

# CSE 311: Foundations of Computing

---

## Topic 5: Languages



# Strings

---

- An *alphabet*  $\Sigma$  is any finite set of characters
- The set  $\Sigma^*$  of *strings* over the alphabet  $\Sigma$ 
  - example:  $\{0,1\}^*$  is the set of *binary strings*  
0, 1, 00, 01, 10, 11, 000, 001, ... and ""
- $\Sigma^*$  is defined recursively by
  - **Basis:**  $\varepsilon \in \Sigma^*$  ( $\varepsilon$  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  $\Sigma$
  - palindromes over  $\Sigma$
  - 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 Theory is Languages

---

- Any predicate can be phrased as " $x \in S$ "
  - computing " $x \in S$ " 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 " $x \in L$ " for language  $L$ 
  - only restriction is that predicates have `boolean` output
  - but this is usually not a *real* restriction
    - each bit of any output is a T/F value
    - so you computing the individual bits can be phrased as " $x \in S$ "

# **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**

# A Recursively-Defined Set of String

---

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

## **Basis:**

$\varepsilon$  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  $\varepsilon a \cdot p \cdot \varepsilon a$ )

# Regular Expressions

---

## Regular expressions over $\Sigma$

- **Basis:**

$\varepsilon$  is a regular expression (could also include  $\emptyset$ )

$a$  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”

---

$\epsilon$  matches only the **empty string**

$a$  matches only the one-character string  $a$

$A \cup B$  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 ( $\epsilon \cup A \cup AA \cup AAA \cup \dots$ )

Definition of the *language*  
matched by a regular expression

# Language of a Regular Expression

---

The language defined by a regular expression:

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

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

$$L(A \cup B) = L(A) \cup L(B)$$

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

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

$A^n$  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.,  $0^*(1 \cup 1000^*)^*(\epsilon \cup 10)$

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  $s_1, s_2, \dots, s_n$

Construct the regular expression

$$s_1 \cup s_2 \cup \dots \cup s_n$$

(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  $\varepsilon$ :  $L(\varepsilon) = \{\varepsilon\}$ , which is finite

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

Case  $A \cup B$ :

$$L(A \cup B) = L(A) \cup L(B)$$

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  $AB$ :**

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

**By the IH,  $L(A)$  and  $L(B)$  are finite.**

**Every element of  $L(AB)$  is covered by a pair  $(y, z)$  where  $y \in L(A)$  and  $z \in L(B)$ , so  $L(AB)$  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:**  $S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$

**How does this grammar generate 0110?**

## Example Context-Free Grammars

---

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

How does this grammar generate 0110?

$S \rightarrow 0S0 \rightarrow 01S10 \rightarrow 01\varepsilon10 = 0110$

## 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

## Example Context-Free Grammars

---

**Example:**      $S \rightarrow A \mid B$   
                   $A \rightarrow 0A \mid \varepsilon$   
                   $B \rightarrow 1B \mid \varepsilon$

**How does this grammar generate 000?**

# Example Context-Free Grammars

---

**Example:**      $S \rightarrow A \mid B$   
                   $A \rightarrow 0A \mid \varepsilon$   
                   $B \rightarrow 1B \mid \varepsilon$

**How does this grammar generate 000?**

$S \rightarrow A \rightarrow 0A \rightarrow 00A \rightarrow 000A \rightarrow 000\varepsilon = 000$

# Example Context-Free Grammars

---

**Example:**      $S \rightarrow A \mid B$   
                   $A \rightarrow 0A \mid \varepsilon$   
                   $B \rightarrow 1B \mid \varepsilon$

**How to describe all strings generated?**

strings of all 0s or all 1s

(all 0s)  $\cup$  (all 1s)

# 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  $\Sigma$  of *terminal symbols* that can't be replaced
  - One variable, usually  $\mathbf{S}$ , is called the *start symbol*
- The substitution rules involving a variable  $\mathbf{A}$ , 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  $\mathbf{xAy} \Rightarrow \mathbf{xwy}$
  - 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:  $S \rightarrow 0S \mid S1 \mid \varepsilon$

# Example Context-Free Grammars

---

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

$0^*1^*$

# 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)

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

# 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)

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

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)

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

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

$$S \rightarrow 0S11 \mid \varepsilon$$

# 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)

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

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)

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

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

$$S \rightarrow A10$$

$$A \rightarrow 0A1 \mid \varepsilon$$

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

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?

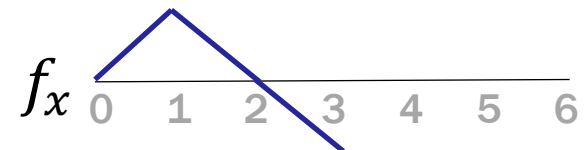
# Example Context-Free Grammars

---

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"	$1 - 2 = -1$



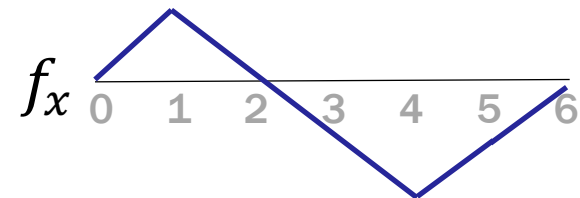
# Example Context-Free Grammars

---

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"	$3 - 3 = 0$

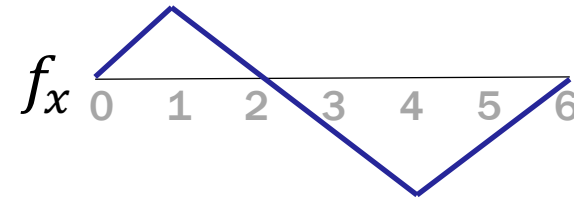


# Example Context-Free Grammars

---

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  $x$ .

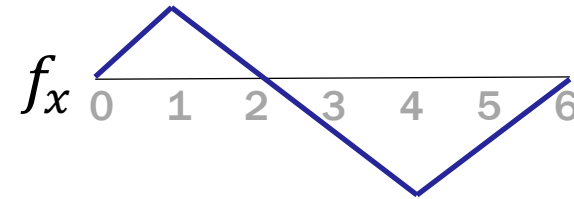
$f_x(k) = 0$  when first  $k$  characters have #0s = #1s

# 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                       $f(0) = 0$

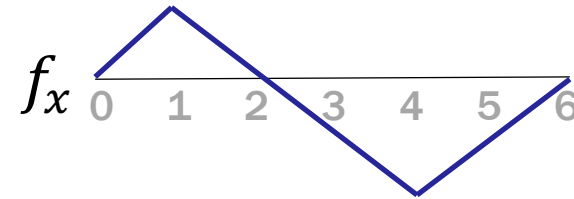
– ends at 0                               $f(n) = 0$

# Example Context-Free Grammars

---

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  $f_x(k) = f_x(k - 1) - 1$

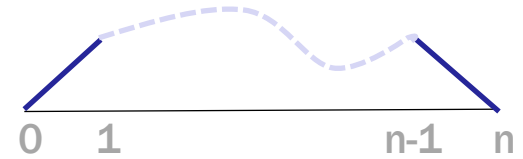
# Example Context-Free Grammars

---

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

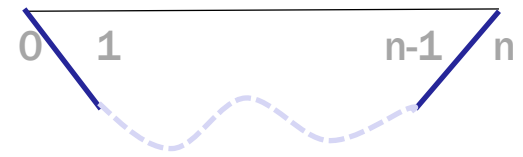
- $f_x(k) > 0$  for all such  $k$

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



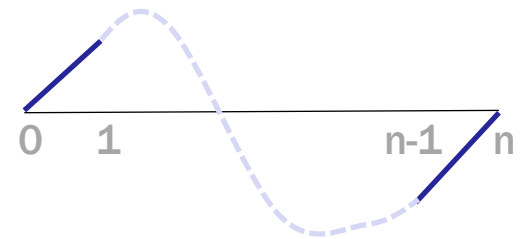
- $f_x(k) < 0$  for all such  $k$

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



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

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



# Parse Trees

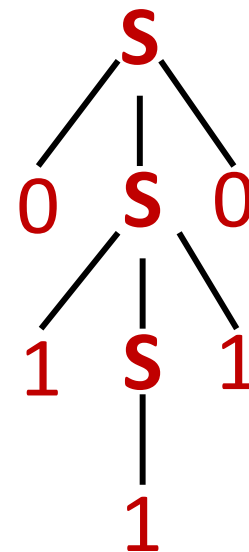
---

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  $01110$



# 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

### **Basis:**

$\varepsilon$  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  $a \in \{0, 1\}$

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

Grammar

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

# 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
- 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 $\Sigma$

---

- **Basis:**

- $\epsilon$  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 are more general than REs

---

- CFG to match RE  $\epsilon$

$$S \rightarrow \epsilon$$

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

$$S \rightarrow a$$

# CFGs are more general than REs

---

Suppose CFG with start symbol  $S_1$  matches RE **A**

CFG with start symbol  $S_2$  matches RE **B**

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

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

- CFG to match RE **AB**

$S \rightarrow S_1 S_2$  + rules from original CFGs

# CFGs are more general than REs

---

Suppose CFG with start symbol  $S_1$  matches RE  $A$

- CFG to match RE  $A^*$  ( $= \varepsilon \cup A \cup AA \cup AAA \cup \dots$ )

$S \rightarrow S_1 S \mid \varepsilon$

+ rules from CFG with  $S_1$

# Last time: Languages — REs and CFGs

---

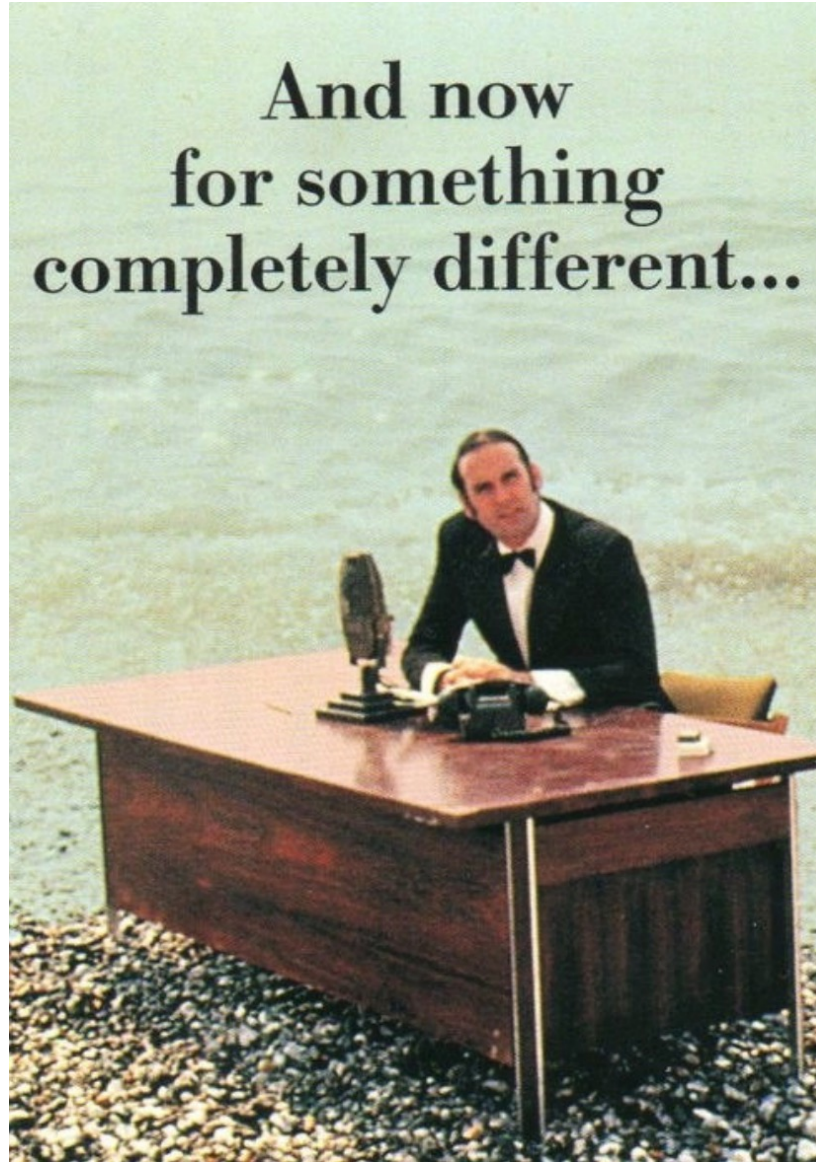
Saw two new ways of defining languages

- **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
  - ( $\approx$  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

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

“The set of all  $(a, b)$  such that  $a \in A$  and  $b \in B$ ”

Can define a subset of pairs satisfying  $P(a,b)$ :

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

# 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  $\Sigma^*$

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  $\Sigma^*$  :

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

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

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

$R$  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  $\Sigma^*$  : Reflexive, Symmetric, Antisymmetric, Transitive

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

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

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

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

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

# Examples

---

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

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

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

Aunt

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

Parent  $\cap$  HasSister

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

# 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

---

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

$$R^0 := \{(a, a) : a \in A\} \quad \text{“the equality relation on } A\text{”}$$

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

$$\begin{aligned} \text{e.g., } R^1 &= R^0 \circ R = R \\ R^2 &= R^1 \circ R = R \circ R \end{aligned}$$

# 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  $f(a)$ .

# Recall: Greatest Common Divisor

---

## GCD Theorem

For  $a, b$  with  $a > 0$

there exist a *unique* integer  $n$  s.t.  $n \mid a$  and  $n \mid b$   
and, for all  $d$ , if  $d \mid a$  and  $d \mid b$ , then  $d \leq n$

We will denote this unique number as  $n = \gcd(a, b)$

# Recall: Division Theorem

---

## Division Theorem

For  $a, b$  with  $b > 0$

there exist *unique* integers  $q, r$  with  $0 \leq r < b$   
such that  $a = qb + r$ .

