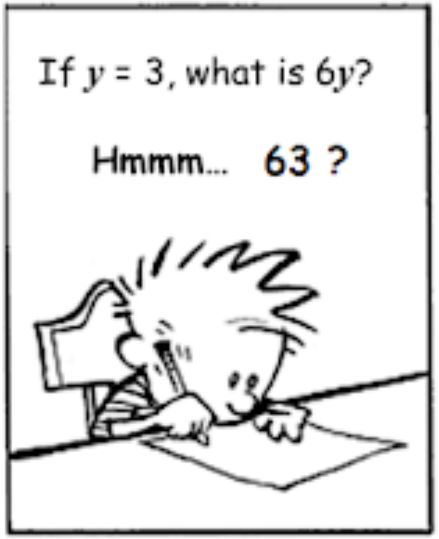
CSE 311: Foundations of Computing

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

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

Grammar	Language
S → 0**S** | **S**1 | ε	***0*1****
S → 0**S**0 | 1**S**1 | 0 | 1 | ε	The set of all binary palindromes
S → 0**S**1 | ε	$\{0^n 1^n : n \geq 0\}$
S → 0**S**11 | ε	$\{0^n 1^{2n} : n \geq 0\}$
S → A10 **A** → 0**A**1 | ε	$\{0^n 1^{n+1} 0 : n \geq 0\}$
S → (**S**) | **SS** | ε	The set of all strings of matched parentheses

Example Context-Free Grammars

Binary strings with equal numbers of 0s and 1s

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

$$S \rightarrow \varepsilon \mid S10 \mid S01 \mid 01S \mid 10S \mid 1S0$$

$$S \rightarrow 0S1$$

Example Context-Free Grammars

Binary strings with equal numbers of 0s and 1s
(not just 0^n1^n, also 0101, 0110, etc.)

0110

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

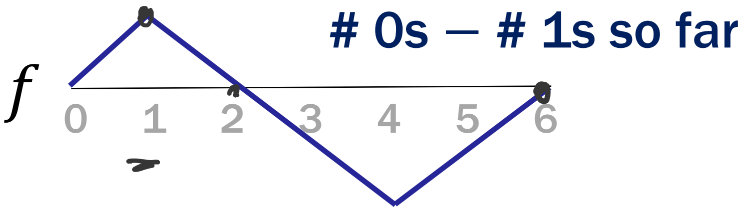
A standard structural induction can show that everything generated by S has an equal # of 0s and 1s

