CS420

Introduction to the Theory of Computation

Lecture 1: Introduction; finite automata

Tiago Cogumbreiro

About the course

- **Intructor:** Tiago (蒂亚戈) Cogumbreiro
- **Classes:** Tuesday & Thursday
  5:30pm to 6:45pm at W-02-0158, Wheatley
- **Office hours:** Tuesday & Thursday
  3:30pm to 5:00pm at S-3-183, Science Center

Course homepage

cogumbreiro.github.io/teaching/cs420/f19/

(At the bottom right of my homepage.)

- **Forum & announcements:** piazza.com/umb/fall2019/cs420/home
- **Attendance tracking:** www.estalee.com
  Course code: ZS40HJD
- **Homework assignment:** https://tinyurl.com/yy4f9n4d (Blackboard)
- **Syllabus**, **Slides**, **Video recordings**

Course grading

- Course is divided into 3 modules (8 lessons)
- Each module is evaluated with a mini-test (32%)
- Mini-tests evaluate a single module
- Each module has a recap lesson
- Attendance and participation counts (4%)
  Tracking starts Tuesday, Sept 10
- Weekly homeworks
  (ungraded; may be used as extra credit when between grades; see syllabus)

A birdseye view of CS420

What are the limits of programs?

Limits of computation

- Different classes of machines
- The limits of each of these classes
- What properties each class enjoys

Limits of computation

- Different classes of machines
- The limits of each of these classes
- What properties each class enjoys

Classes of machines

Finite Automata	Parse regular expressions
Pushdown Automata	Parse structured data (programs)
Turing Machines	Any program

Some of what we will learn

- Can we write a program that checks if two regex are equivalent?

Some of what we will learn

- Can we write a program that checks if two regex are equivalent?
- Are two grammars equal?

Some of what we will learn

- Can we write a program that checks if two regex are equivalent?
- Are two grammars equal?
- We need to parse some data; do we need a regex or a grammar?

Some of what we will learn

- Can we write a program that checks if two regex are equivalent?
- Are two grammars equal?
- We need to parse some data; do we need a regex or a grammar?
- Can we know if a program terminates without running it?

Some of what we will learn

- Can we write a program that checks if two regex are equivalent?
- Are two grammars equal?
- We need to parse some data; do we need a regex or a grammar?
- Can we know if a program terminates without running it?
- Are two machines/programs equal?

Some of what we will learn

- Can we write a program that checks if two regex are equivalent?
- Are two grammars equal?
- We need to parse some data; do we need a regex or a grammar?
- Can we know if a program terminates without running it?
- Are two machines/programs equal?
- Can a given algorithm give an answer for all inputs?

Techniques

- **State-machines**
  Structure concurrency/parallelism/User Interfaces; UML diagrams

Techniques

- **State-machines**
  Structure concurrency/parallelism/User Interfaces; UML diagrams
- **Regular expressions** (regex)
  String matching rules

Techniques

- **State-machines**
  Structure concurrency/parallelism/User Interfaces; UML diagrams
- **Regular expressions** (regex)
  String matching rules
- **Grammars**
  Data specification; Parsing data

Techniques

- **State-machines**
  Structure concurrency/parallelism/User Interfaces; UML diagrams
- **Regular expressions** (regex)
  String matching rules
- **Grammars**
  Data specification; Parsing data
- **Turing machines**
  Theory of computation

Techniques

- **State-machines**
  Structure concurrency/parallelism/User Interfaces; UML diagrams
- **Regular expressions** (regex)
  String matching rules
- **Grammars**
  Data specification; Parsing data
- **Turing machines**
  Theory of computation
- **Proofs by contradiction**
  Formal proofs

CS420

- Study **algorithms** and **abstractions**
- Theoretical study of the **boundaries of computing**

Finite state automata

Today we will learn...

- Finite automata theory
- State diagram
- Implementation of a finite automaton
- Formal definition of a finite automaton
- Language of a finite automaton

Section 1.1

Decision problem

- We will study **Decision Problems**: yes/no answer
- The set of inputs the problems answers yes are called the **formal language**

