CSE 311: Foundations of Computing

Topic 7: Languages

Strings

- An *alphabet* Σ is any finite set of characters

- The set Σ^* of *strings* over the alphabet Σ
  - example: {0,1}* is the set of *binary strings*
    0, 1, 00, 01, 10, 11, 000, 001, … and “”

- Σ^* is defined recursively by
  - Basis: $\varepsilon \in \Sigma^*$ (ε is the empty string, i.e., “”)
  - Recursive: if $w \in \Sigma^*$, $a \in \Sigma$, then $wa \in \Sigma^*$

Languages: Sets of Strings

- Subsets of strings are called *languages*
- Examples:
  - $\Sigma^* =$ All strings over alphabet Σ
  - palindromes over Σ
  - binary strings with an equal # of 0's and 1's
  - syntactically correct Java/C/C++ programs
  - valid English sentences
  - correct solutions to coding problems:

$$S = \{x\#y \mid y \text{ is Java code that does what } x \text{ says}\}$$

Recall: Building Sets from Predicates

We can define a set from a predicate P:

$$S := \{x : P(x)\}$$

S = the set of all x for which $P(x)$ is true

Almost All of CS is Languages

- **Any predicate can be phrased as "x ∈ S"**
  - **computing "x ∈ S" is as hard as computing any predicate**

- **All math objects can be encoded as strings**
  - **see Java Object's `toString` function**

- **Almost anything can be phrased "x ∈ L" for language L**
  - **only restriction is that predicates have `boolean` output**
  - **but this is usually not a *real* restriction**

    each <u>bit</u> of any output is a T/F value

    so you computing the individual bits can be phrased as "x ∈ S"

Theoretical Computer Science

Foreword on Intro to Theory C.S.

- Look at different ways of defining languages
- See which are more expressive than others
  - i.e., which can define more languages

- Later: connect ways of defining languages to different types of (restricted) computers
  - computers capable of recognizing those languages i.e., distinguishing strings in the language from not

- Consequence: computers that recognize more expressive languages are more powerful

Palindromes

Palindromes are strings that are the same when read backwards and forwards

Basis:

ε is a palindrome

any $a \in \Sigma$ is a palindrome

Recursive step:

If p is a palindrome,

then apa is a palindrome for every $a \in \Sigma$

(note that "apa" really means $\varepsilon a \bullet p \bullet \varepsilon a$)

Regular Expressions

Regular expressions over Σ

- **Basis**:

  ε is a regular expression (could also include $\varnothing$)

  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*

Each Regular Expression is a "pattern"

ε matches only the **empty string**

a matches only the one-character string *a*

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

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 (**ε** ∪ **A** ∪ **AA** ∪ **AAA** ∪ **…**)

Definition of the *language* matched by a regular expression

Language of a Regular Expression

The language defined by a regular expression:

$$\mathrm{L}(\varepsilon) = \{\varepsilon\}$$

$$\mathrm{L}(a) = \{a\}$$

$$\mathrm{L}(A \cup B) = L(A) \cup L(B)$$

$$\mathrm{L}(AB) = \{y \bullet z : \; y \in L(A), z \in L(B)\}$$

$$\mathrm{L}(A^*) = \bigcup_{n=0}^{\infty} L(A^n)$$

A^n defined recursively by

$$A^0 = \{\varepsilon\}$$

$$A^{n+1} = A^n A$$

Examples

001*

0*1*

Examples

001*

{00, 001, 0011, 00111, ...}

0*1*

Any number of 0's followed by any number of 1's

Examples

$(0 \cup 1)\ 0\ (0 \cup 1)\ 0$

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

Examples

$(0 \cup 1)\,0\,(0 \cup 1)\,0$

{0000, 0010, 1000, 1010}

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

All binary strings

Examples

- All binary strings that contain 0110

$$(0 \cup 1)^*\ 0110\ (0 \cup 1)^*$$

- All binary strings that begin with a string of doubled characters (00 or 11) followed by 01010 or 10001 followed by anything

$$(00 \cup 11)^*\ (01010 \cup 10001)\ (0 \cup 1)^*$$

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., $0^*(1 \cup 1000^*)^*(\varepsilon \cup 10)$

  at least two 0s between 1s

Finite languages vs Regular Expressions

- All finite languages have a regular expression.

  (a language is finite if its elements can be put into a list)

  Why?

- Given a list of strings s_1, s_2, ..., s_n

  Construct the regular expression

$$s_1 \cup s_2 \cup \ldots \cup s_n$$

(Could make this formal by induction on n)

Finite languages vs Regular Expressions

- Every regular expression that does not use *
  generates a finite language.

  Why?

- Prove by structural induction on the syntax of regular expressions!

Star-free implies finite

Let A be a regular expression that does not use *. Then L(A) is finite.

Proof: We proceed by structural induction on A.

Case ε: L(ε) = {ε}, which is finite

Case *a*: L(*a*) = {*a*}, which is finite

Case A ∪ B:

L(A ∪ B) = L(A) ∪ L(B)

By the IH, each is finite, so their union is finite.

Star-free implies finite

Let A be a regular expression that does not use *. Then L(A) is finite.

Proof: We proceed by structural induction on A.

Case AB:

$$\mathrm{L(AB)} = \{y \bullet z : \ y \in L(A), z \in L(B)\}$$

By the IH, $\mathrm{L(A)}$ and $\mathrm{L(B)}$ are finite.

Every element of $\mathrm{L(AB)}$ is covered by a pair (y, z) where $y \in L(A)$ and $z \in L(B)$, so $\mathrm{L(AB)}$ is finite.

(No case for A*!)

Finite languages vs Regular Expressions

Key takeaways:

– Regular expressions can represent all finite languages

– To prove a language is "regular", just give the regular expression that describes it.

– Regular expressions are more powerful than finite languages (e.g., 0* is an infinite language)

– To prove something about *all* regular expressions, use structural induction on the syntax.

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** ∪ **B**)
a**?**	zero or one of a	(**A** ∪ ε)
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)

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.

Example Context-Free Grammars

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

How does this grammar generate 0110?

Example Context-Free Grammars

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

How does this grammar generate 0110?

$\mathbf{S} \to 0\mathbf{S}0 \to 01\mathbf{S}10 \to 01\varepsilon10 = 0110$

Example Context-Free Grammars

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

How to describe all strings generated?

The set of all binary palindromes

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

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 describes are all strings, containing no variables, that can be *generated* in this manner (after a finite number of steps)

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

- **This is a claim of set equality**
  - first set defined by a **CFG**, second by a **predicate**

$$\{x \in \{0,1\}^* : S \rightarrow^* x\} = \{x \in \{0,1\}^* : x^R = x\}$$

- **Usually to argue subset directions separately**

Example Context-Free Grammars

Example:

$$\mathbf{S} \rightarrow \mathbf{A} \mid \mathbf{B}$$

$$\mathbf{A} \rightarrow 0\mathbf{A} \mid \varepsilon$$

$$\mathbf{B} \rightarrow 1\mathbf{B} \mid \varepsilon$$

How does this grammar generate 000?

Example Context-Free Grammars

Example: $\mathbf{S} \rightarrow \mathbf{A} \mid \mathbf{B}$

$\mathbf{A} \rightarrow 0\mathbf{A} \mid \varepsilon$

$\mathbf{B} \rightarrow 1\mathbf{B} \mid \varepsilon$

How does this grammar generate 000?