To put it another way, if we divide  $b$  into  $a$ , we get a  
unique quotient  $q = a \text{ div } b$   
and non-negative remainder  $r = a \text{ mod } b$

# 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  $(a, b) \in f$

$$D = \{((a, b), q) : \exists r ((a = qb+r) \wedge (0 \leq r < m))\}$$

When attempting to define non-constructively, we say the function is “**well defined**” if the “*exactly one*” part holds.

# Directed Graphs

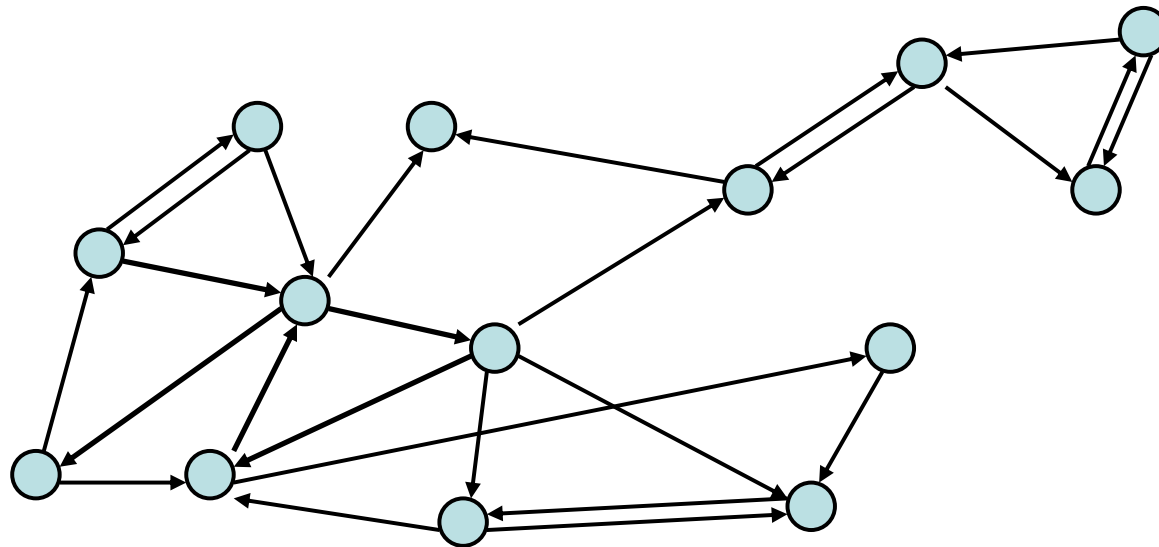
---

$G = (V, E)$

$V$  – vertices

$E$  – edges

(relation on  $V$ )





# Directed Graphs

---

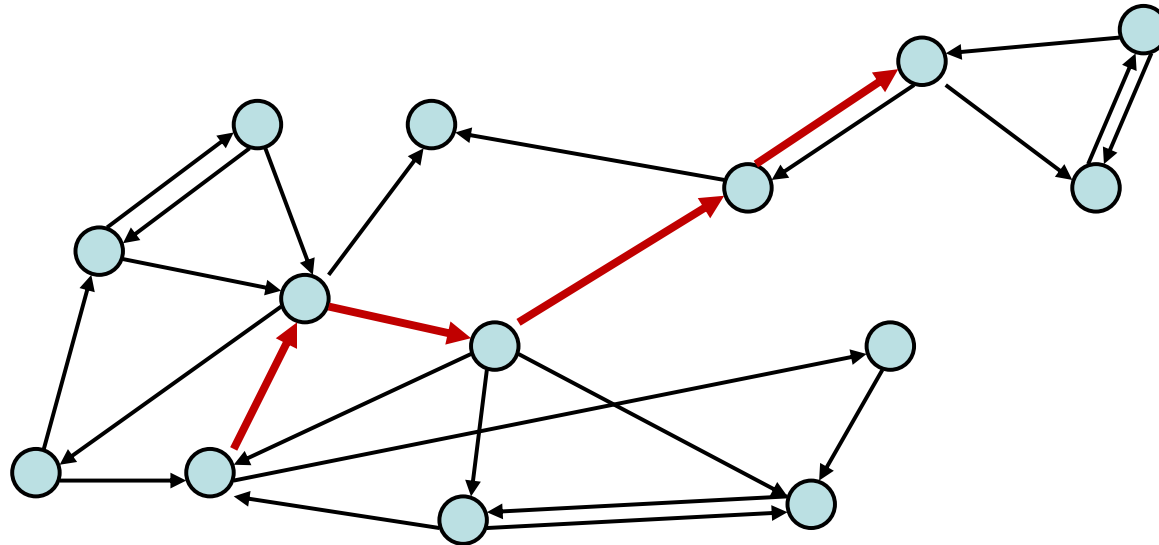
$G = (V, E)$        $V$  – vertices  
                          $E$  – edges      (relation on  $V$ )

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

**Simple Path:** none of  $v_0, \dots, v_k$  repeated

**Cycle:**  $v_0 = v_k$

**Simple Cycle:**  $v_0 = v_k$ , none of  $v_1, \dots, v_k$  repeated



# Directed Graphs

---

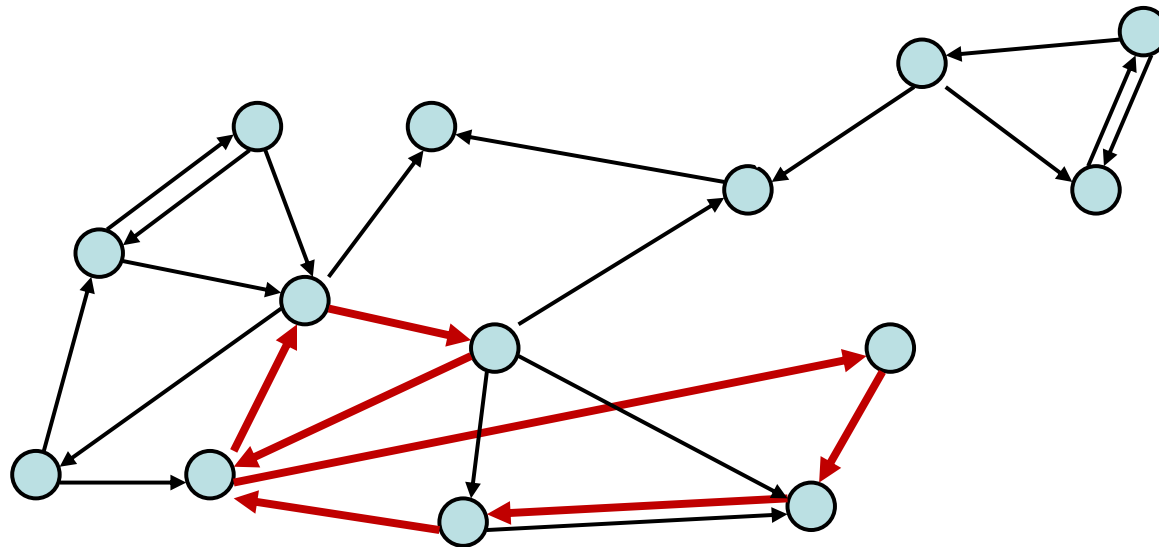
$G = (V, E)$        $V$  – vertices  
                          $E$  – edges      (relation on  $V$ )

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

**Simple Path:** none of  $v_0, \dots, v_k$  repeated

**Cycle:**  $v_0 = v_k$

**Simple Cycle:**  $v_0 = v_k$ , none of  $v_1, \dots, v_k$  repeated



# Directed Graphs

---

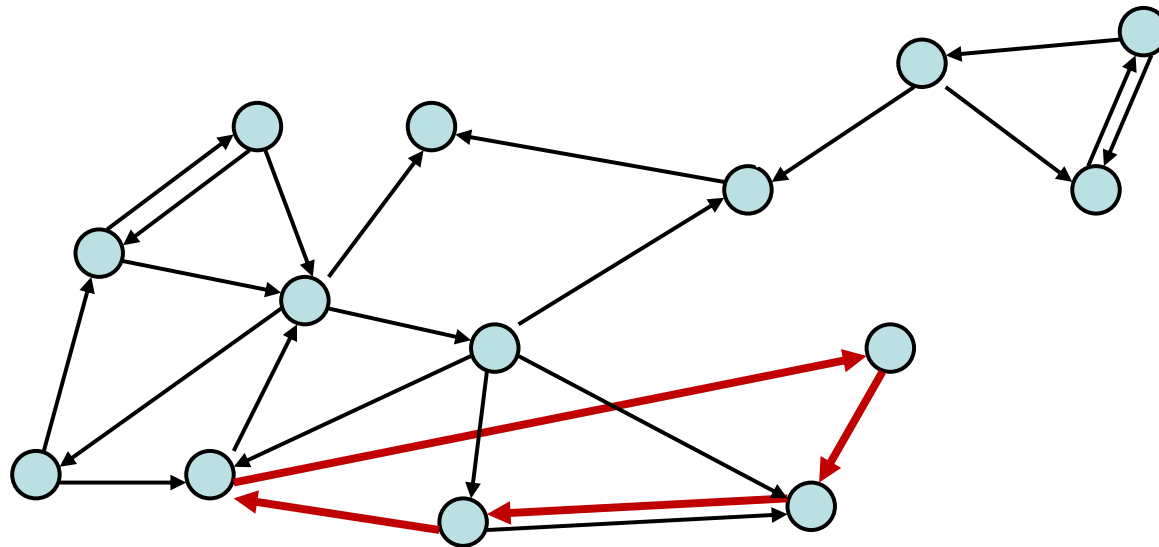
$G = (V, E)$        $V$  – vertices  
                          $E$  – edges      (relation on  $V$ )

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

**Simple Path:** none of  $v_0, \dots, v_k$  repeated

**Cycle:**  $v_0 = v_k$

**Simple Cycle:**  $v_0 = v_k$ , none of  $v_1, \dots, v_k$  repeated

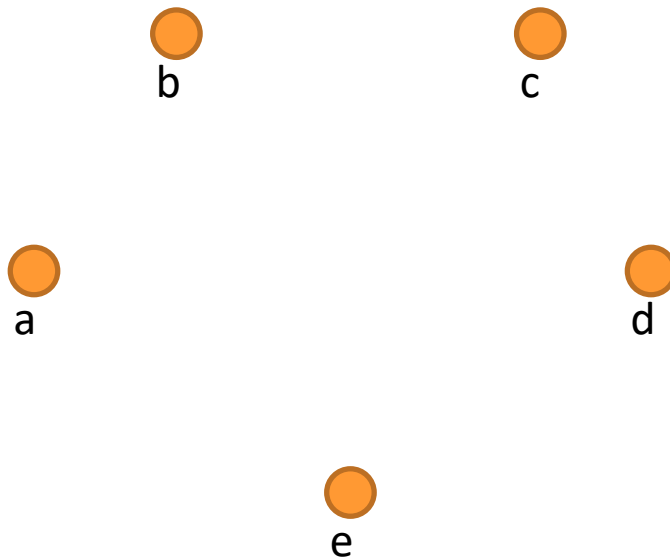


# Representation of Relations

---

## Directed Graph Representation (Digraph)

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

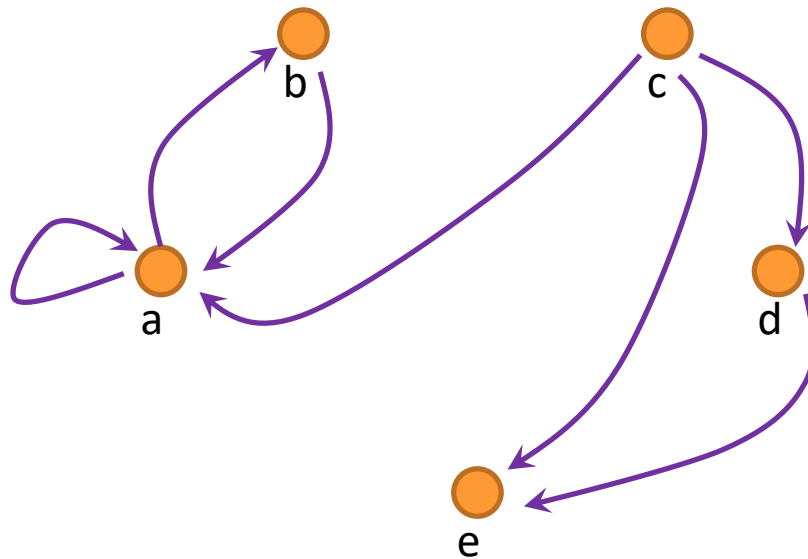


# Representation of Relations

---

## Directed Graph Representation (Digraph)

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



# Relational Composition using Digraphs

---

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

Compute  $R \circ S$

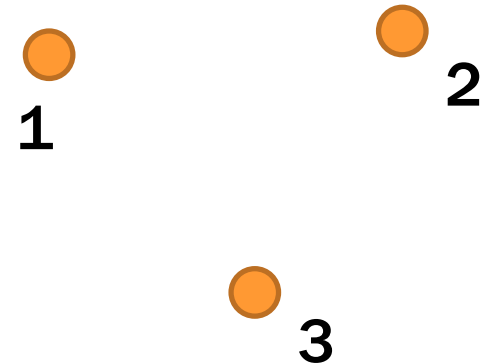
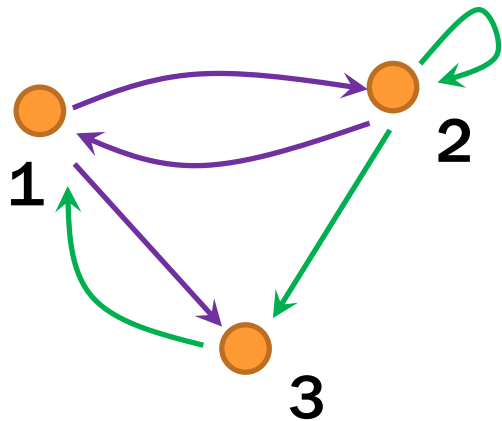