Finite Automata

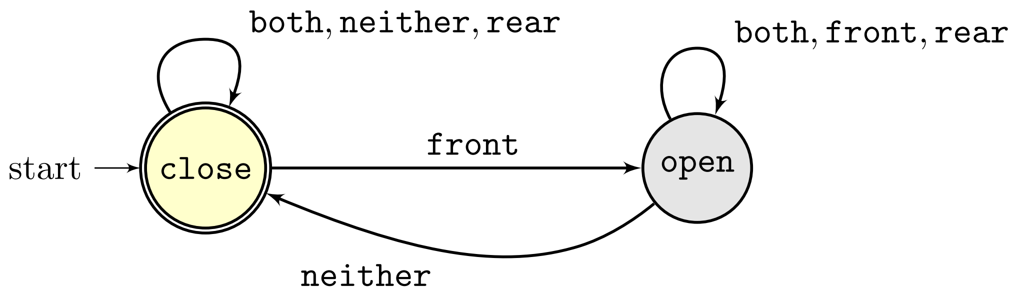
a.k.a. finite state machine

A turnstile controller

> Allows one-directional passage. Opens when the front sensor is triggered. It should remain open while any sensor is triggered, and then close once neither is triggered.

- **States:** open, close
- **Inputs:** front, rear, both, neither

State Diagram



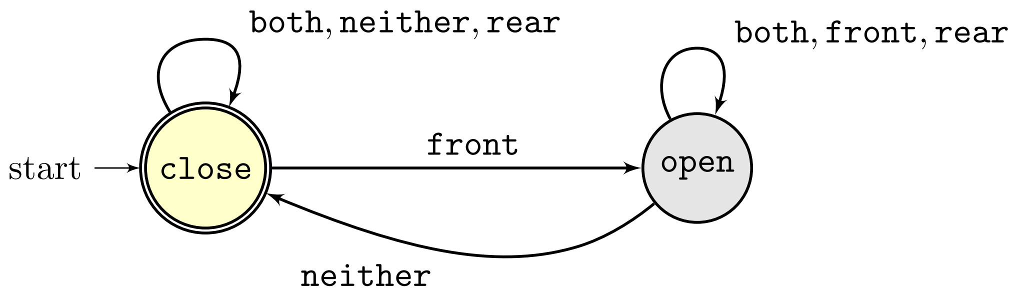
> Each state must have exactly one transition per element of the alphabet (all states must have same transition count)

Definition

- Graph-based diagram
- **Nodes:** called states; annotated with a name
  (Distinct names!)
- **Edges:** called transitions; annotated with inputs
- Initial state has an incoming edge (only one)
- Accepted nodes have a double circle (zero or more)
- Multiple inputs are comma separated

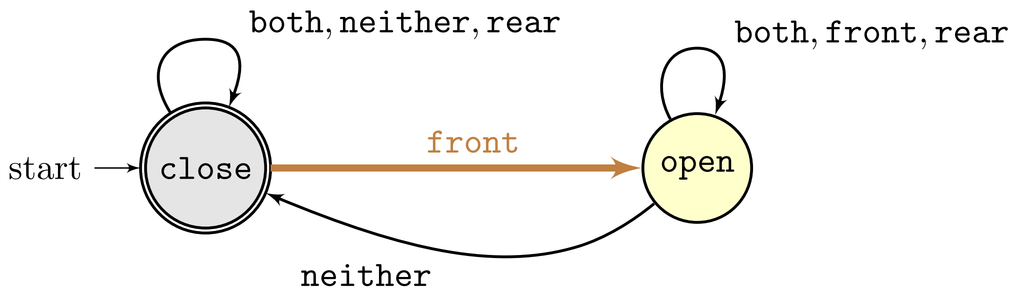
In the example: Two states: `open`, `close`. State `close` is an *accepting* state. State `close` is also the *initial* state

State Diagram: example 1



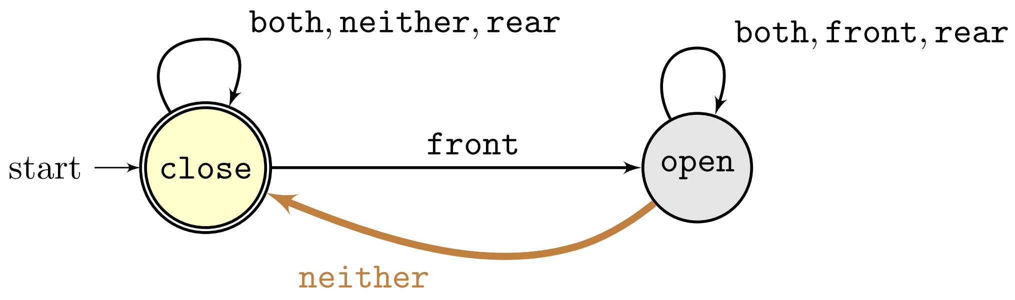
Input: [Front, Neither]