$\mathbf{S} \rightarrow \mathbf{A} \rightarrow 0\mathbf{A} \rightarrow 00\mathbf{A} \rightarrow 000\mathbf{A} \rightarrow 000\varepsilon = 000$

Example Context-Free Grammars

Example:

$$\mathbf{S} \to \mathbf{A} \mid \mathbf{B}$$

$$\mathbf{A} \to 0\mathbf{A} \mid \varepsilon$$

$$\mathbf{B} \to 1\mathbf{B} \mid \varepsilon$$

How to describe all strings generated?

strings of all 0s or all 1s

(all 0s) ∪ (all 1s)

Example Context-Free Grammars

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

Example Context-Free Grammars

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

0*1*

Example Context-Free Grammars

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

(i.e., matching `0*1*` but with same number of 0's and 1's)

Example Context-Free Grammars

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

(i.e., matching `0*1*` but with same number of 0's and 1's)

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

Example Context-Free Grammars

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

(i.e., matching `0*1*` but with same number of 0's and 1's)

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

Grammar for $\{0^n 1^{2n} : n \geq 0\}$

Example Context-Free Grammars

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

(i.e., matching 0*1* but with same number of 0's and 1's)

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

Grammar for $\{0^n 1^{2n} : n \geq 0\}$

$$\mathbf{S} \rightarrow 0\mathbf{S}11 \mid \varepsilon$$

Example Context-Free Grammars

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

(i.e., matching 0*1* but with same number of 0's and 1's)

S $\rightarrow$ 0**S**1 | ε

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

Example Context-Free Grammars

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

(i.e., matching 0*1* but with same number of 0's and 1's)

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

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

$\mathbf{S} \rightarrow \mathbf{A}10$

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

Example Context-Free Grammars

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

The set of all strings of matched parentheses

- This is a claim of set equality
  - first set defined by a CFG, second by a predicate
  - not at all obvious!

Example Context-Free Grammars

Binary strings with equal numbers of 0s and 1s

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

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

Example Context-Free Grammars

Example: **S** $\rightarrow$ **SS | 0S1 | 1S0 |** ε

Set of all $x \in \{0,1\}^*$ with $\#_0(x) = \#_1(x)$

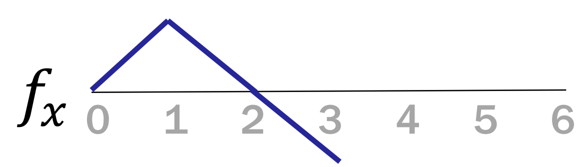
- This is a claim of set equality
  - first set defined by a CFG, second by a predicate
- Need to argue subset directions separately
  - clear that strings from CFG equal 0s and 1s
  - but can the CFG produce any such string?

Example Context-Free Grammars

Define $f_x(k)$ to be the number of "0"s – "1"s in first k characters of x.

E.g., for x = 011100

first 0 characters	""	0 – 0 = 0
first 1 character	"0"	1 – 0 = 1
first 2 characters	"01"	1 – 1 = 0
first 3 characters	"011"	1 – 2 = -1

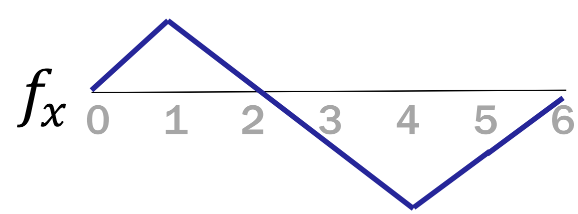


Example Context-Free Grammars

Define $f_x(k)$ to be the number of "0"s – "1"s in first k characters of x.

E.g., for x = 011100

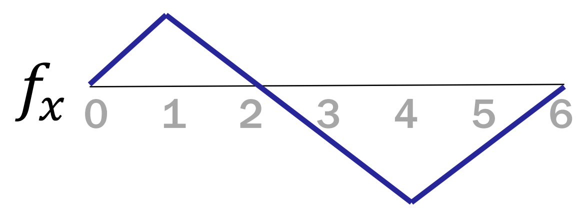
first 0 characters	""	0 – 0 = 0
first 1 character	"0"	1 – 0 = 1
first 2 characters	"01"	1 – 1 = 0
first 3 characters	"011"	1 – 2 = -1
first 4 characters	"0111"	1 – 3 = -2
first 5 characters	"01110"	2 – 3 = -1
all 6 characters	"011100"	3 – 3 = 0



Example Context-Free Grammars

Define $f_x(k)$ to be the number of "0"s – "1"s in first k characters of x.

E.g., for x = 011100



Define $f_x(k)$ to be the number of "0"s – "1"s in first k characters of x.

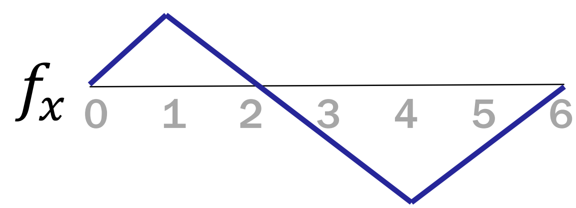
If k-th character is 0, then $f_x(k) = f_x(k-1) + 1$

If k-th character is 1, then $f_x(k) = f_x(k-1) - 1$

Example Context-Free Grammars

Define $f_x(k)$ to be the number of "0"s – "1"s in first k characters of x.

E.g., for x = 011100



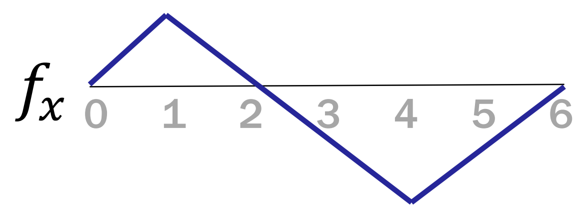
Define $f_x(k)$ to be the number of "0"s – "1"s in first k characters of x.

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

Example Context-Free Grammars

Define $f_x(k)$ to be the number of "0"s – "1"s in 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 $f(0) = 0$

– ends at 0 $f(n) = 0$

Example Context-Free Grammars

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

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

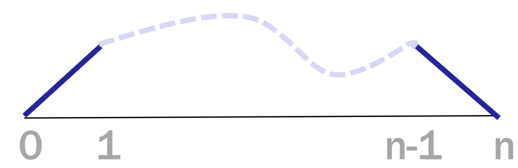
$f_x(k) = 0$ when first k characters have #0s = #1s

- starts out at 0 (immediate) $f(0) = 0$
- ends at 0 iff x is in the language $f(n) = 0$

Example Context-Free Grammars

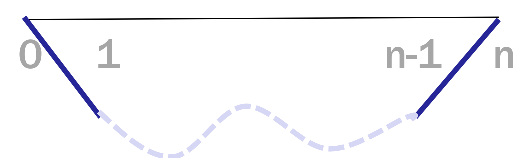
Three possibilities for $f_x(\mathrm{k})$ for $k \in \{1, \ldots, n-1\}$