# Relational Composition using Digraphs

---

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

Compute  $R \circ S$

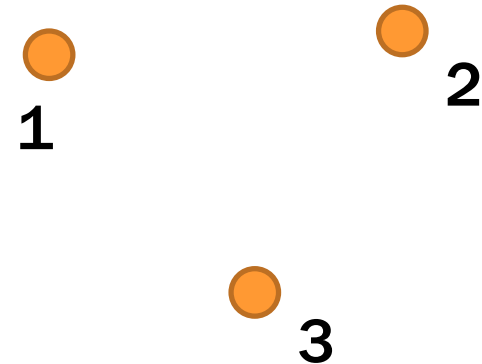
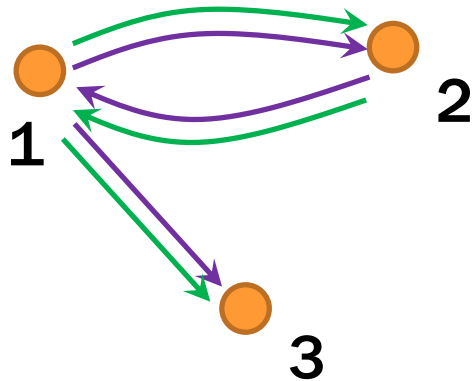


# Relational Composition using Digraphs

---

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

Compute  $R \circ R$



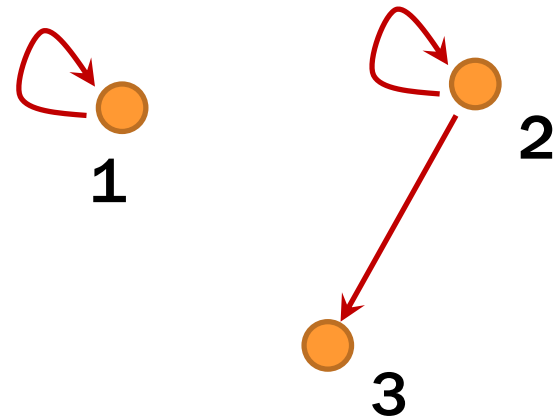
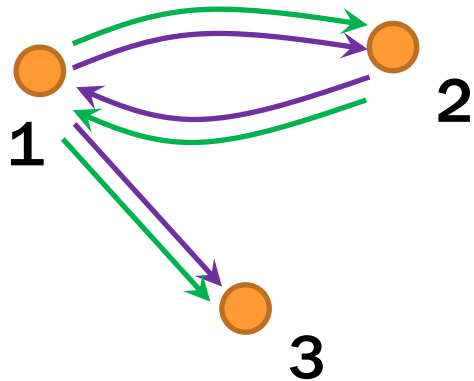
$(a, c) \in R \circ R = R^2$  iff  $\exists b ((a, b) \in R \wedge (b, c) \in R)$   
iff  $\exists b$  such that  $a, b, c$  is a path

# Relational Composition using Digraphs

---

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

Compute  $R \circ R$



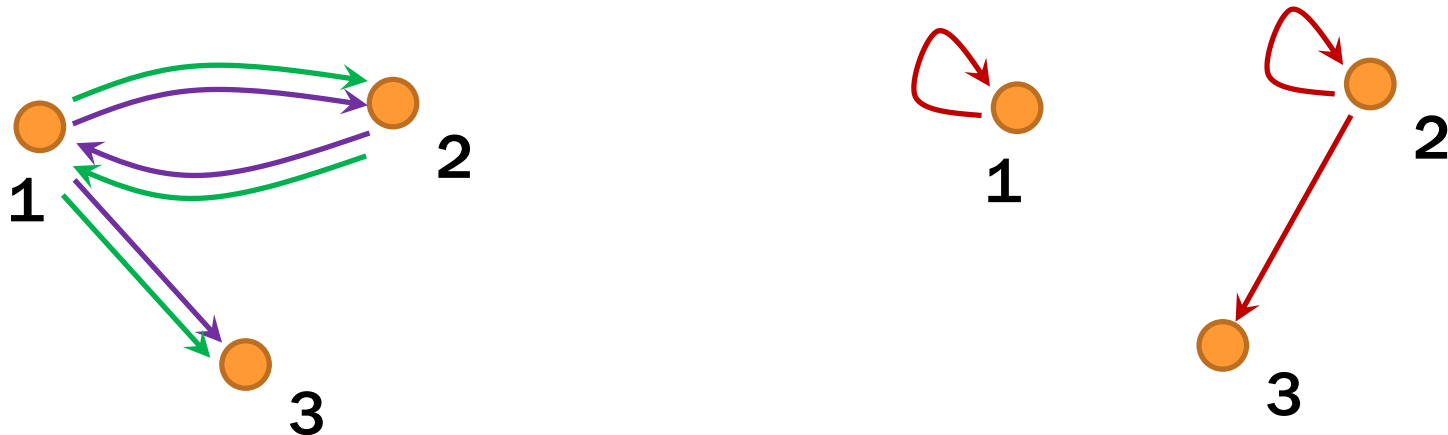
$(a, c) \in R \circ R = R^2$  iff  $\exists b ((a, b) \in R \wedge (b, c) \in R)$   
iff  $\exists b$  such that  $a, b, c$  is a path

# 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

# 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

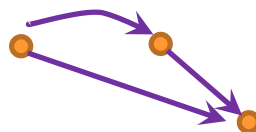
$R$  is **symmetric** iff  $(a,b) \in R$  implies  $(b,a) \in R$



$R$  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  $(a,c) \in R$



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

# Connectivity In Graphs

---

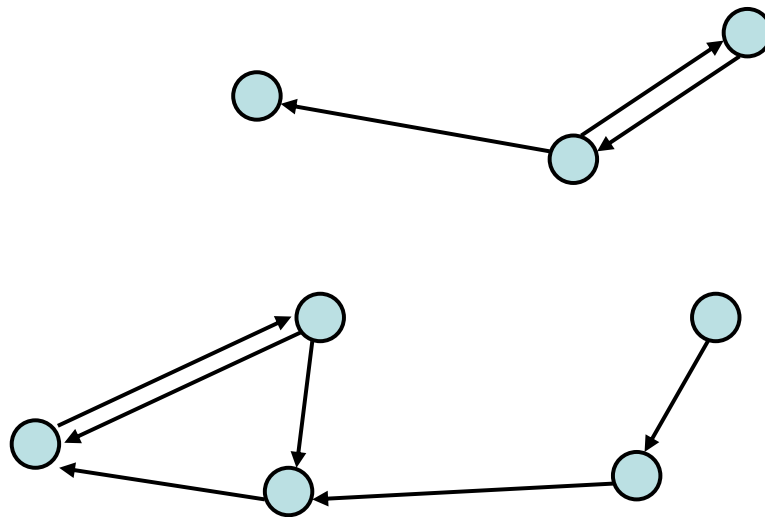
**Def:** Two vertices in a graph are **connected** iff there is a path between them.

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

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

# Transitive-Reflexive Closure

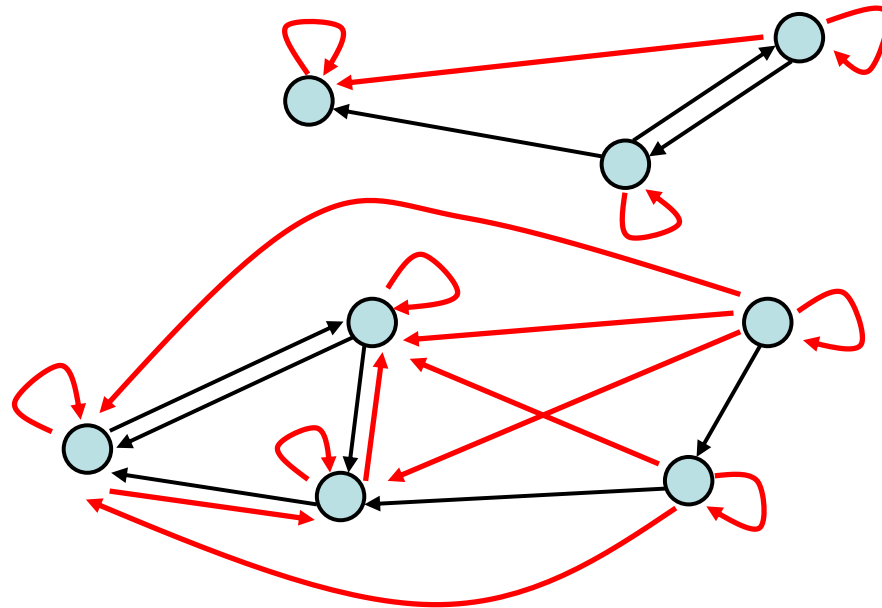
---



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  $R$  is the connectivity relation  $R^*$

