \mathtt{y} \mid \mathtt{z} \mid \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \mathtt{3} \mid \mathtt{4} \mid \mathtt{5} \mid \mathtt{6} \mid \mathtt{7} \mid \mathtt{8} \mid \mathtt{9}$$

Generate x+y∗z in two ways that give two *different* parse trees

Simple Arithmetic Expressions

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

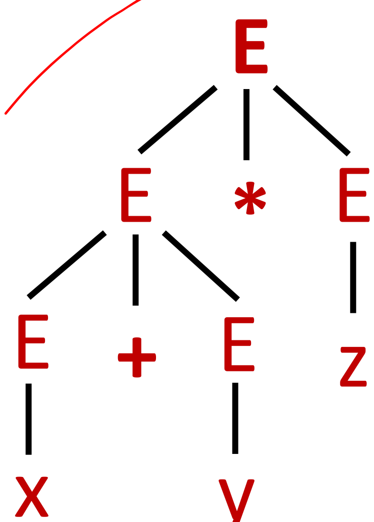
Generate x+y∗z in ways that give two *different* parse trees



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

(add x to the product of y and z)

x+(y*z)



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

(add x to y, then multiply by z)

(x+y)*z

building precedence in simple arithmetic expressions

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

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

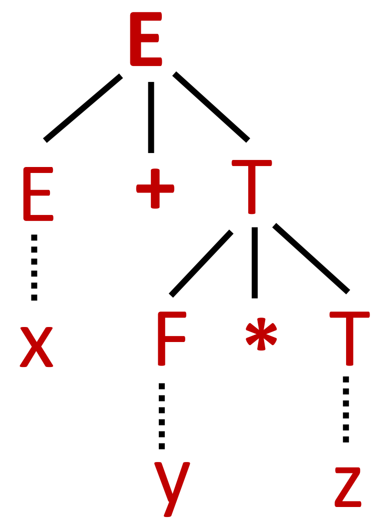
$$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F}*\mathbf{T}$$

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

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

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

$x+y*z$



$E \Rightarrow E+T \Rightarrow T+T \Rightarrow T+F*T$

$\Rightarrow \ldots$

building precedence in simple arithmetic expressions

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

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

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

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

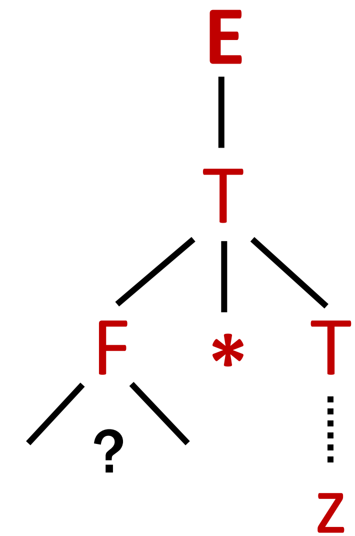
$\mathbf{I} \rightarrow x \mid y \mid z$

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

$x + y * z$

$E \Rightarrow T \Rightarrow F*T \Rightarrow F*F$
$\Rightarrow F*I$
$\Rightarrow F*z$

No longer allows:



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

recursive sets

=

CFGs $\supsetneq$ Reg exp

CFGs and regular expressions

Theorem: For all regular expressions A there is a CFG that generates precisely the strings A matches

Proof: Structural Induction

- **Basis:**
  - ε is a regular expression
  - ***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*

CFGs can do everything REs can

- CFG to match RE **ε**

$$\mathbf{S} \rightarrow \varepsilon$$

- CFG to match RE ***a*** (for any a $\in \Sigma$)

$$\mathbf{S} \rightarrow \text{a}$$

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

    A $\cup$ B

    AB

    A*

CFGs can do everything REEs can

Suppose CFG with start symbol $\mathbf{S_A}$ matches RE **A**

IHs: CFG with start symbol $\mathbf{S_B}$ matches RE **B**

(Then rename variables so no vars used in both)

- CFG to match RE **A** $\cup$ **B**

  Add $\mathbf{S} \rightarrow \mathbf{S_A} \mid \mathbf{S_B}$

  \+ rules from both CFGs

$S \Rightarrow S_A$

$S \Rightarrow S_B$

- CFG to match RE **AB**

  Add $\mathbf{S} \rightarrow \mathbf{S}_A\, \mathbf{S}_B$

  \+ rules from both CFGs

$S \Rightarrow S_A S_B$

$\Rightarrow \cdots$

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

    A $\cup$ **B**

    AB

    A*

CFGs can do everything that REs can

- CFG to match RE **A***

  Add $\mathbf{S} \rightarrow \mathbf{S_A}\,\mathbf{S} \mid \varepsilon$

  + rules from CFG with $\mathbf{S_A}$

"S = list of A"

$S \Rightarrow \varepsilon$

$S \Rightarrow S_A S \Rightarrow S_A S_A S \Rightarrow S_A S_A S_A S$

$\Rightarrow S_A S_A S_A$

- **Basis:**
  - ε is a regular expression
  - ***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*

So far: Languages — REs and CFGs

Two new ways of defining languages

- Regular Expressions $(0 \cup 1)^*\,0110\,(0 \cup 1)^*$
  - easy to understand (declarative)
- Context-free Grammars $S \rightarrow SS \mid 0S1 \mid 1S0 \mid \varepsilon$
  - more expressive
  - (a way of recursively-defining sets)

We will connect these to machines shortly.

But first, we need some new math terminology….