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 $f_x(0) = 0$

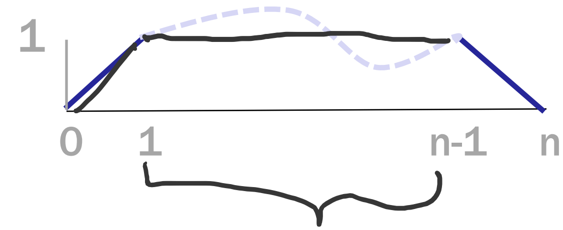
– ends at 0 $f_x(n) = 0$

Example Context-Free Grammars

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

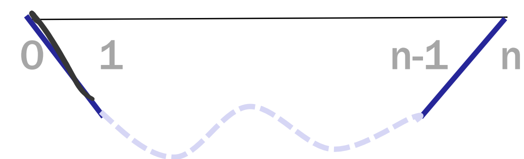
- $f_x(k) > 0$ for all such k

  S → 0S1



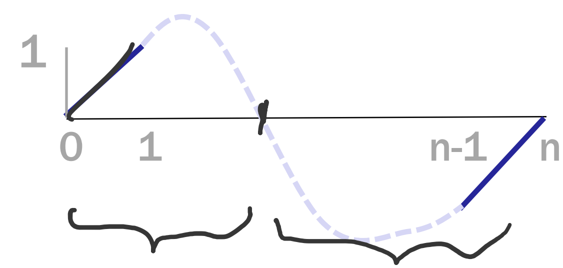
- $f_x(k) < 0$ for all such k

  S → 1S0



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

  S → 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 $(2*x) + y$

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

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 $(2*x) + y$

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

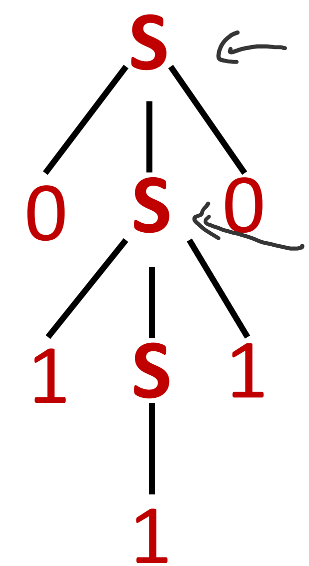
Parse Trees

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

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

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

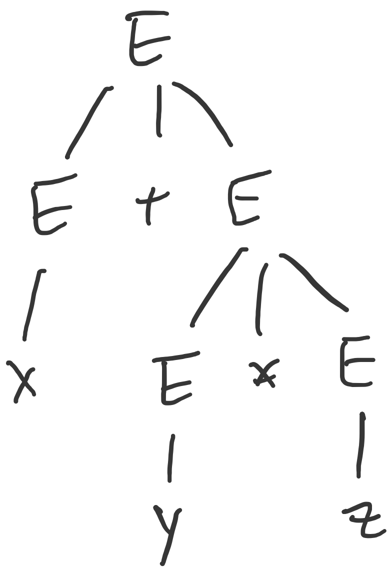


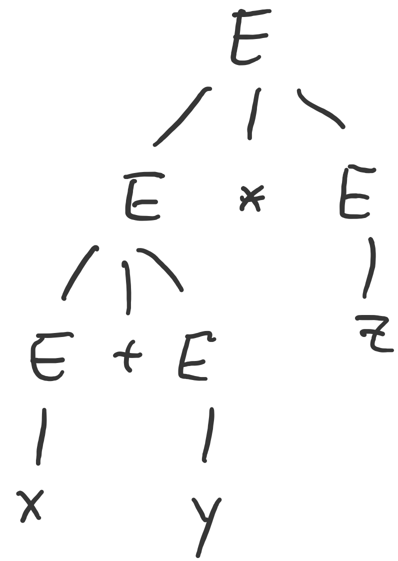
Parse tree of 01110

Simple Arithmetic Expressions

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

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

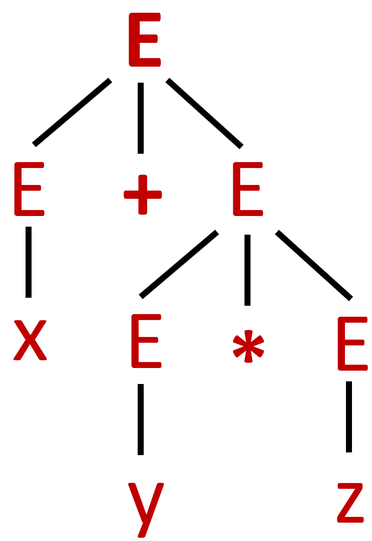




Simple Arithmetic Expressions

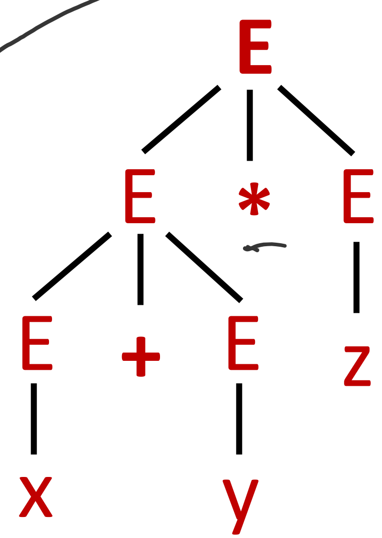
$$E \rightarrow E+E \mid E*E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

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**)



$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow x+E*E$

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

(add **x** to **y**, then multiply by **z**)

building precedence in simple arithmetic expressions

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

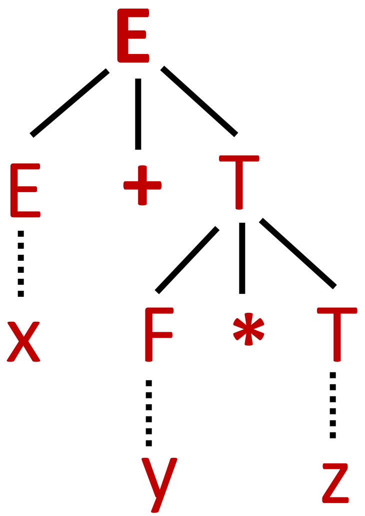
$$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E}+\mathbf{T}$$