## Back to Languages

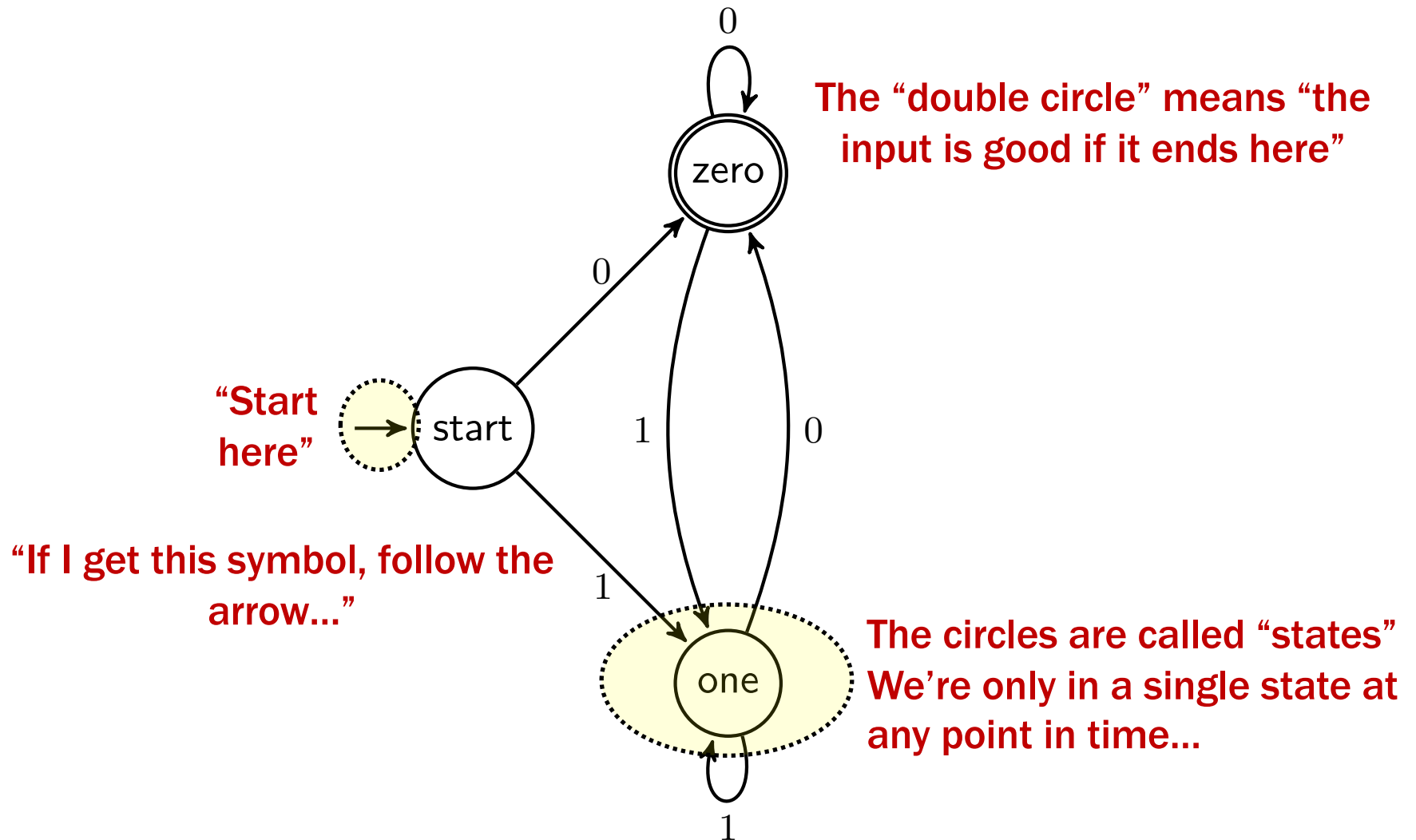
---



**AND NOW BACK TO  
OUR REGULARLY  
SCHEDULED  
PROGRAMMING**

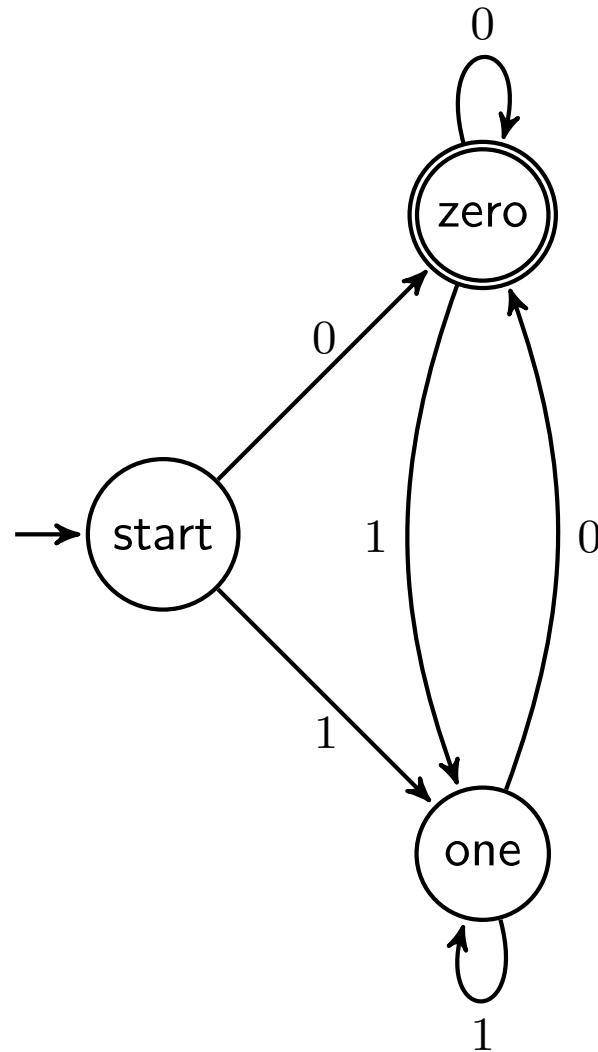
# Finite State Machines

---



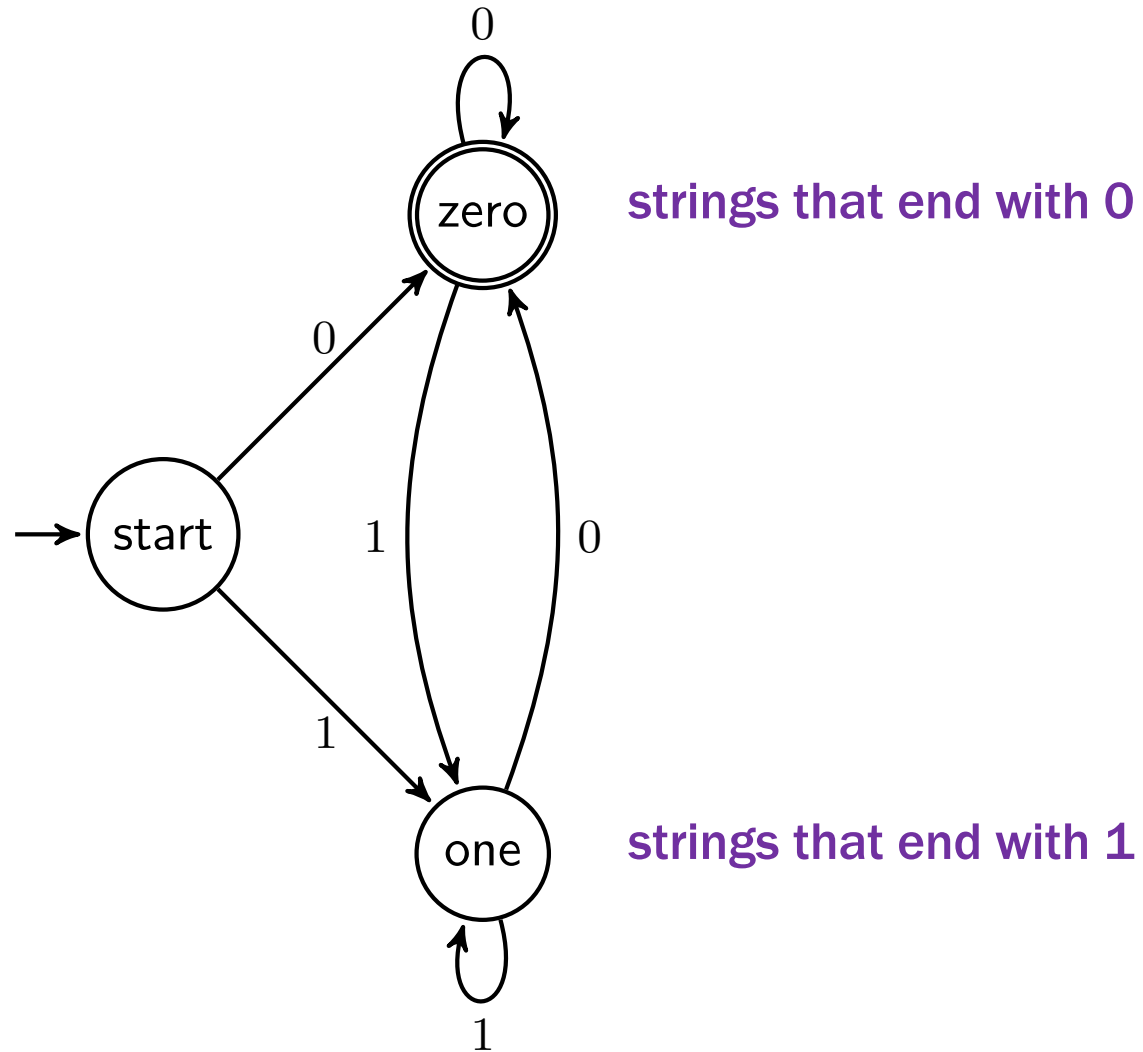
# Where does this machine do on "0110"?

---



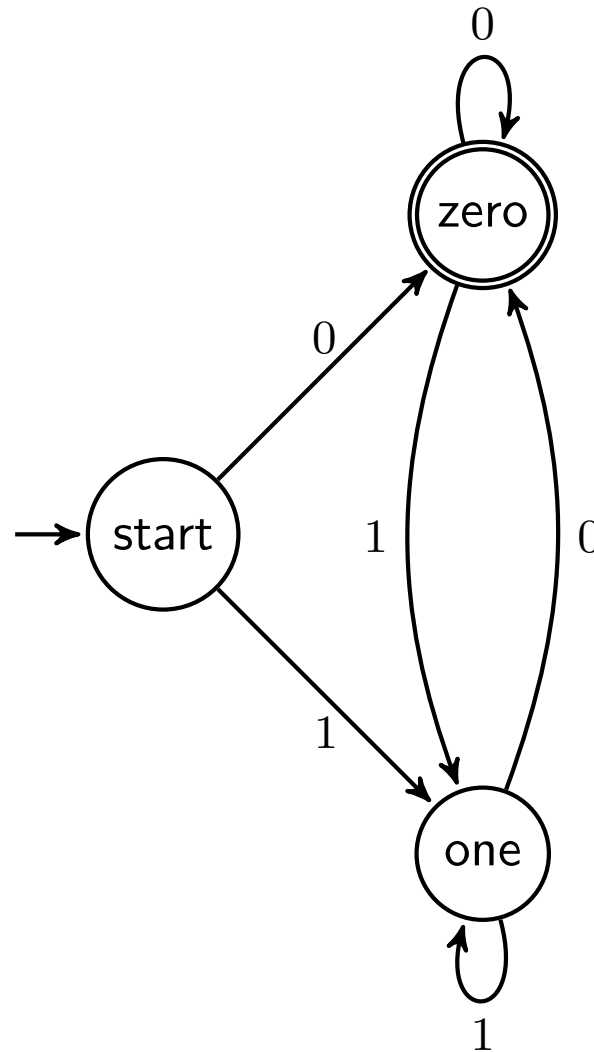
# Which strings reach each state?

---



# Which strings does this machine say are OK?

---



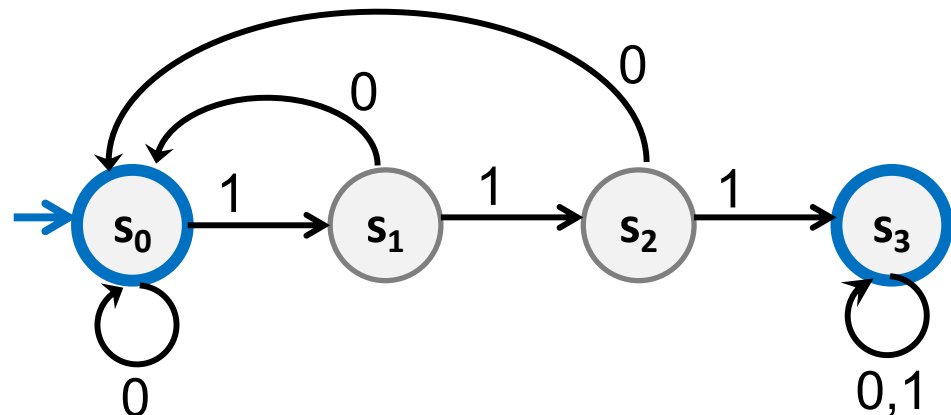
The set of all binary strings that end in 0

# Finite State Machines

---

- States
- Transitions on input symbols
- Start state and final states
- The “language recognized” by the machine is the set of strings that reach a final state from the start

Old State	0	1
$s_0$	$s_0$	$s_1$
$s_1$	$s_0$	$s_2$
$s_2$	$s_0$	$s_3$
$s_3$	$s_3$	$s_3$

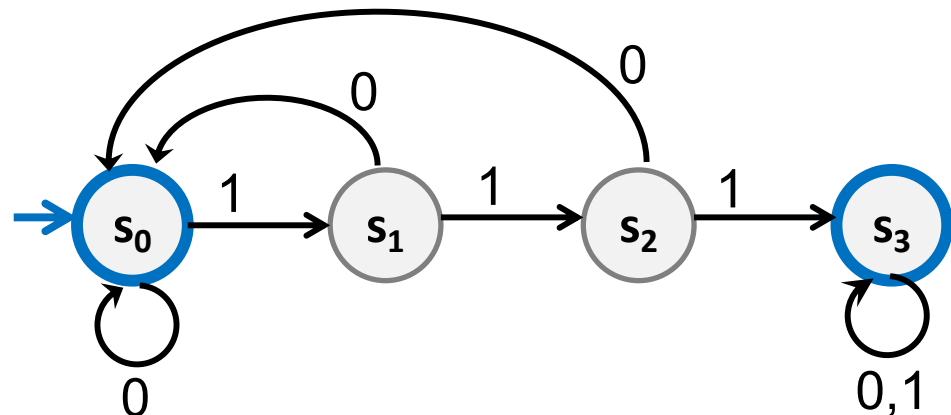


# Finite State Machines

---

- Each machine designed for strings over some fixed alphabet  $\Sigma$ .
- Must have a transition defined from each state for **every** symbol in  $\Sigma$ .

Old State	0	1
$s_0$	$s_0$	$s_1$
$s_1$	$s_0$	$s_2$
$s_2$	$s_0$	$s_3$
$s_3$	$s_3$	$s_3$



# What strings reach each state?

---

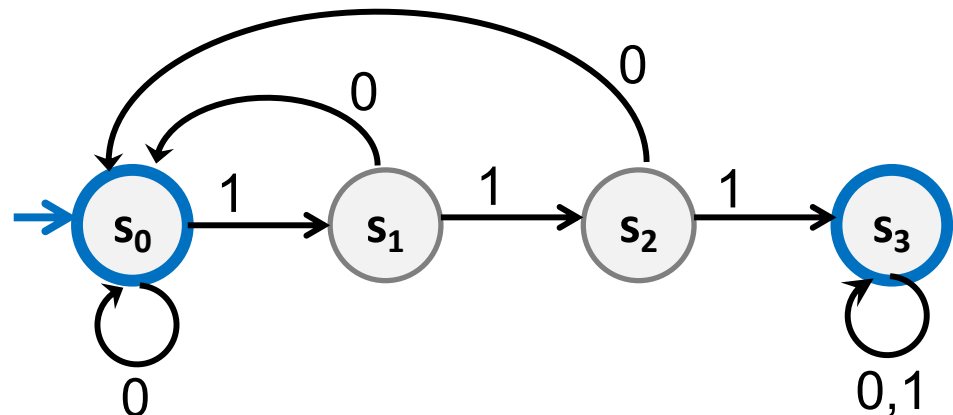
$S_0$  strings that end with 0 (or  $\epsilon$ )

$S_1$  strings that end with 1

$S_2$  strings that end with 11

$S_3$  strings that contain 111

Old State	0	1
$S_0$	$S_0$	$S_1$
$S_1$	$S_0$	$S_2$
$S_2$	$S_0$	$S_3$
$S_3$	$S_3$	$S_3$

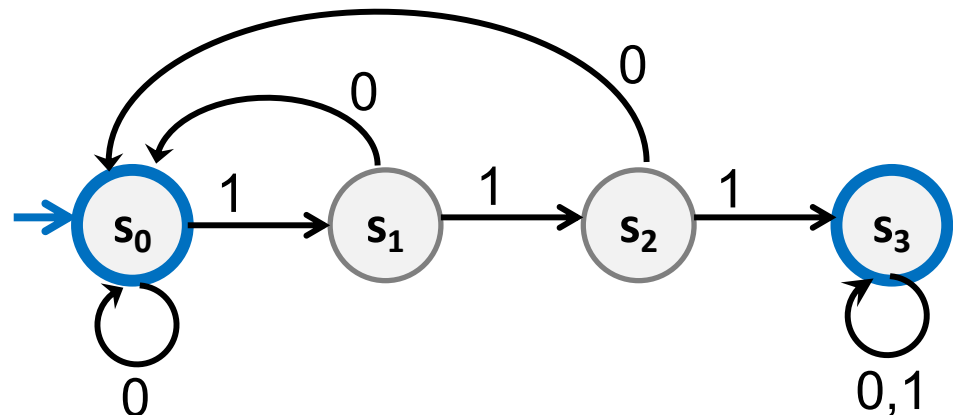


# What language does this machine recognize?

---

The set of all binary strings that contain **111** or end with **0** or are  $\epsilon$

Old State	0	1
$s_0$	$s_0$	$s_1$
$s_1$	$s_0$	$s_2$
$s_2$	$s_0$	$s_3$
$s_3$	$s_3$	$s_3$



# **Applications of FSMs (a.k.a. Finite Automata)**

---

- **Implementation of regular expression matching in programs like `grep`**
- **Control structures for sequential logic in digital circuits**
- **Algorithms for communication and cache-coherence protocols**
  - **Each agent runs its own FSM**
- **Design specifications for reactive systems**
  - **Components are communicating FSMs**

# **Applications of FSMs (a.k.a. Finite Automata)**

---

- **Formal verification of systems**
  - Is an unsafe state reachable?
- **Computer games**
  - FSMs implement non-player characters
- **Minimization algorithms for FSMs can be extended to more general models used in**
  - Text prediction
  - Speech recognition

# State Machine Design Recipe

---

Given a language, how do you design a state machine for it?