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



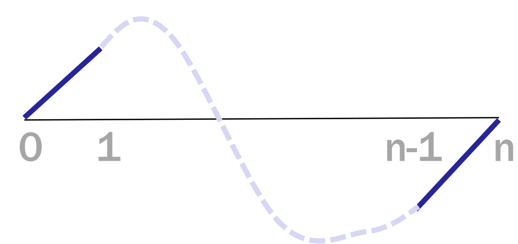
S → 0S1

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



S → 1S0

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



S → SS

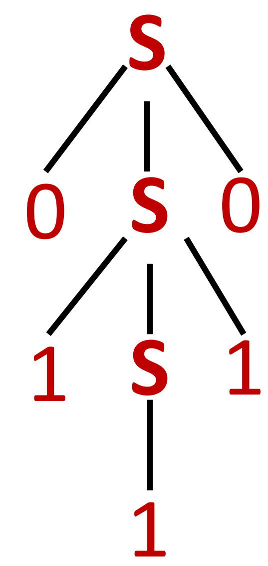
Parse Trees

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

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

Parse tree of 01110

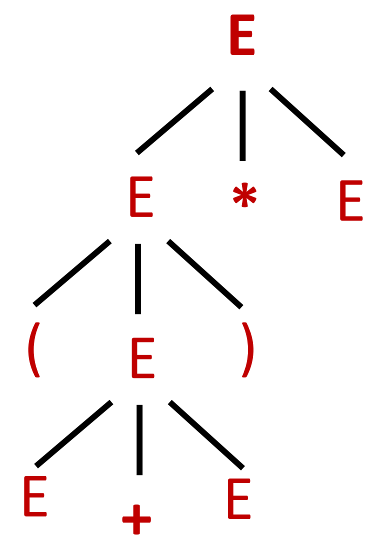


Simple Arithmetic Expressions

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

Generate (2 + x) * y

$$\begin{aligned} E &\rightarrow E*E \\ &\rightarrow (E)*E \\ &\rightarrow (E+E)*E \end{aligned}$$

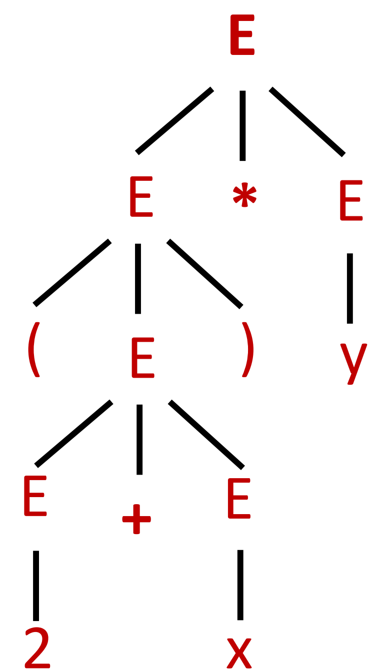


Simple Arithmetic Expressions

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



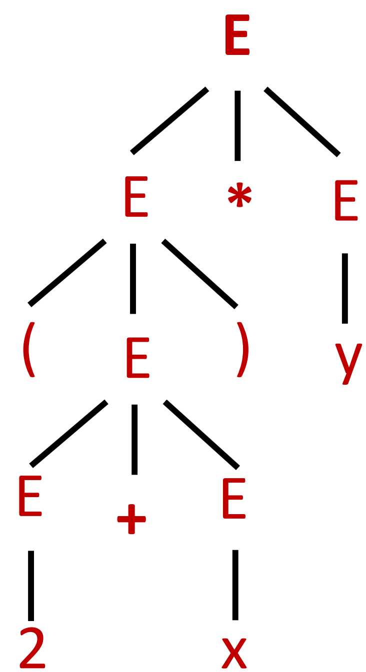
Simple Arithmetic Expressions

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

Generate (2 + x) * y

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

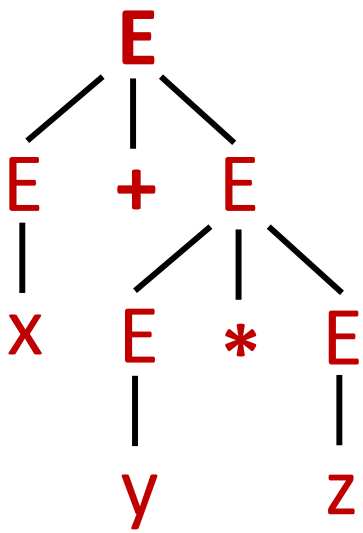
or...
$\rightarrow (E+E)*y$
$\rightarrow (E+x)*y$
$\rightarrow (2+x)*y$



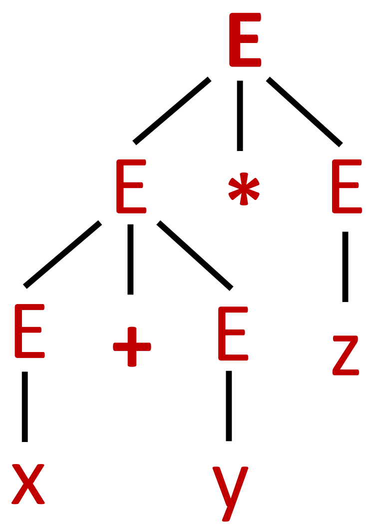
Simple Arithmetic Expressions

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

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



E ⇒ E+E ⇒ x+E ⇒ x+E*E ⇒ x+y*E ⇒ x+y*z
(multiply y with z and then add to x)



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

Induction on Parse Trees

Structural induction is the tool used to prove many more interesting theorems

- General associativity follows from our one rule
  - likewise for generalized De Morgan's laws
- Okay to substitute y for x everywhere in a modular equation when we know that $x \equiv_m y$
- The "Meta Theorem" on set operators

These are proven by induction on parse trees

- parse trees are recursively defined

Two ways to Define Binary Palindromes

Recursively-Defined Set

Recursively-defined sets of strings have the **same power** as grammars

Basis:

ε is a palindrome

any $a \in \{0, 1\}$ is a palindrome

Recursive step:

If p is a palindrome,

then apa is a palindrome for every $a \in \{0, 1\}$

Grammar

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

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
  - define **S** as a tree and then *traverse* it to get a string

We will explore this in HW7

- 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 any set of strings (language) A described by a regular expression, there is a CFG that recognizes A.

Proof idea:

P(A) is "A is recognized by some CFG"

Structural induction based on the recursive definition of regular expressions...

Regular Expressions over Σ

- **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

    A*

CFGs are more general than REs

- CFG to match RE **ε**

  S $\rightarrow \varepsilon$

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

  S $\rightarrow$ a

CFGs are more general than REs

Suppose CFG with start symbol $\mathbf{S_1}$ matches RE **A**
CFG with start symbol $\mathbf{S_2}$ matches RE **B**

- CFG to match RE $\mathbf{A} \cup \mathbf{B}$

$\mathbf{S} \rightarrow \mathbf{S_1} \mid \mathbf{S_2}$ + rules from original CFGs

- CFG to match RE **AB**

$\mathbf{S} \rightarrow \mathbf{S}_1 \mathbf{S}_2$ + rules from original CFGs

CFGs are more general than REs

Suppose CFG with start symbol $\mathbf{S_1}$ matches RE **A**

- CFG to match RE $\mathbf{A}^*$ (= $\boldsymbol{\varepsilon} \cup \mathbf{A} \cup \mathbf{AA} \cup \mathbf{AAA} \cup \ldots$)

$\mathbf{S} \rightarrow \mathbf{S_1}\,\mathbf{S} \mid \varepsilon$ + rules from CFG with $\mathbf{S_1}$