State Diagram: example 1



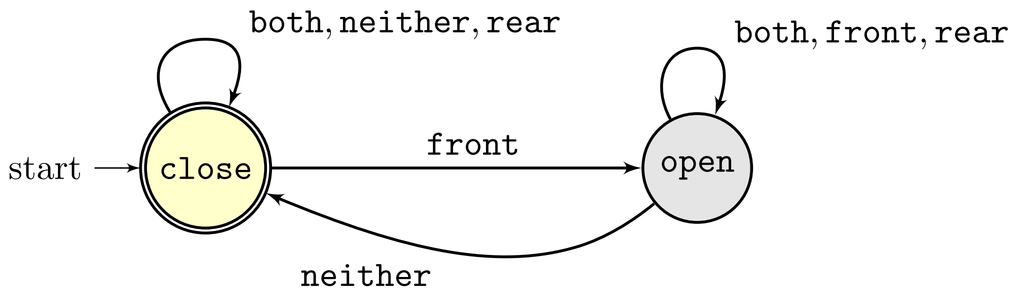
Input: [**Front**, Neither]

State Diagram: example 1



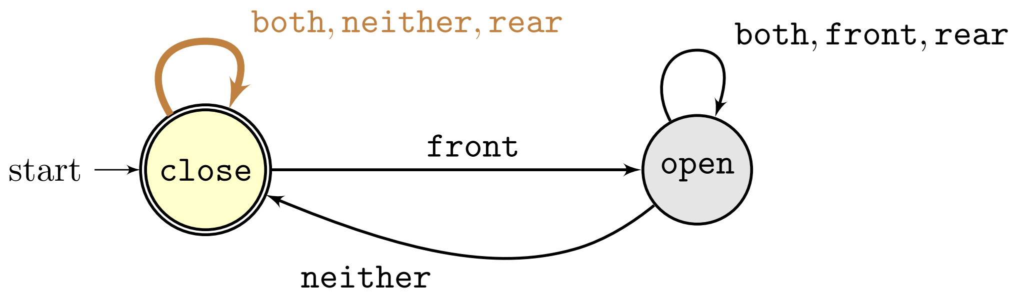
Input: [Front, **Neither**]

State Diagram: example 2



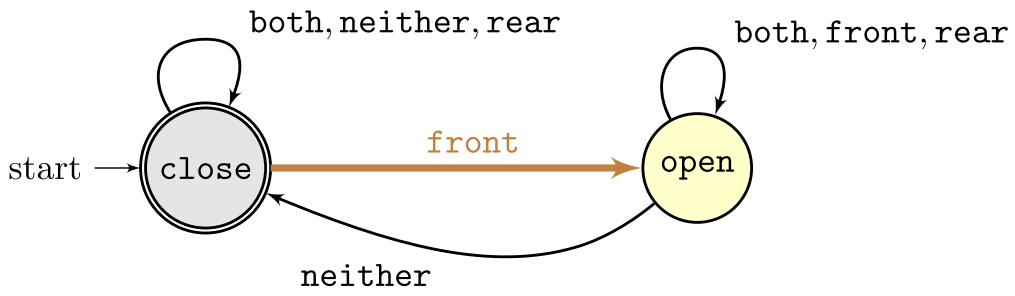
Input: [Rear, Front, Rear, Neither, Rear]

State Diagram: example 2



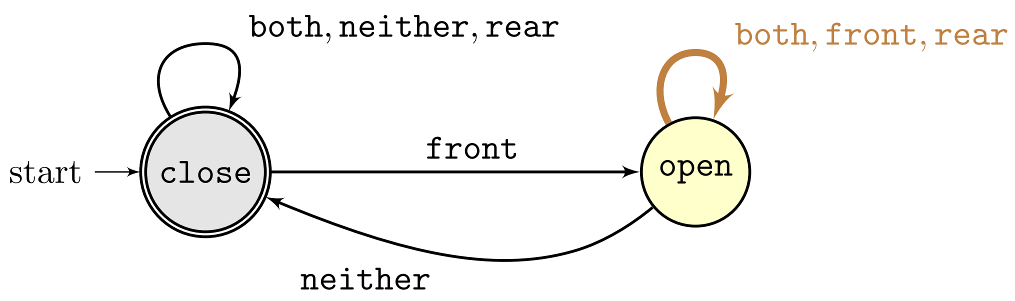
Input: [**Rear**, Front, Rear, Neither, Rear]