Need enough states to:

- Decide whether to accept or reject at the end
- Update the state on each new character

## Strings over $\{0, 1, 2\}$

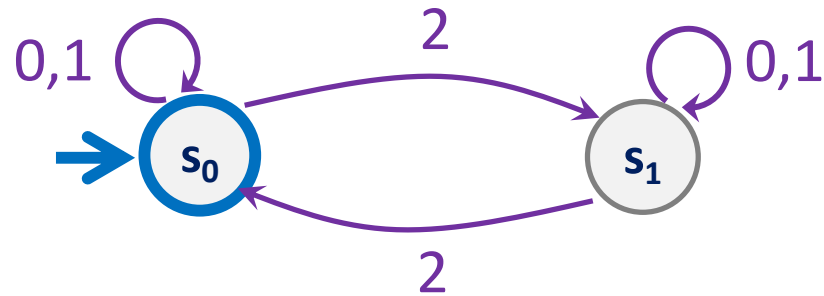
---

$M_1$ : Strings with an even number of 2's

# Strings over $\{0, 1, 2\}$

---

$M_1$ : Strings with an even number of 2's



# State Machine Design Recipe

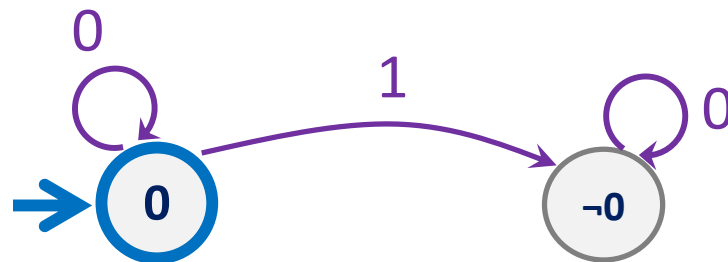
---

$M_2$ : Strings where the sum of digits mod 3 is 0

Can we get away with two states?

- One for 0 mod 3 and one for everything else

This would be enough to decide at the end!



where does this go on a 1?

# State Machine Design Recipe

---

$M_2$ : Strings where the sum of digits mod 3 is 0

Can we get away with two states?

- One for 0 mod 3 and one for everything else

This would be enough to decide at the end!

But can't update the state on each new character:

- If you're in the "not 0 mod 3" state, and the next character is 1, which state should you go to?

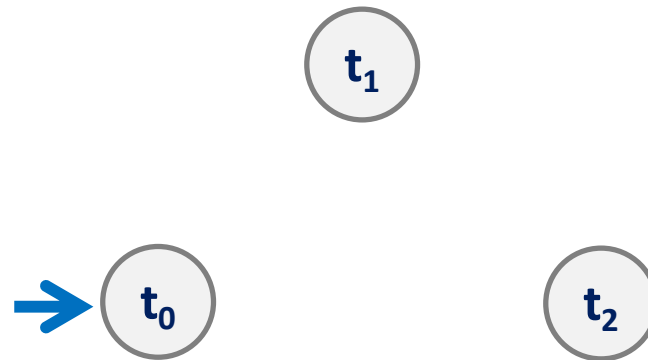
# State Machine Design Recipe

---

$M_2$ : Strings where the sum of digits mod 3 is 0

So, we need three states:

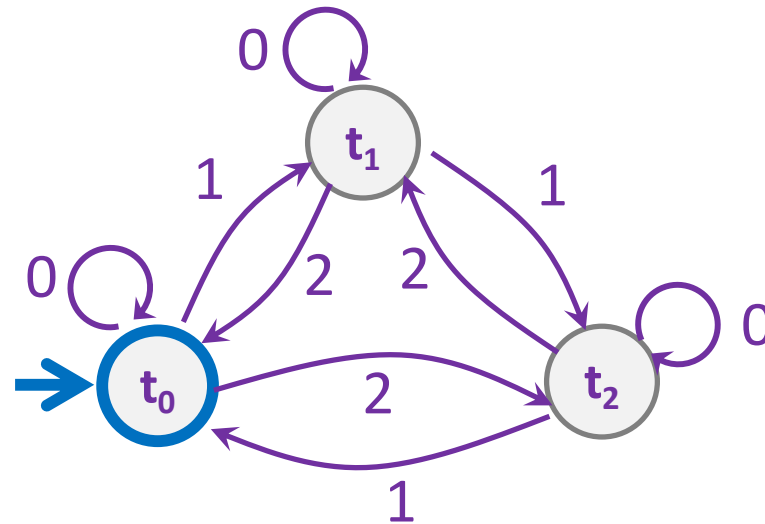
sum of digits mod 3 is 0, 1, or 2



# Strings over $\{0, 1, 2\}$

---

$M_2$ : Strings where the sum of digits mod 3 is 0



# FSM as abstraction of Java code

---

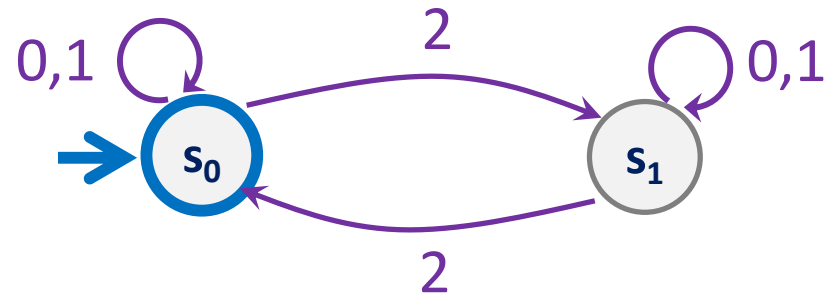
```
boolean sumCongruentToZero(String str) {
    int sum = 0;
    for (int i = 0; i < str.length(); i++) {
        if (str.charAt(i) == '2')
            sum = (sum + 2) % 3;
        if (str.charAt(i) == '1')
            sum = (sum + 1) % 3;
        if (str.charAt(i) == '0')
            sum = (sum + 0) % 3;
    }
    return sum == 0;
}
```

FSMs can model Java code with a finite number of fixed-size variables that makes one pass through input

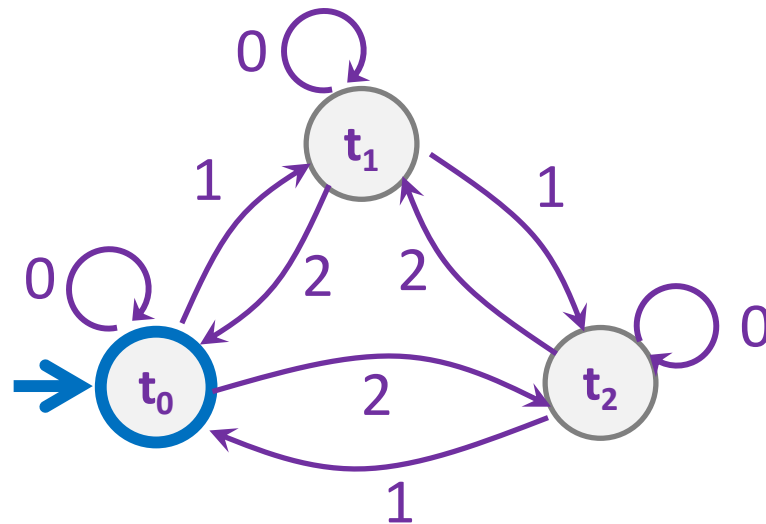
# Strings over $\{0, 1, 2\}$

---

$M_1$ : Strings with an even number of 2's

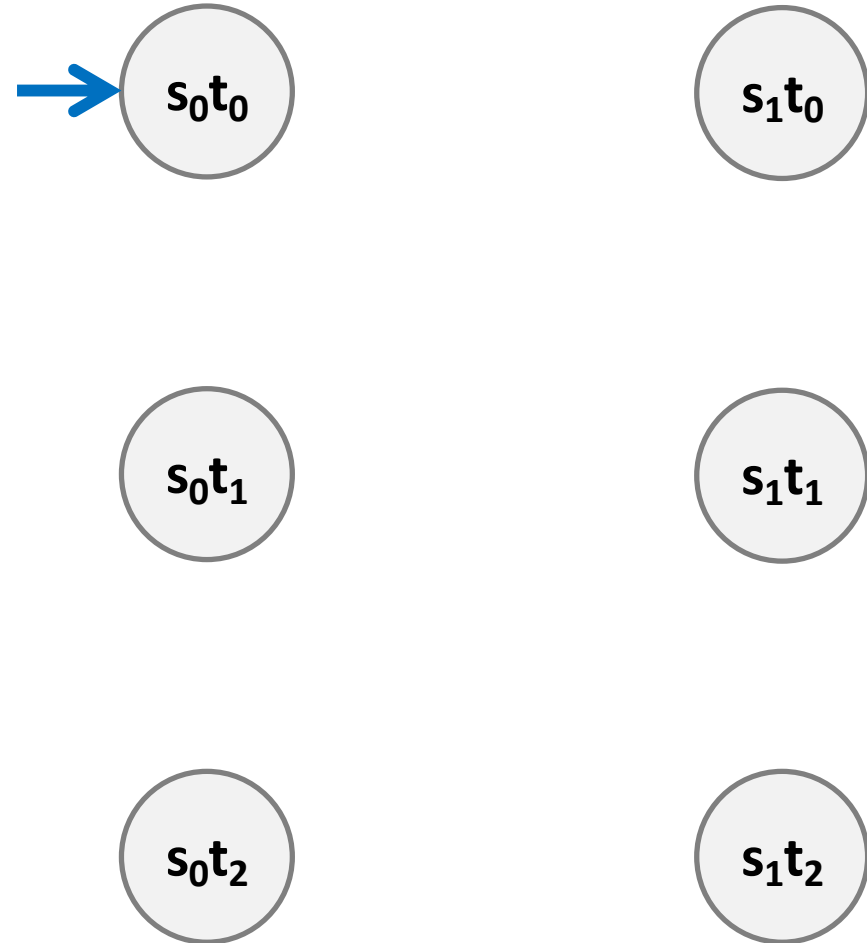
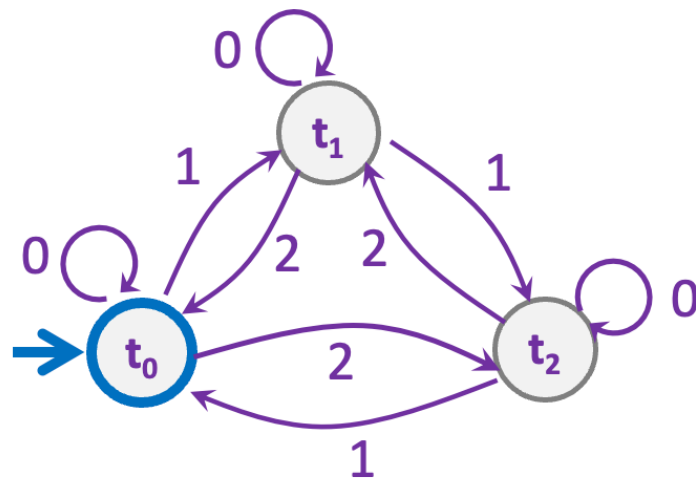
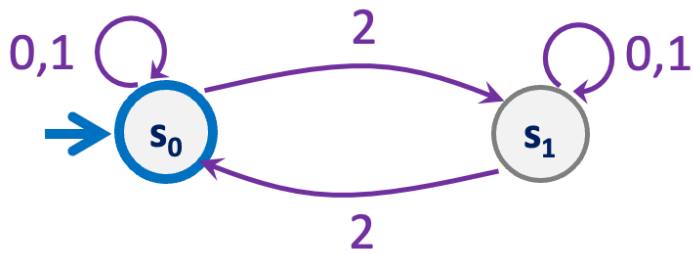