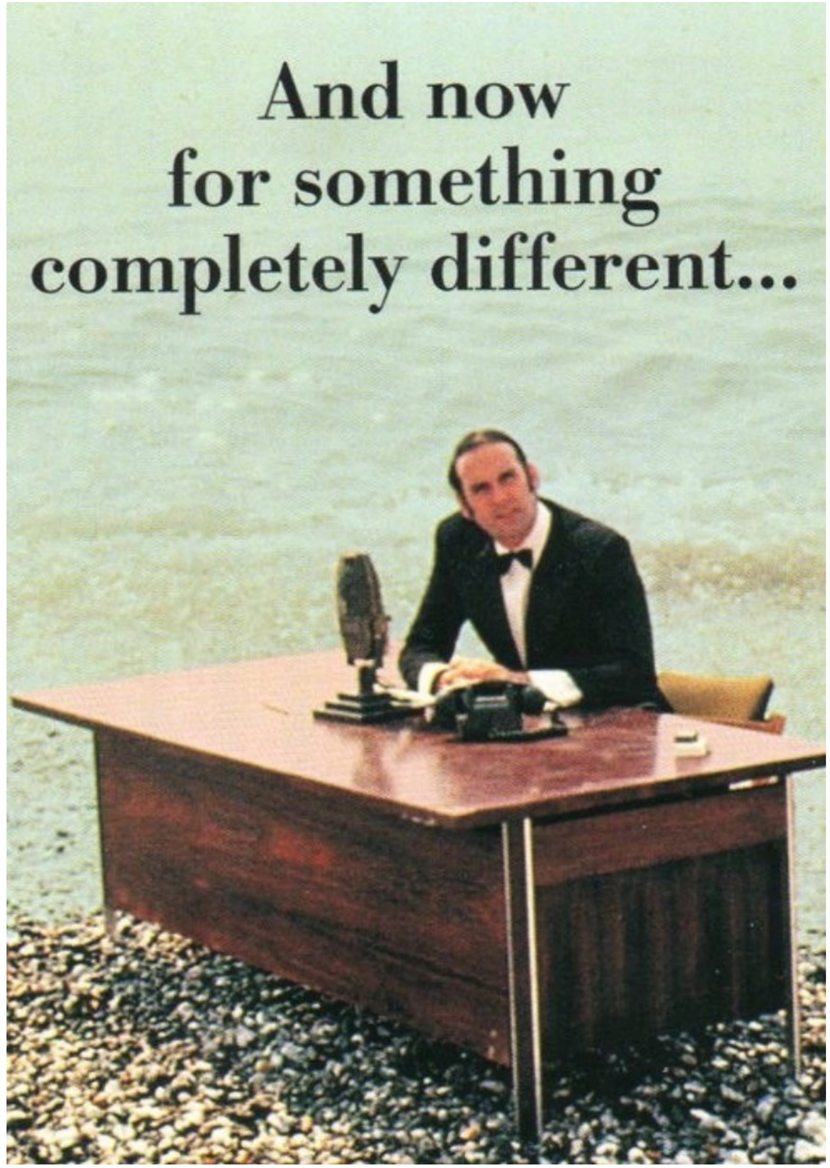
Last time: Languages — REs and CFGs

Saw 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
  - (≈ recursively-defined sets)

We will connect these to machines shortly.

But first, we need some new math terminology....


And now
for something
completely different...

Cartesian Product

We defined Cartesian Product as

$$A \times B := \{(a, b) : a \in A, b \in B\}$$

"The set of all (a, b) such that a ∈ A and b ∈ B"

Can define a <u>subset</u> of pairs satisfying P(a,b)**:**

$$\{(a, b) : \boldsymbol{P(a, b)}, a \in A, b \in B\}$$

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

$R_1 = \{(x, y) : x \equiv_5 y\}$

$R_2 = \{(c_1, c_2) : c_1 \text{ is a prerequisite of } c_2\}$

$R_3 = \{(s, c) : \text{student } s \text{ has taken course } c\}$

$R_4 = \{(a, 1), (a, 2), (b, 1), (b, 3), (c, 3)\}$

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 $\mathbb{N}$:

$<$ on $\mathbb{R}$:

$=$ on Σ^* :

$\subseteq$ on $\mathcal{P}(U)$:

R_2 = $\{(x, y) : x \equiv_5 y\}$:

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

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

$R_2 = \{(x, y) : x \equiv_5 y\}$: 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 $(a, c) \in R$

Combining Relations

Let R be a relation from A to B.
Let S be a relation from B to C.

The composition of R and S, $R \circ S$ is the relation from A to C defined by:

$$R \circ S = \{(a, c) : \exists \text{ b such that } (a, b) \in R \text{ and } (b, c) \in S\}$$

Intuitively, a pair is in the composition if there is a "connection" from the first to the second.

Examples

(a,b) $\in$ Parent iff b is a parent of a

(a,b) $\in$ Sister iff b is a sister of a

When is (x,y) $\in$ Parent $\circ$ Sister?

Aunt

When is (x,y) $\in$ Sister $\circ$ Parent?

Parent $\cap$ HasSister

$R \circ S = \{(a, c) : \exists$ b such that $(a,b) \in R$ and $(b,c) \in S\}$

Examples

Using only the relations Parent, Child, Father, Son, Brother, Sibling, Husband

and **composition**, express the following:

Uncle: b is an uncle of a

Parent ∘ Brother

Cousin: b is a cousin of a

Parent ∘ Sibling ∘ Child

or Parent ∘ (Brother ∪ Sister ∪ ...) ∘ Child

remember that relations are still sets

Powers of a Relation

$$\boldsymbol{R^2} ::= \boldsymbol{R} \circ \boldsymbol{R}$$
$$= \{(\boldsymbol{a}, \boldsymbol{c}) : \exists \boldsymbol{b} \text{ such that } (\boldsymbol{a}, \boldsymbol{b}) \in \boldsymbol{R} \text{ and } (\boldsymbol{b}, \boldsymbol{c}) \in \boldsymbol{R}\}$$

$\boldsymbol{R^0} ::= \{(\boldsymbol{a}, \boldsymbol{a}) : \boldsymbol{a} \in \boldsymbol{A}\}$ "the equality relation on $\boldsymbol{A}$"

$\boldsymbol{R^{n+1}} ::= \boldsymbol{R^n} \circ \boldsymbol{R}$ for $\boldsymbol{n \geq 0}$

e.g., $\boldsymbol{R^1} = \boldsymbol{R^0} \circ \boldsymbol{R} = \boldsymbol{R}$

$\boldsymbol{R^2} = \boldsymbol{R^1} \circ \boldsymbol{R} = \boldsymbol{R} \circ \boldsymbol{R}$

Non-constructive Definitions

Recursively defined sets and functions describe these objects by explaining how to **construct** / compute them

But sets can also be defined non-constructively:

$$S = \{x : P(x)\}$$

How can we define **functions** non-constructively?

– (useful for writing a function specification)

Functions

A function $f : A \rightarrow B$ (A as input and B as output) is a special type of relation.

A **function** f **from** A **to** B is a relation from A to B such that:
for every $a \in A$, there is *exactly one* $b \in B$ with $(a, b) \in f$

I.e., for every input $a \in A$, there is one output $b \in B$.
We denote this b by $f(a)$.