State Diagram: example 2



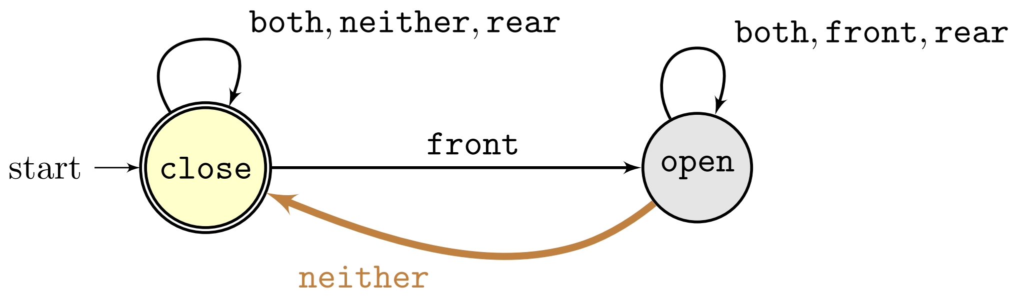
Input: [Rear, **Front**, Rear, Neither, Rear]

State Diagram: example 2



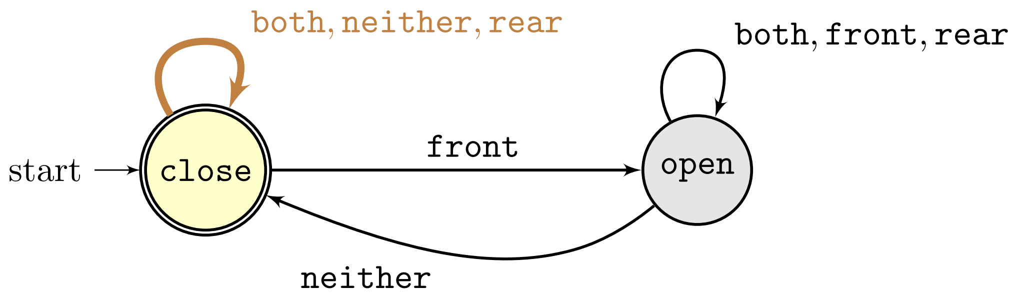
Input: [Rear, Front, **Rear**, Neither, Rear]

State Diagram: example 2



Input: [Rear, Front, Rear, **Neither**, Rear]

State Diagram: example 2



Input: [Rear, Front, Rear, Neither, **Rear**]

The controller of a turnstile

State transition

close	**open**	close	close	close
open	open	open	open	**close**

```python
from enum import *

class State(Enum): Open = 0; Close = 1

class Input(Enum): Neither = 0; Front = 1; Rear = 2; Both = 3

def state_transition(old_st, i):
  if old_st == State.Close and i == Input.Front: return State.Open
  if old_st == State.Open and i == Input.Neither: return State.Close
  return old_st
```

An automaton

An automaton receives a sequence of inputs, processes them, and outputs whether it accepts the sequence.

- *Input:* a string of inputs, and an initial state
- *Output:* `accept` or `reject`

Implementation example

```
def automaton_accepts(inputs):
  st = State.Close
  for i in inputs:
    st = state_transition(st, i)
  return st is State.Close
```

An automaton acceptance examples

```
>>> automaton_accepts([])
True
>>> automaton_accepts([Input.Front, Input.Neither])
True
>>> automaton_accepts([Input.Rear, Input.Front, Input.Front])
False
>>> automaton_accepts([Input.Rear, Input.Front, Input.Rear, Input.Neither, Input.Rear])
True
```

Creating an Automaton library

```
class FiniteAutomaton:
  def __init__(self, states, alphabet, transition_func, start_state, accepted_states):
    assert start_state in states
    assert all(x in states for x in accepted_states)
    self.states = states
    self.alphabet = alphabet
    self.transition_func = transition_func
    self.start_state = start_state
    self.accepted_states = accepted_states

  def accepts(self, inputs):
    st = self.start_state
    for i in inputs:
      assert i in self.alphabet
      st = self.transition_func(st, i)
      assert st in self.states
    return st in self.accepted_states # We accept now multiple states
```

Finite automaton library example

```
>>> a = FiniteAutomaton(State, Input, state_transition, State.Close, [State.Close])
>>> a.accepts([])
True
>>> a.accepts([Input.Front, Input.Neither])
True
>>> a.accepts([Input.Rear, Input.Front, Input.Front])
False
>>> a.accepts([Input.Rear, Input.Front, Input.Rear, Input.Neither, Input.Rear])
True
```

Strings

Alphabet

Let Σ represent a **finite** set of some elements.