$M_2$ : Strings where the sum of digits mod 3 is 0



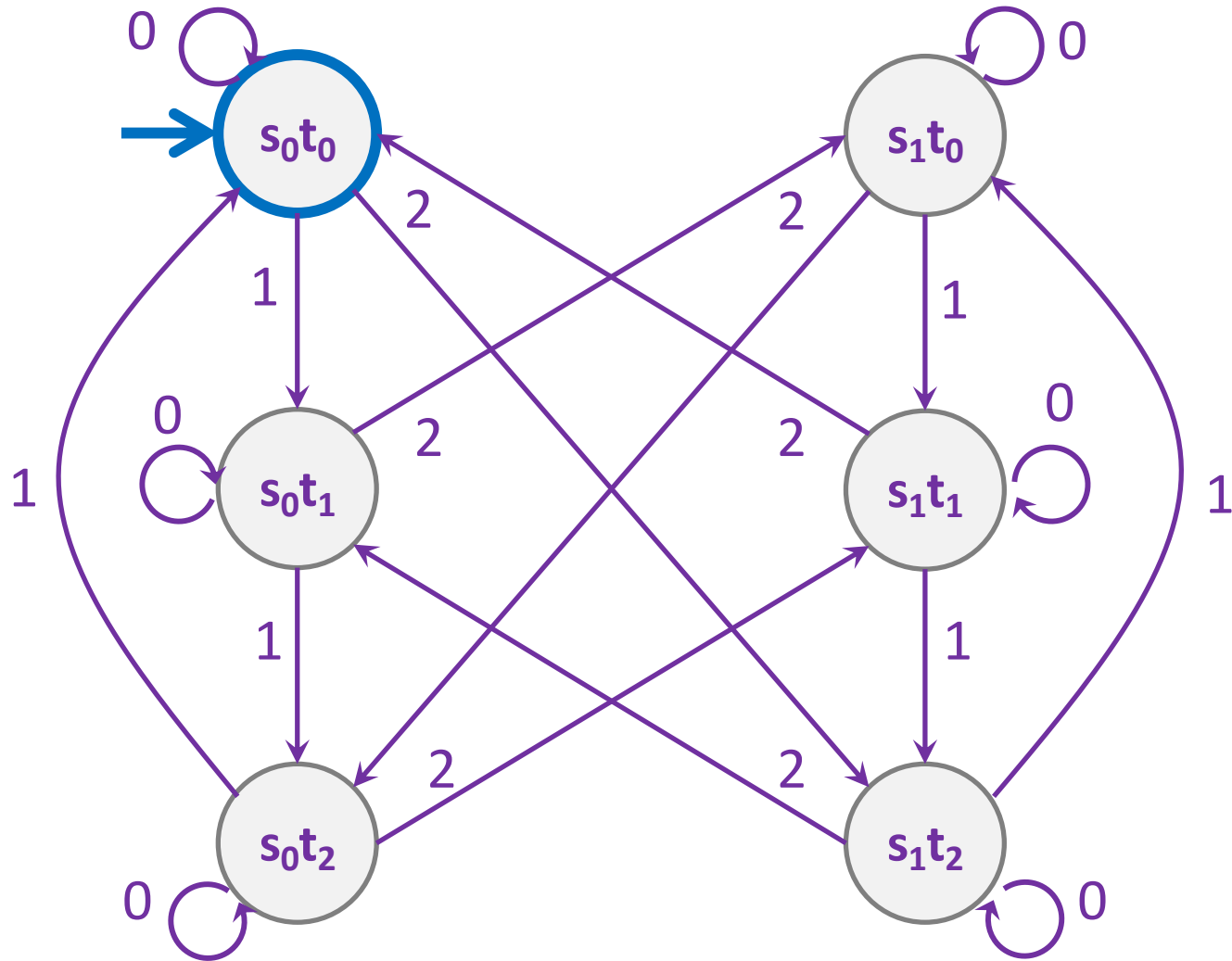
# Strings over $\{0,1,2\}$ w/ even number of 2's AND mod 3 sum 0

---



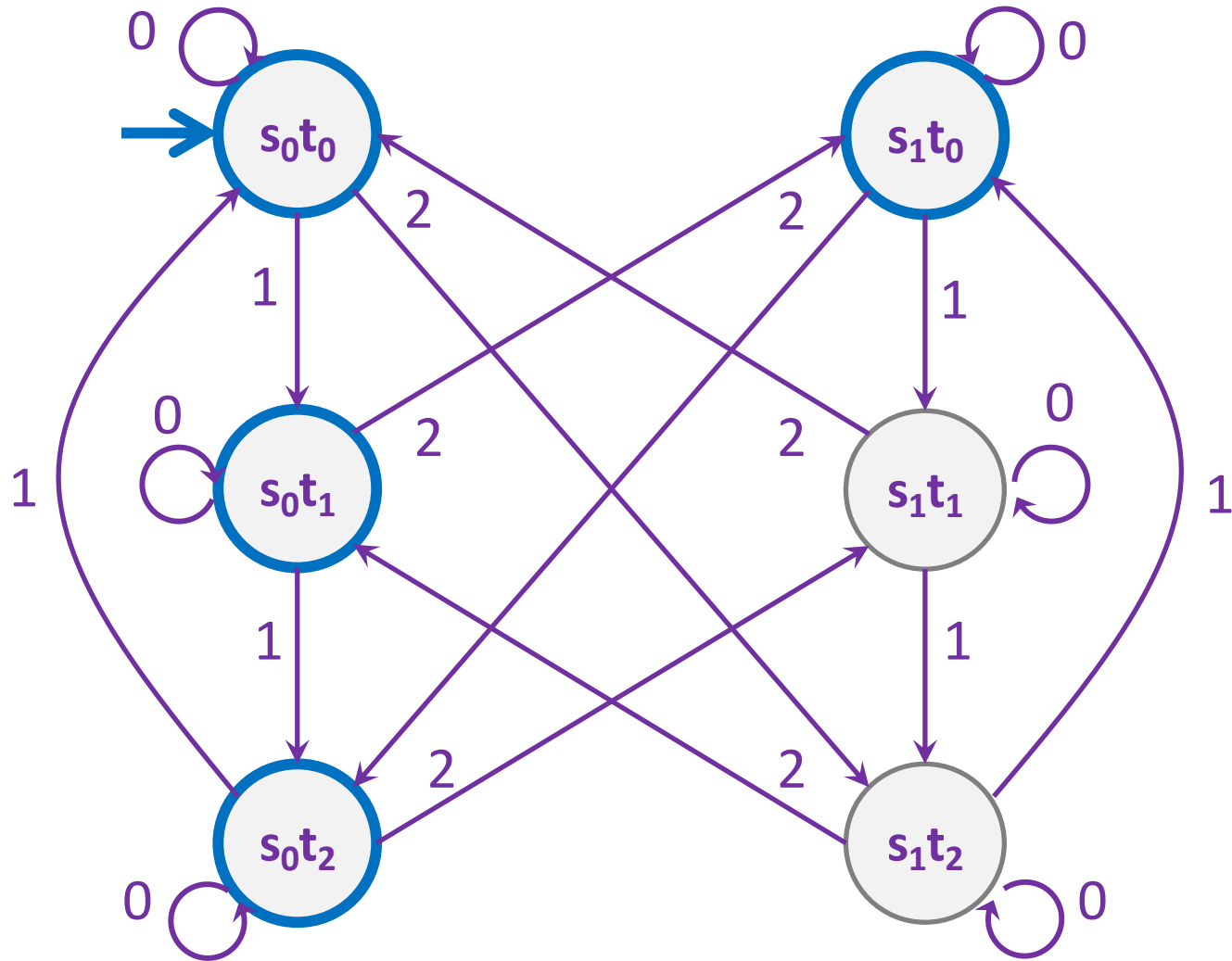
# Strings over $\{0,1,2\}$ w/ even number of 2's AND mod 3 sum 0

---



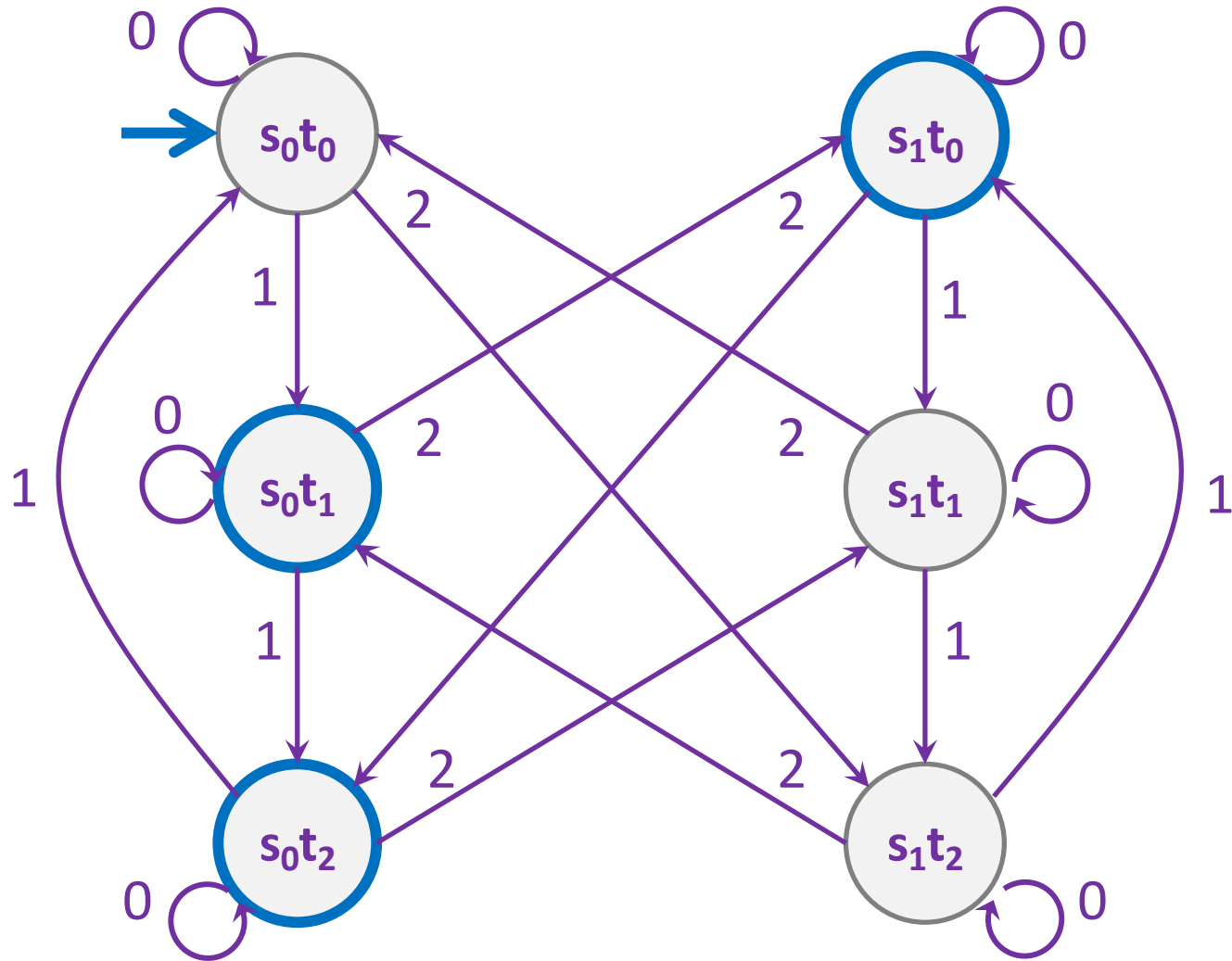
# Strings over $\{0,1,2\}$ w/ even number of 2's OR mod 3 sum 0

---



# Strings over $\{0,1,2\}$ w/ even number of 2's XOR mod 3 sum 0

---

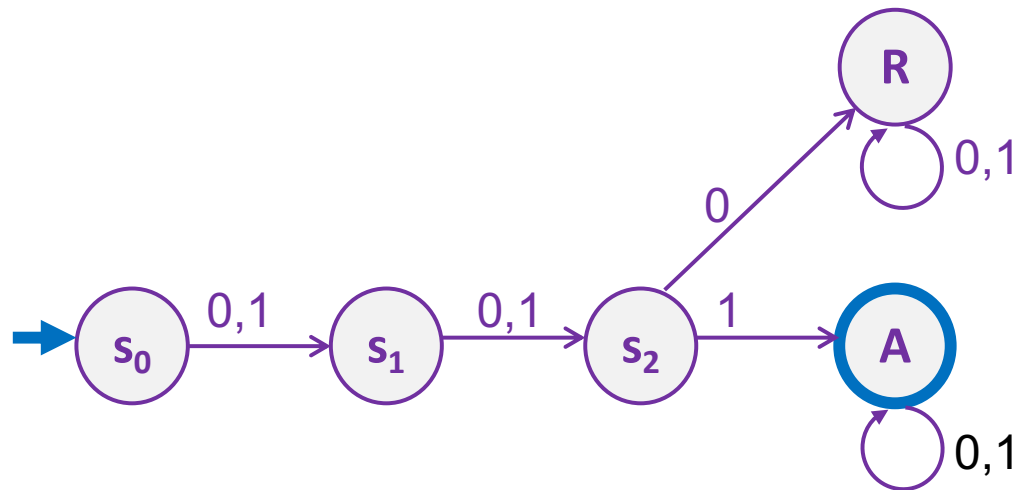


The set of binary strings with a **1** in the 3<sup>rd</sup> position from the start

---

The set of binary strings with a **1** in the 3<sup>rd</sup> position from the start

---

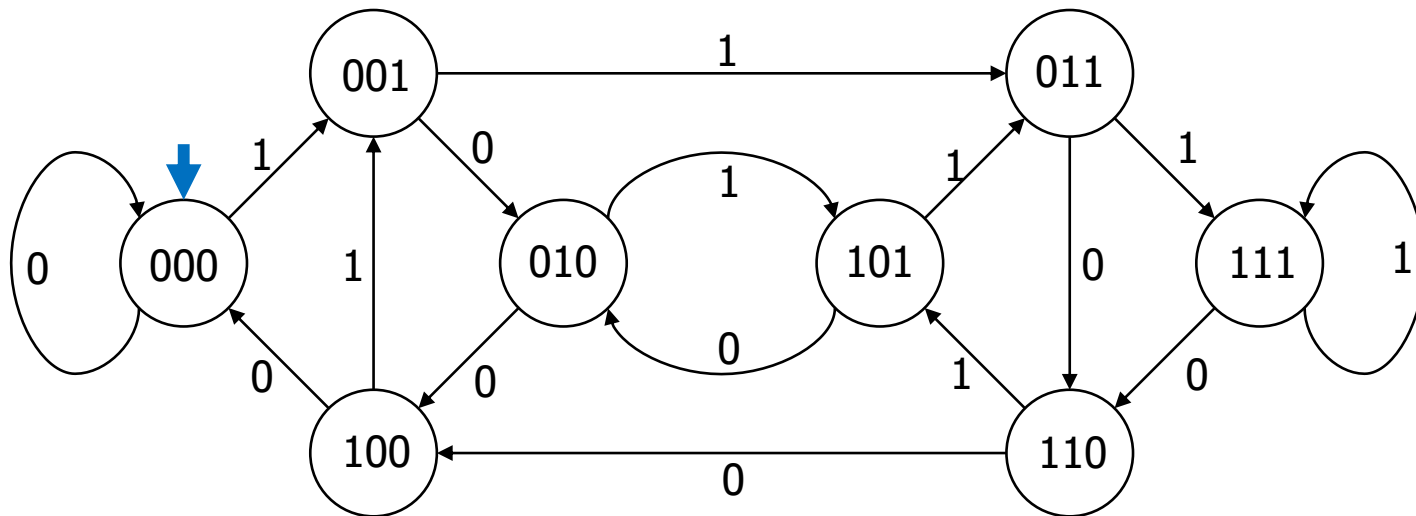


The set of binary strings with a **1** in the 3<sup>rd</sup> position from the end