Functions

A function $f : A \to B$ (A as input and B as output) is a special type of relation.

A **function** f **from** A **to** B is a relation from A to B such that: for every $a \in A$, there is *exactly one* $b \in B$ with $(a, b) \in f$

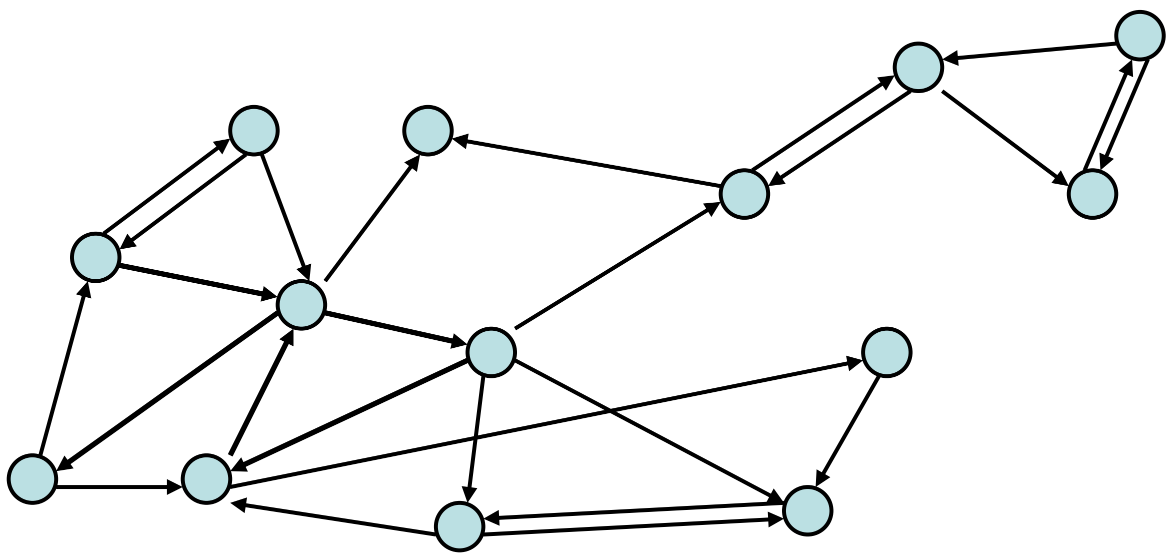
Ex: {((a, b), d) : d is the largest integer dividing a and b}

- **gcd : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$**
- **defined without knowing how to compute it**

(When attempting to define a non-constructively, we sometimes say the function is "**well defined**" if the "*exactly one*" part holds)

Directed Graphs

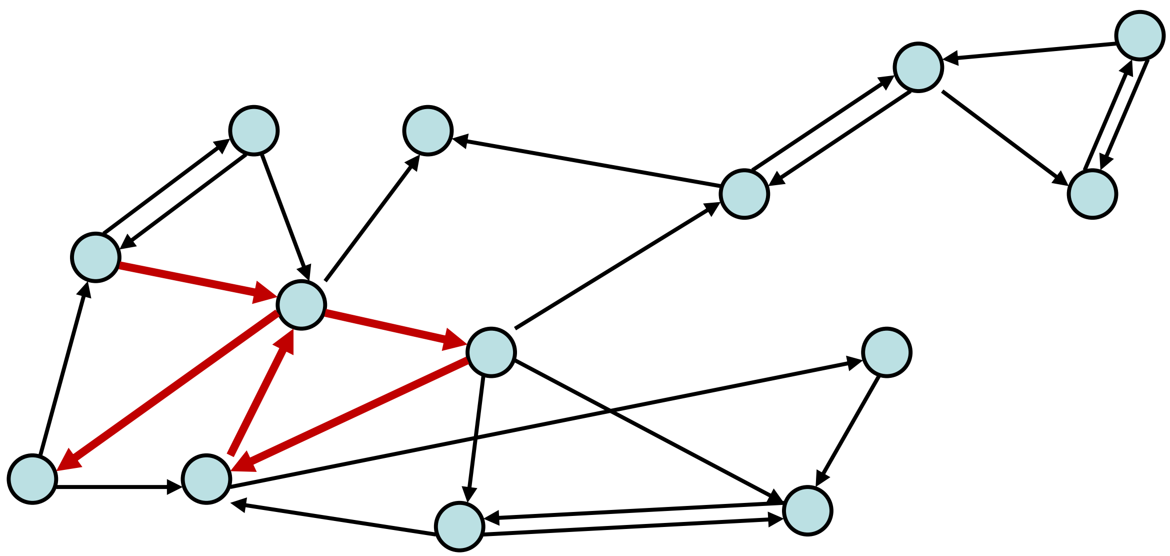
G = (V, E) V – vertices
E – edges (relation on V)



Directed Graphs

G = (V, E)   V – vertices
E – edges   (relation on V)

Path: $v_0, v_1, \ldots, v_k$ with each (v_i, v_{i+1}) in E

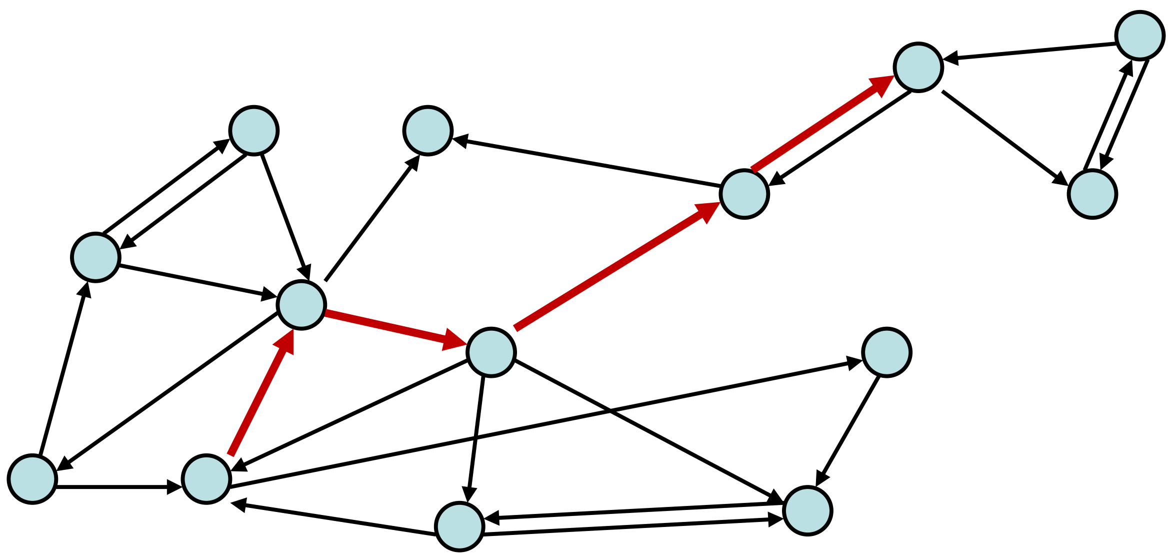


Directed Graphs

G = (V, E) V – vertices
E – edges (relation on V)