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

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

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

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



building precedence in simple arithmetic expressions

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

E → **T** | **E+T**

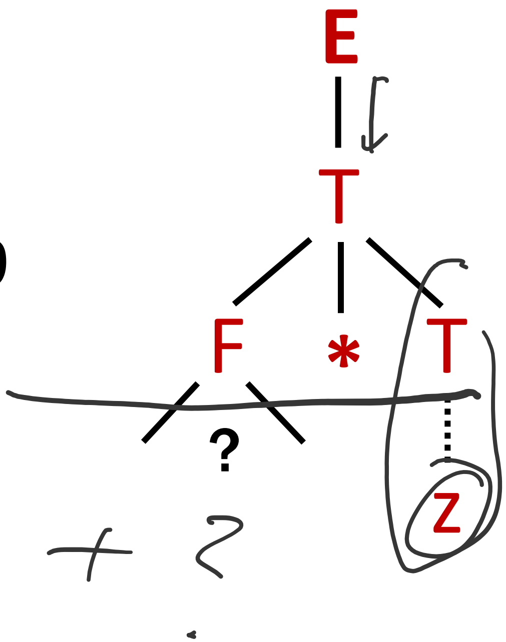
T → **F** | **F∗T**

F → (**E**) | **I** | **N**

I → x | y | z

N → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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

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:

    $A \cup B$

    AB

    A*

CFGs can do everything REs can

- CFG to match RE **ε**

  $S \rightarrow \varepsilon$

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

  $S \rightarrow a$

- **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*

CFGs can do everything REs can

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

- CFG to match RE **AB**

  Add $\mathbf{S} \rightarrow \mathbf{S}_A\, \mathbf{S}_B$
  + rules from both CFGs

- **Basis:**
  - ε is a regular expression
  - *a* is a regular expression for any *a* ∈ Σ
- **Recursive step:**
  - If A and B are regular expressions then so are:

    A ∪ B

    AB

    A*

CFGs can do everything that REs can

- CFG to match RE $\mathbf{A}^*$

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

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

- **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:

    $\mathbf{A} \cup \mathbf{B}$

    AB

    $\mathbf{A}^*$

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 )?
```

BNF for (Simple) English

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

So far: Languages — REs and CFGs

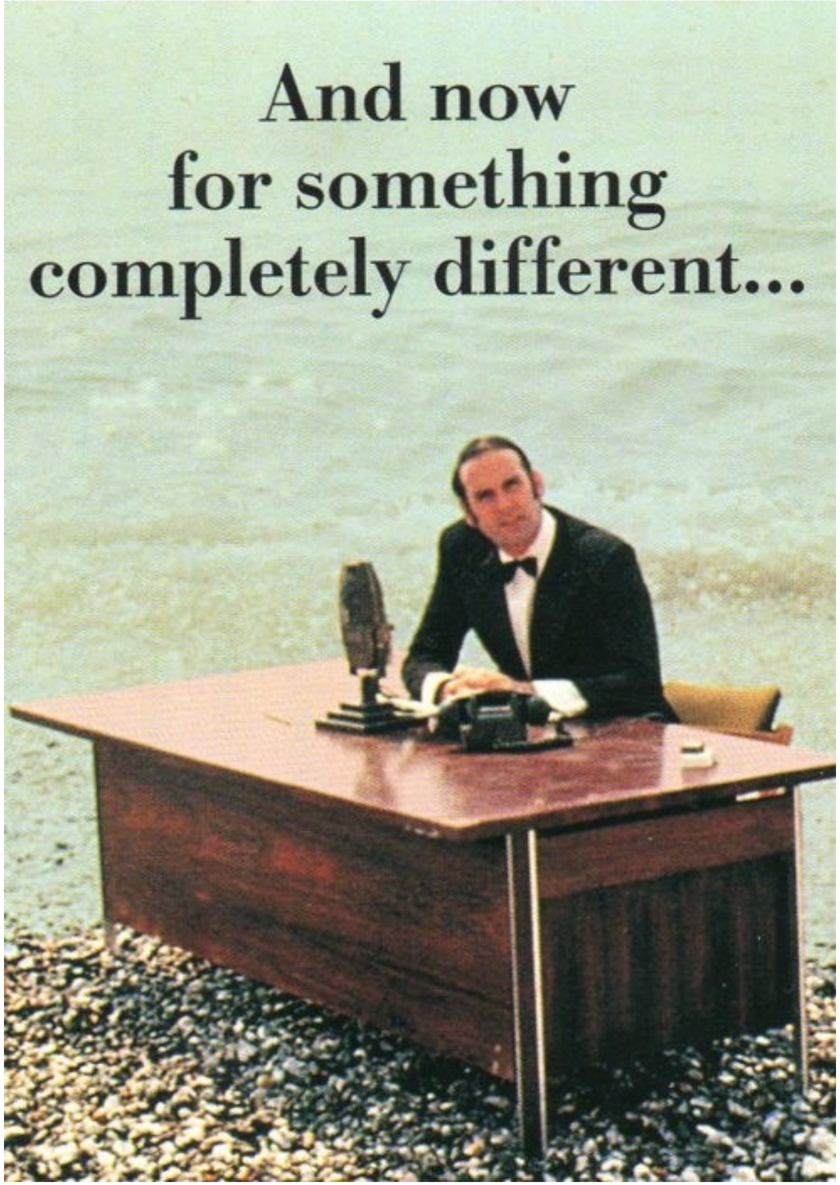
Two new ways of defining languages

- 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….

Relations and Directed Graphs



Relations

Let A and B be sets,
A **binary relation from** A **to** B is a subset of $A \times B$

Let A be a set,
A **binary relation on** A is a subset of $A \times A$

Relations You Already Know

$\geq$ on $\mathbb{N}$

That is, $\{(x,y) : x \geq y \text{ and } x, y \in \mathbb{N}\}$

$<$ on $\mathbb{R}$

That is, $\{(x,y) : x < y \text{ and } x, y \in \mathbb{R}\}$

$=$ on Σ^*

That is, $\{(x,y) : x = y \text{ and } x, y \in \Sigma^*\}$

$\subseteq$ on $\mathcal{P}(U)$ for universe U

That is, $\{(A,B) : A \subseteq B \text{ and } A, B \in \mathcal{P}(U)\}$

More Relation Examples

$\mathbf{R_1}$ **=** {(a, 1), (a, 2), (b, 1), (b, 3), (c, 3)}

$\mathbf{R_2}$ **=** {(x, y) : x ≡ y (mod 5) }

$\mathbf{R_3}$ **=** {(c_1, c_2) : c_1 is a prerequisite of c_2 }

$\mathbf{R_4}$ **=** {(s, c) : student s has taken course c }

Properties of Relations

Let R be a relation on A.

R is **reflexive** iff $(a,a) \in R$ for every $a \in A$

R is **symmetric** iff $(a,b) \in R$ implies $(b,a) \in R$

R is **antisymmetric** iff $(a,b) \in R$ and $a \neq b$ implies $(b,a) \notin R$

R is **transitive** iff $(a,b) \in R$ and $(b,c) \in R$ implies $(a,c) \in R$

Which relations have which properties?

$\geq$ on $\mathbb{N}$: R. not S. A.

$<$ on $\mathbb{R}$: not R. not S.

$=$ on Σ^* : R. S

$\subseteq$ on $\mathcal{P}(U)$: R. not S

$R_2 = \{(x, y) : x \equiv y \pmod 5\}$: R. S

$R_3 = \{(c_1, c_2) : c_1 \text{ is a prerequisite of } c_2\}$: not R, not S.

R is **reflexive** iff $(a,a) \in R$ for every $a \in A$
R is **symmetric** iff $(a,b) \in R$ implies $(b, a) \in R$
R is **antisymmetric** iff $(a,b) \in R$ and $a \neq b$ implies $(b,a) \notin R$
R is **transitive** iff $(a,b) \in R$ and $(b, c) \in R$ implies $(a, c) \in R$

Which relations have which properties?

$\geq$ **on** $\mathbb{N}$ **: Reflexive, Antisymmetric, Transitive**

$<$ **on** $\mathbb{R}$ **: Antisymmetric, Transitive**

$=$ **on** Σ^* **: Reflexive, Symmetric, Antisymmetric, Transitive**

$\subseteq$ **on** $\mathcal{P}$**(U): Reflexive, Antisymmetric, Transitive**

R_2 **=** {(x, y) : x ≡ y (mod 5)}: **Reflexive, Symmetric, Transitive**

R_3 **=** {(c_1, c_2) : c_1 is a prerequisite of c_2 }: **Antisymmetric**

R is **reflexive** iff (a,a) $\in$ R for every a $\in$ A
R is **symmetric** iff (a,b) $\in$ R implies (b, a)$\in$ R
R is **antisymmetric** iff (a,b) $\in$ R and a $\neq$ b implies (b,a) $\notin$ R
R is **transitive** iff (a,b)$\in$ R and (b, c)$\in$ R implies (a, c) $\in$ R