---

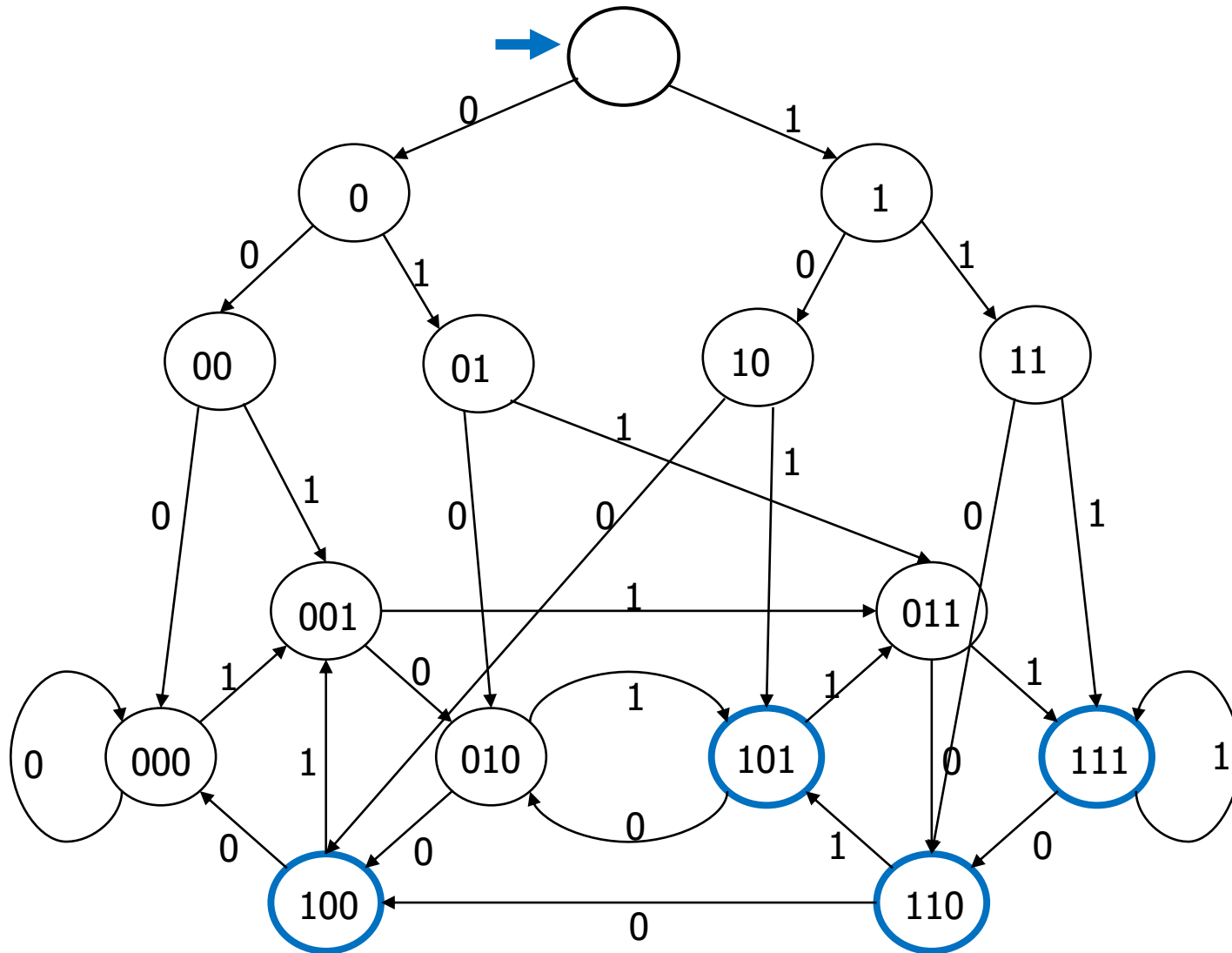
# 3 bit shift register “Remember the last three bits”

---



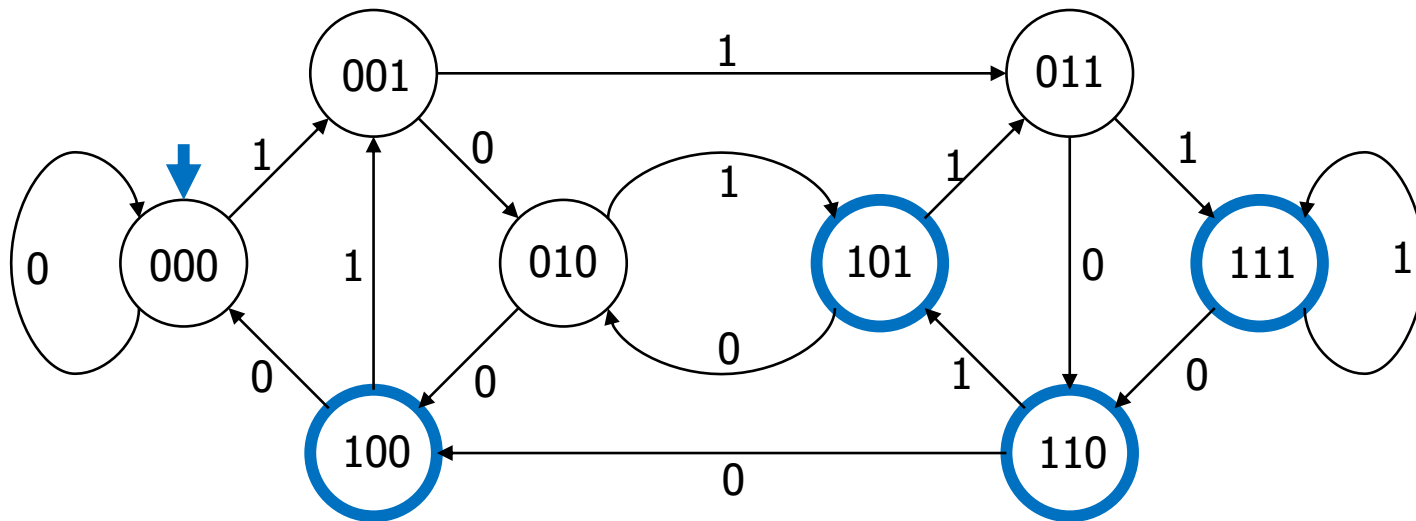
The set of binary strings with a 1 in the 3<sup>rd</sup> position from the end

---



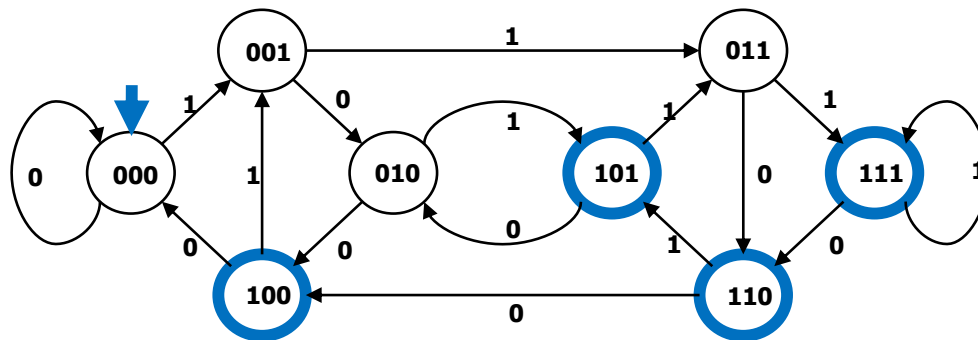
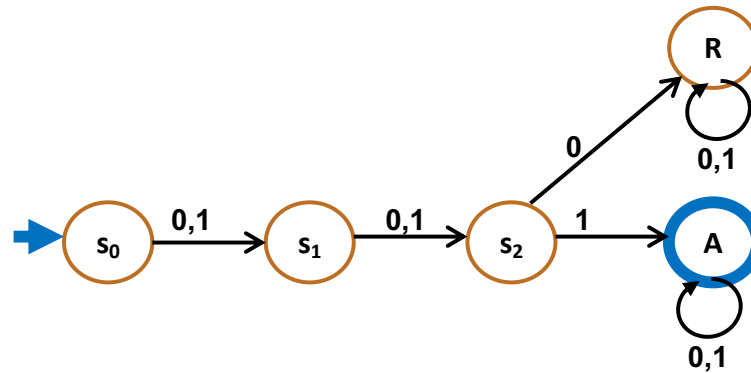
The set of binary strings with a 1 in the 3<sup>rd</sup> position from the end

---



# The beginning versus the end

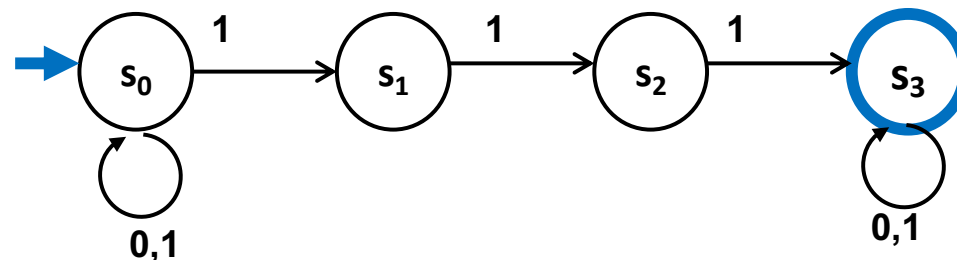
---



# Nondeterministic Finite Automata (NFA)

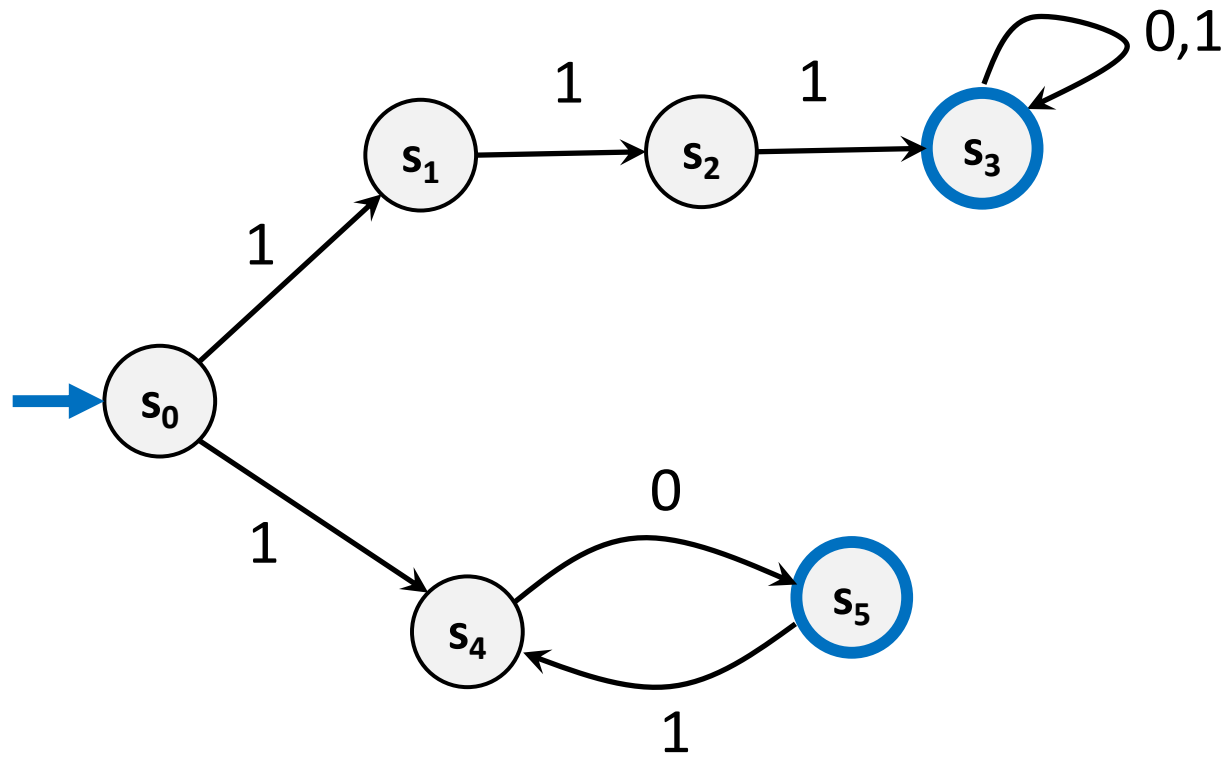
---

- Graph with start state, final states, edges labeled by symbols (like DFA) but
  - Not required to have exactly 1 edge out of each state labeled by each symbol— can have 0 or  $>1$
  - Also can have edges labeled by empty string  $\epsilon$
- **Definition:**  $x$  is in the language recognized by an NFA if and only if some valid execution of the machine gets to an accept state