Path: $v_0, v_1, \ldots, v_k$ with each (v_i, v_{i+1}) in E

Simple Path: none of $\mathbf{v_0}, \ldots, \mathbf{v_k}$ repeated
Cycle: $\mathbf{v_0} = \mathbf{v_k}$
Simple Cycle: $\mathbf{v_0} = \mathbf{v_k}$, none of $\mathbf{v_1}, \ldots, \mathbf{v_k}$ repeated

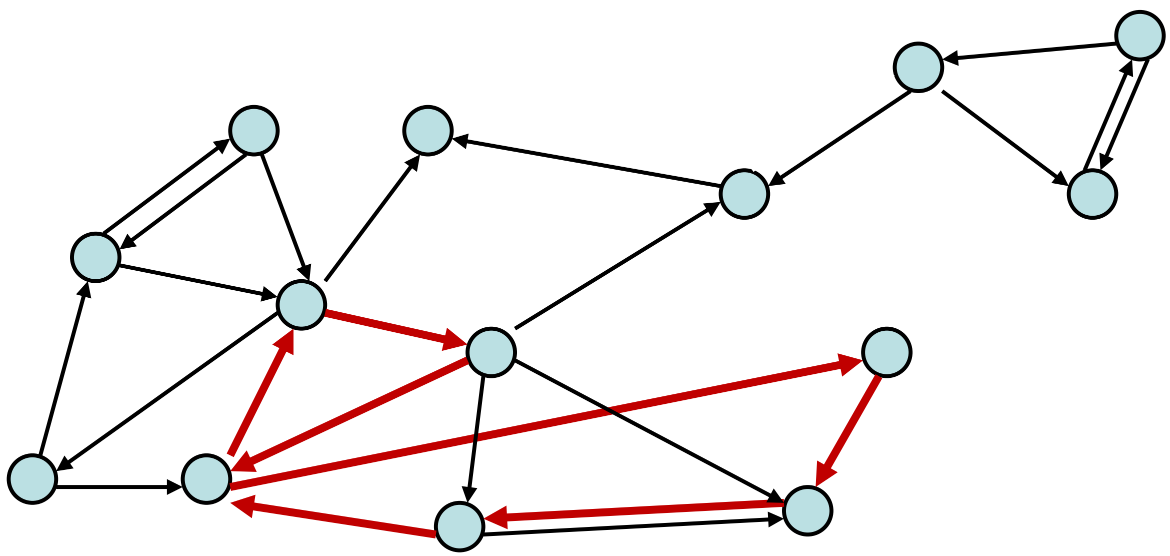


Directed Graphs

G = (V, E)   V – vertices
E – edges   (relation on V)

Path: $v_0, v_1, \ldots, v_k$ with each (v_i, v_{i+1}) in E

Simple Path: none of $\mathbf{v_0}, \ldots, \mathbf{v_k}$ repeated
Cycle: $\mathbf{v_0} = \mathbf{v_k}$
Simple Cycle: $\mathbf{v_0} = \mathbf{v_k}$, none of $\mathbf{v_1}, \ldots, \mathbf{v_k}$ repeated

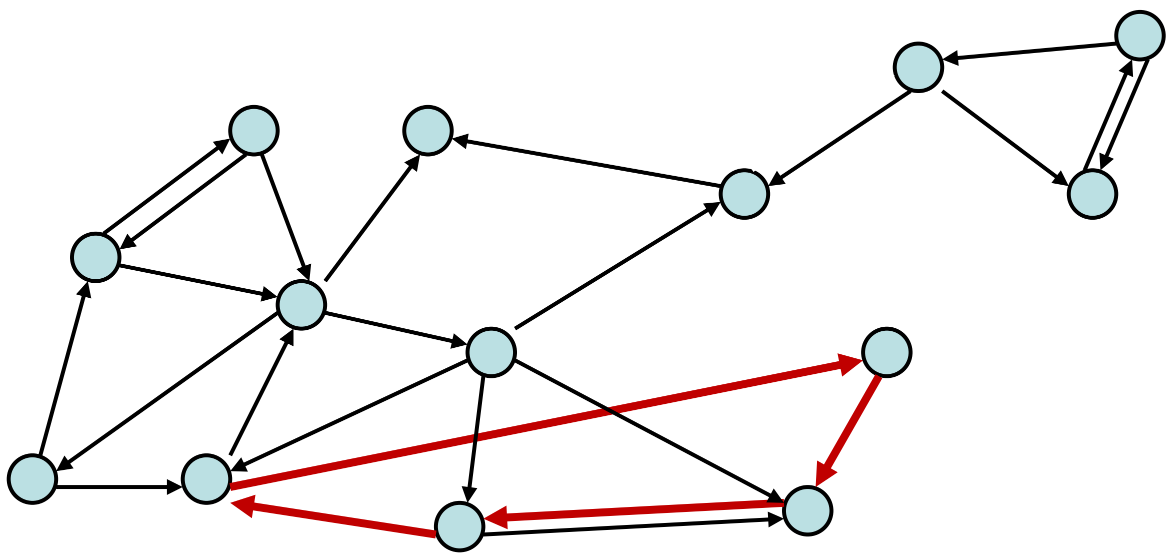


Directed Graphs

G = (V, E) V – vertices
E – edges (relation on V)

Path: $v_0, v_1, \ldots, v_k$ with each (v_i, v_{i+1}) in E

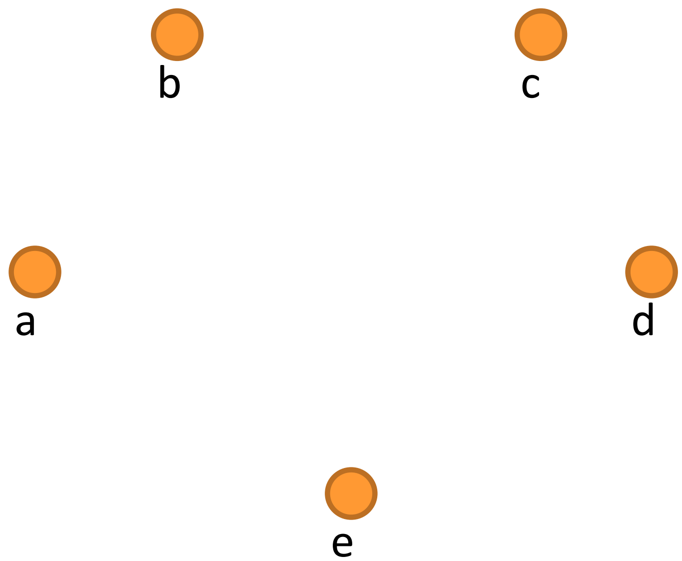
Simple Path: none of $\mathbf{v_0}, \ldots, \mathbf{v_k}$ repeated
Cycle: $\mathbf{v_0} = \mathbf{v_k}$
Simple Cycle: $\mathbf{v_0} = \mathbf{v_k}$, none of $\mathbf{v_1}, \ldots, \mathbf{v_k}$ repeated



Representation of Relations

Directed Graph Representation (Digraph)

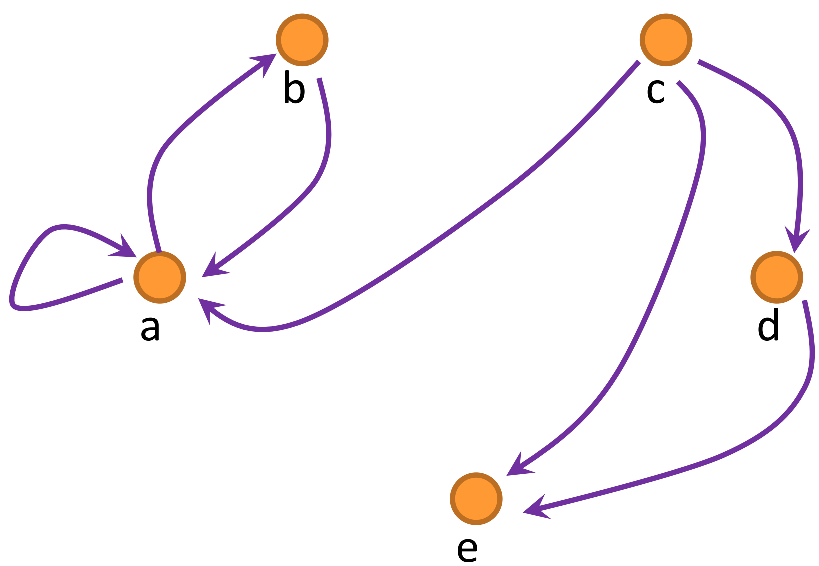
{(a, b), (a, a), (b, a), (c, a), (c, d), (c, e) (d, e) }



Representation of Relations

Directed Graph Representation (Digraph)

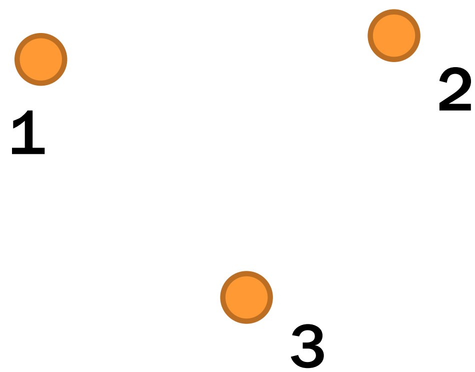
{(a, b), (a, a), (b, a), (c, a), (c, d), (c, e) (d, e) }

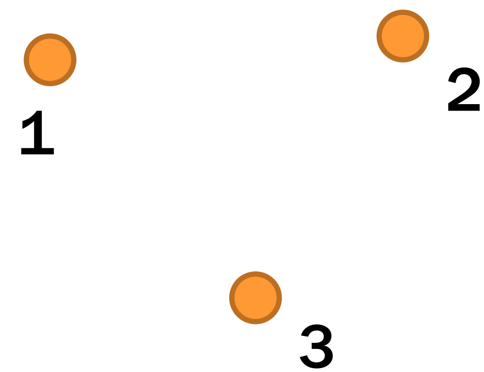


Relational Composition using Digraphs

If $\boldsymbol{S} = \{(2,2),(2,3),(3,1)\}$ and $\boldsymbol{R} = \{(1,2),(2,1),(1,3)\}$

Compute $\boldsymbol{R} \circ \boldsymbol{S}$

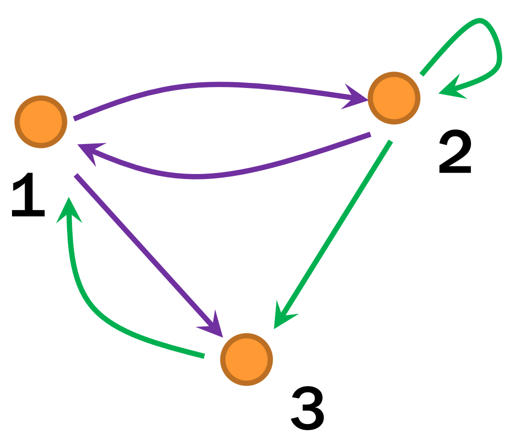


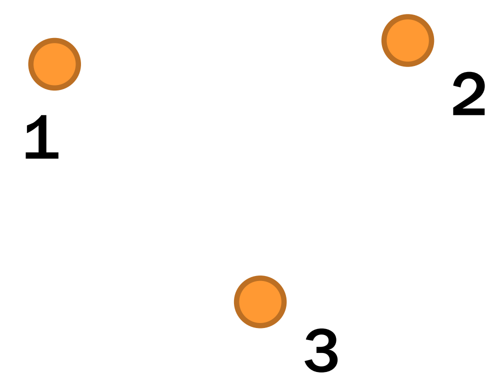


Relational Composition using Digraphs

If $S = \{(2,2),(2,3),(3,1)\}$ and $R = \{(1,2),(2,1),(1,3)\}$

Compute $R \circ S$

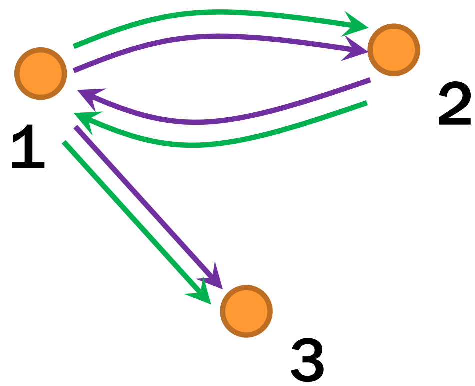


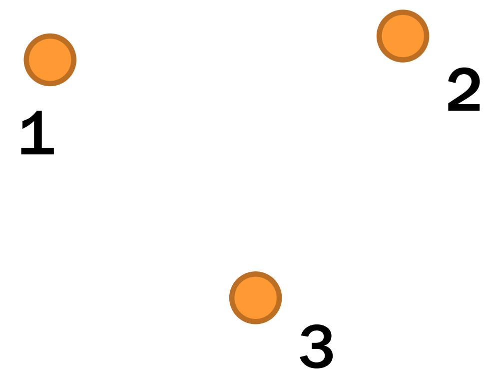


Relational Composition using Digraphs

If $R = \{(1,2),(2,1),(1,3)\}$ and $R = \{(1,2),(2,1),(1,3)\}$

Compute $R \circ R$





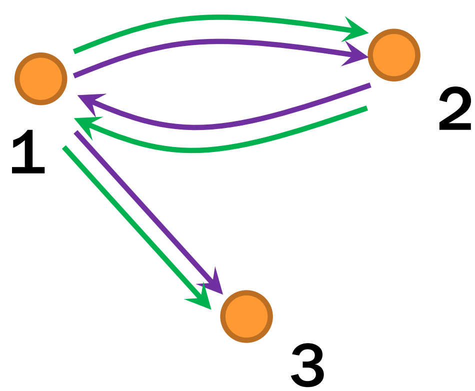
$(a, c) \in R \circ R = R^2$ iff $\exists b\ ((a, b) \in R \land (b, c) \in R)$

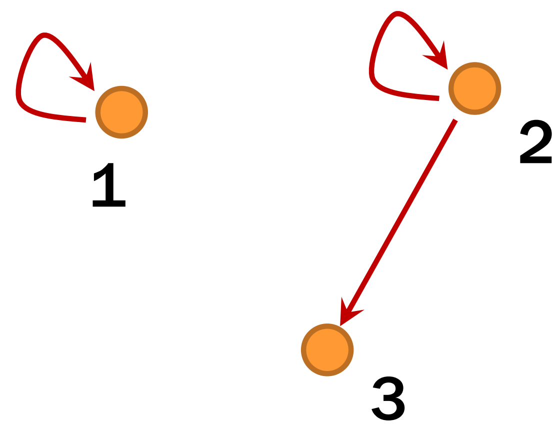
iff $\exists b$ such that a, b, c is a path