## Consider This NFA

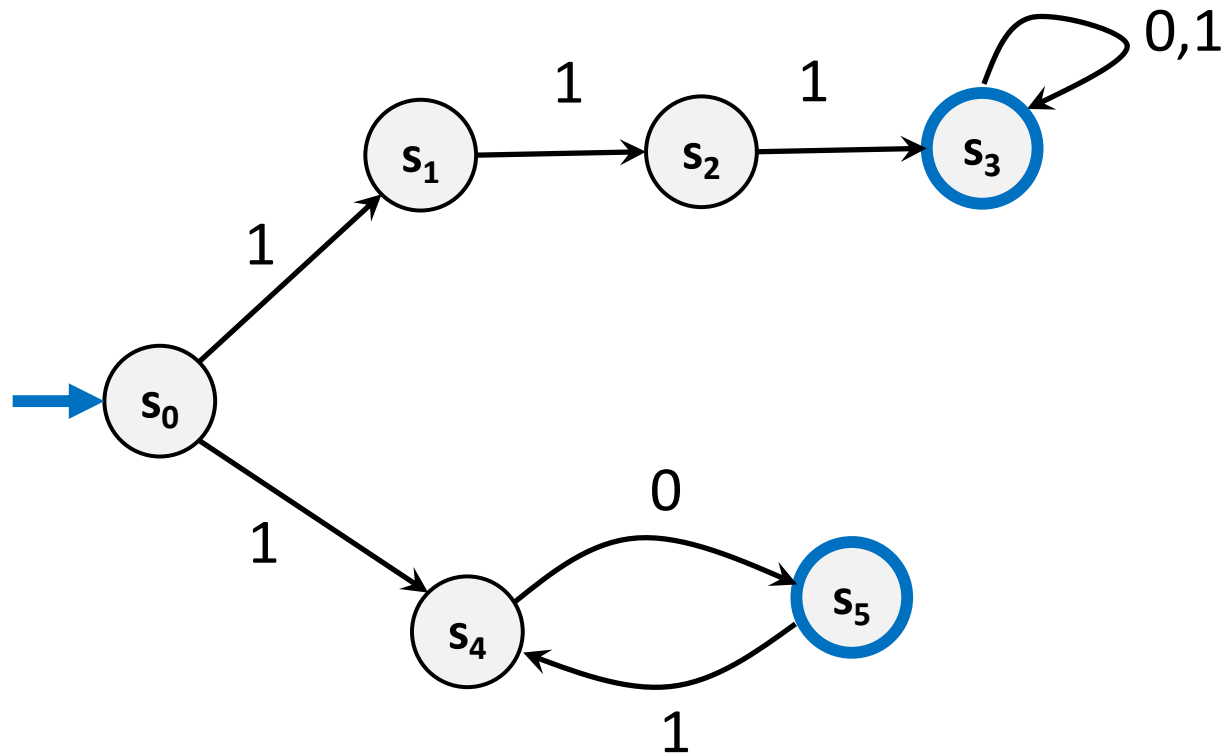
---



What language does this NFA accept?

## Consider This NFA

---

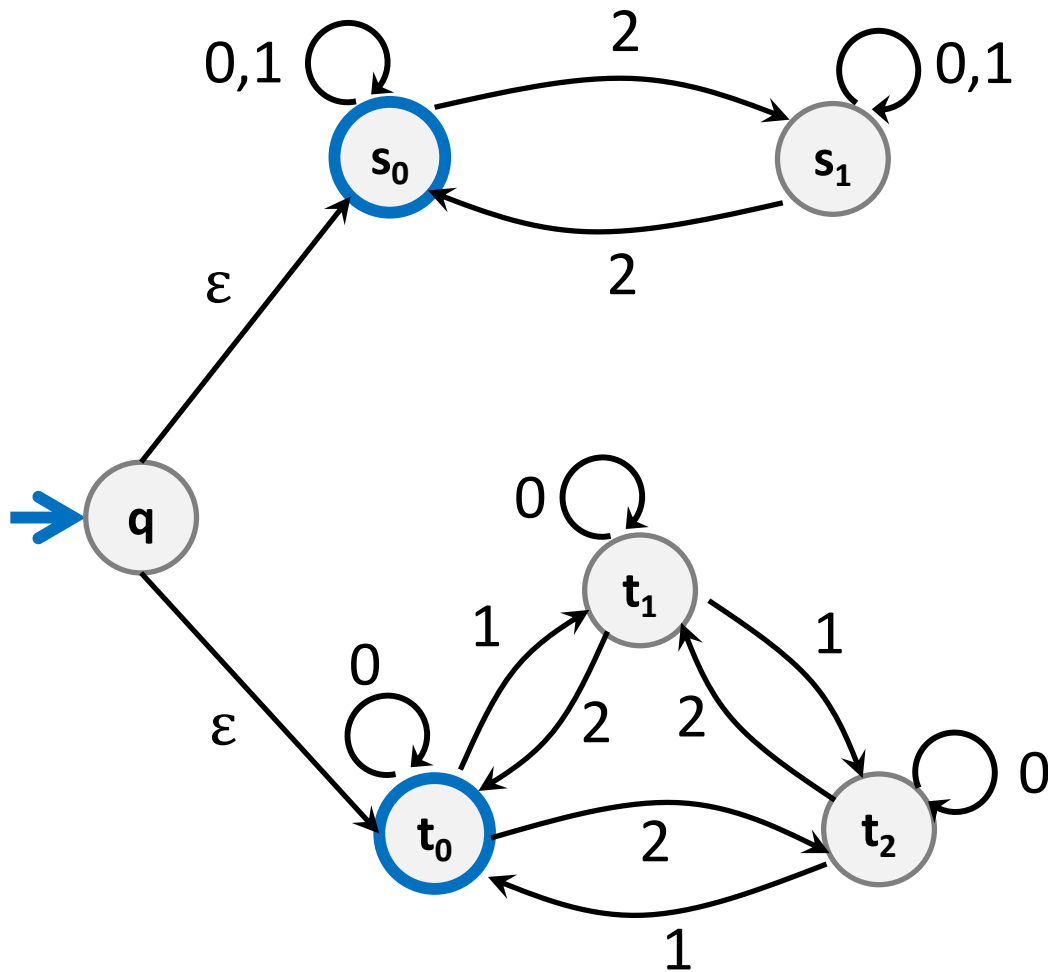


What language does this NFA accept?

$$10(10)^* \cup 111(0 \cup 1)^*$$

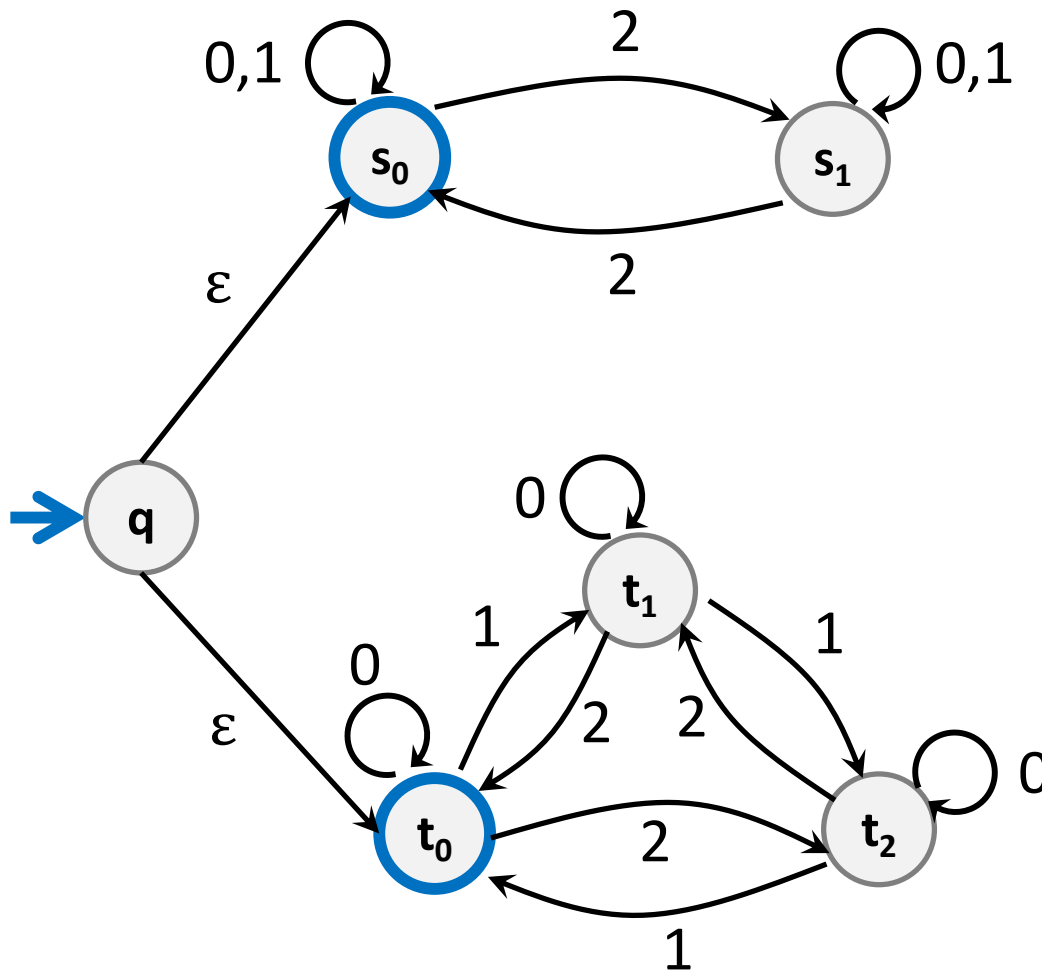
# NFA $\epsilon$ -moves

---



# NFA $\epsilon$ -moves

Strings over  $\{0,1,2\}$  w/even # of 2's OR sum to 0 mod 3

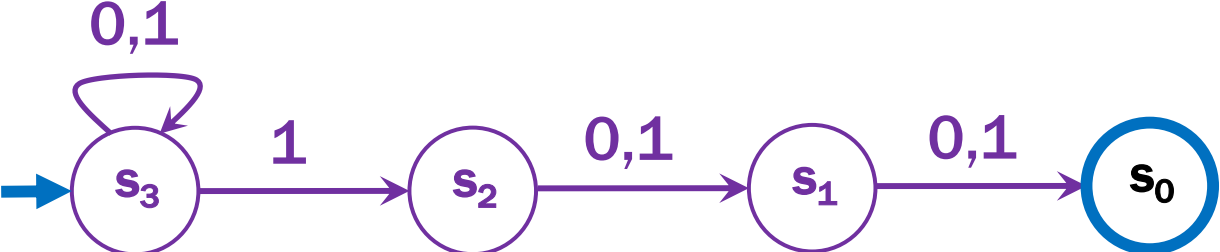


**NFA for set of binary strings with a 1 in the 3<sup>rd</sup> position from the end**

---

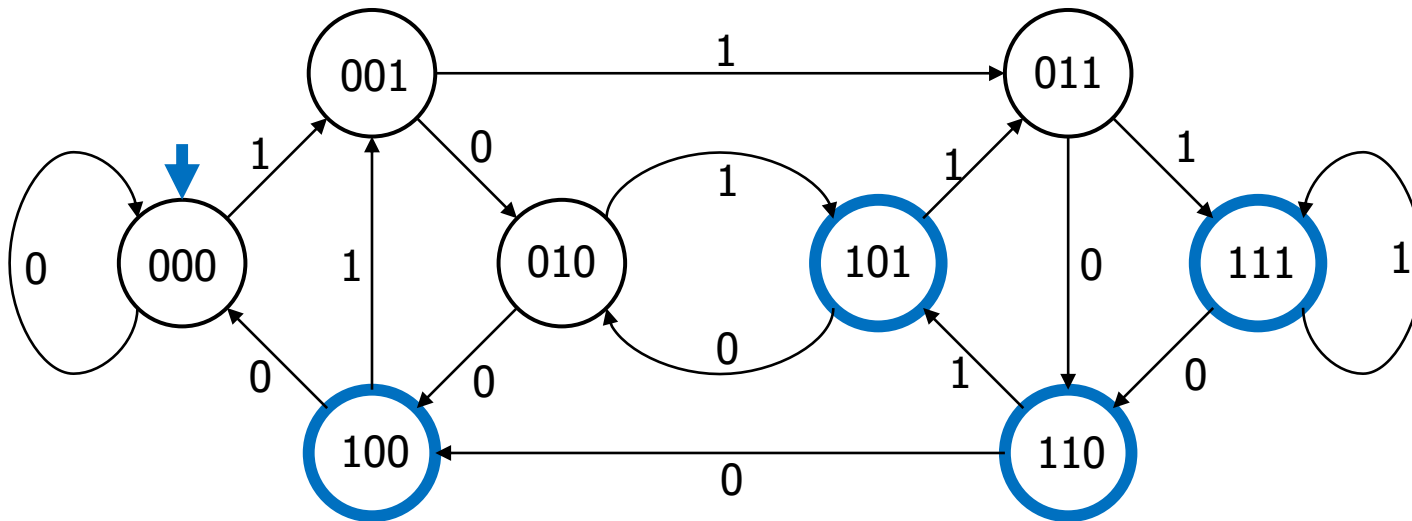
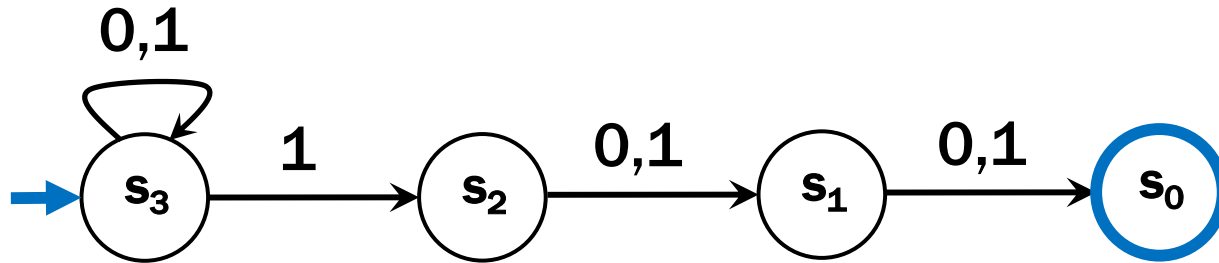
# NFA for set of binary strings with a 1 in the 3<sup>rd</sup> position from the end

---



# Compare with the smallest DFA

---



# Summary of NFAs

---

- **Generalization of DFAs**
  - drop two restrictions of DFAs
  - every DFA is an NFA
- ***Seem* to be more powerful**
  - designing is easier than with DFAs
- ***Seem* related to regular expressions**