Relational Composition using Digraphs

If $\boldsymbol{R} = \{(1,2),(2,1),(1,3)\}$ and $\boldsymbol{R} = \{(1,2),(2,1),(1,3)\}$

Compute $\boldsymbol{R} \circ \boldsymbol{R}$





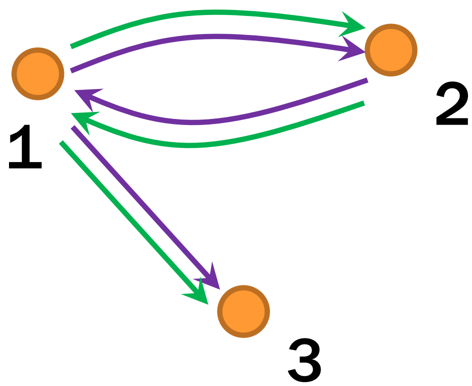
$(a,c) \in R \circ R = R^2$ iff $\exists b\ ((a,b) \in R \wedge (b,c) \in R)$

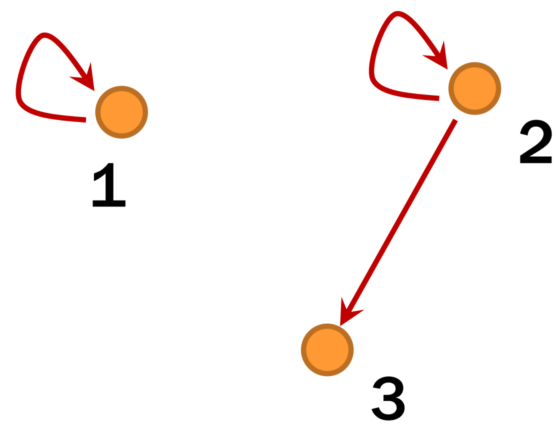
iff $\exists b$ such that a, b, c is a path

Relational Composition using Digraphs

If $R = \{(1,2),(2,1),(1,3)\}$ and $R = \{(1,2),(2,1),(1,3)\}$

Compute $R \circ R$





Special case: $R \circ R$ is paths of length 2.

- R is paths of length 1
- R^0 is paths of length 0 (can't go anywhere)
- $R^3 = R^2 \circ R$ etc, so is R^n paths of length n

Paths in Relations and Graphs

Def: The **length** of a path in a graph is the number of edges in it (counting repetitions if edge used > once).

Let $\boldsymbol{R}$ be a relation on a set $\boldsymbol{A}$. There is a path of length $\boldsymbol{n}$ from **a** to **b** if and only if (**a,b**) $\in \boldsymbol{R}^{\boldsymbol{n}}$

Connectivity In Graphs

Def: Two vertices in a graph are **connected** iff there is a path between them.

Let $\boldsymbol{R}$ be a relation on a set $\boldsymbol{A}$. The **connectivity** relation $\boldsymbol{R}^*$ consists of the pairs (a, b) such that there is a path from a to b in $\boldsymbol{R}$.

$$R^* = \bigcup_{k=0}^{\infty} R^k$$

How Properties of Relations show up in Graphs

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 $(a, c) \in R$

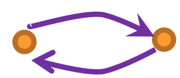
How Properties of Relations show up in Graphs

Let R be a relation on A.

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

 at every node

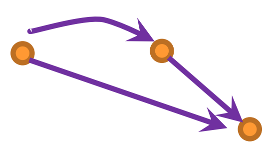
R is **symmetric** iff $(a,b) \in R$ implies $(b, a) \in R$

 or 

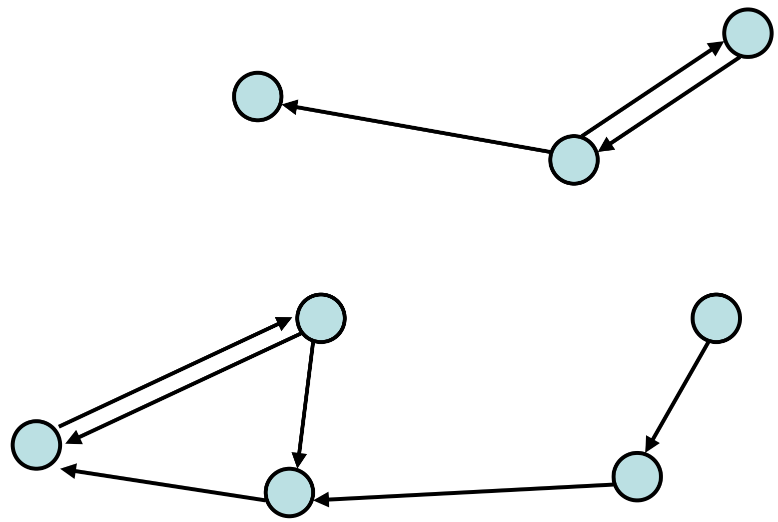
R is **antisymmetric** iff $(a,b) \in R$ and $a \neq b$ implies $(b,a) \notin R$

 or  or 

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

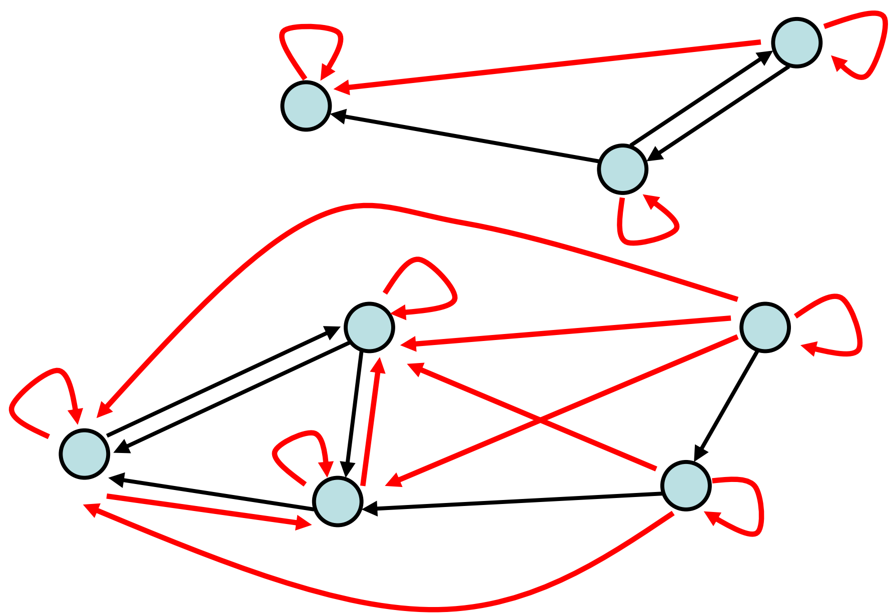


Transitive-Reflexive Closure



Add the **minimum possible** number of edges to make the relation transitive and reflexive.

Transitive-Reflexive Closure



Relation with the **minimum possible** number of **extra edges** to make the relation both transitive and reflexive.

The **transitive-reflexive closure** of a relation $\boldsymbol{R}$ is the connectivity relation $\boldsymbol{R}^*$

Back to Languages