Examples

- bits: $\Sigma = \{0, 1\}$
- vowels: $\Sigma = \{\mathtt{a}, \mathtt{e}, \mathtt{i}, \mathtt{o}, \mathtt{u}\}$ or, perhaps $\Sigma = \{\mathtt{a}, \mathtt{e}, \mathtt{i}, \mathtt{o}, \mathtt{u}, \mathtt{y}\}$

String

A string (also known as a word) over an alphabet Σ is a finite and possibly empty sequence of elements of Σ.

Examples

- $[]$, $[0, 0]$, $[0, 1, 0, 0]$ are strings of $\Sigma = \{0, 1\}$
- $[\mathtt{a}, \mathtt{a}, \mathtt{e}]$, $[\mathtt{a}, \mathtt{e}, \mathtt{i}]$, $[\mathtt{u}, \mathtt{a}, \mathtt{i}, \mathtt{e}, \mathtt{e}, \mathtt{e}, \mathtt{e}]$ are all strings of $\Sigma = \{\mathtt{a}, \mathtt{e}, \mathtt{i}, \mathtt{o}, \mathtt{u}\}$

String type

We use $\Sigma^\star$ to denote the type of a string, whose elements are strings over alphabet Σ.

Examples

Let $\Sigma = \{0, 1\}$.

- $[] \in \Sigma^\star$
- $[0, 0] \in \Sigma^\star$
- $[0, 1, 0, 0] \in \Sigma^\star$

Notes

- The string type is a parametric type. The type of strings is parametric on the type of the alphabet, much like a list is parametric on the type of its contents. Unlike programmers, mathematicians favour short notations over more verbose names, so $\Sigma^\star$ is preferred over $\mathtt{String}\langle\Sigma\rangle$.
- In this course we use the word type and set as synonyms.

Formally defining a string

Defining a string

$$w ::= [] \mid c :: w$$

- The empty string $[]$, also represented as ϵ
- Adding one element c to a string w written as $c :: w$

> We will learn that $w ::= [] \mid c :: w$ is known as grammar.

Formally defining a string

We use the following notation to represent a string

$$[c_1, c_2, ..., c_n] \equiv c_1 :: c_2 :: \cdots :: c_n :: []$$

We may also omit the brackets and commas when there is no ambiguity

$$[c_1, c_2, c_3] = c_1 c_2 c_3$$

Operations on strings

Length

$$|\epsilon| = 0$$
$$|c :: w| = 1 + |w|$$

> We are defining the lenght function by branches. Each branch depends on the **pattern** of the argument.

Example

Show that $|[1, 2]| = 2$.

Proof. The proof follows by applying the definition of the length function.
$|1 :: 2 :: []| = 1 + |2 :: []| = 1 + 1 + |[]| = 1 + 1 + 0 = 2$
$\square$

Operations on strings

Concatenation

Attaches two strings together in a new string.

$$\epsilon \cdot w = w$$
$$c_1 :: w_1 \cdot w_2 = c_1 :: (w_1 \cdot w_2)$$

> Formalization of the usual intuition of string concatenation.

Example

$aba \cdot ca = a :: ba \cdot ca = a :: (ba \cdot ca) = a :: b :: (a \cdot ca) = a :: b :: a :: ([] \cdot ca) = abaca$

Exponent

The exponent concatenates n copies of the same string.

$$w^0 = []$$
$$w^{n+1} = w \cdot w^n$$

Example

$ab^3 = ababab$

$ab^1 = ab$

$ab^0 = [] = \epsilon$

Prefix

Defining predicates by cases.

$$\frac{}{\epsilon \text{ prefix } w} \qquad \frac{w_1 \text{ prefix } w_2}{c :: w_1 \text{ prefix } c :: w_2}$$

How do we read this?

The notation $\frac{P}{Q}$ means if P happens, then we can conclude Q.

```python
def prefix(p, w):
  if len(p) == 0: return True  # Rule 1
  if len(p) < len(w): return False
  return p[0] == w[0] and prefix(p[1:], w[1:]) # Rule 2
```

Languages

Language

A language L is a set of strings of type $\Sigma^\star$, formally $L \subseteq \Sigma^\star$.

Examples

- $\{\epsilon\}$ is a language that only contains the empty string
- $\{[c]\}$ is a language that only contains a string with a single character c
- $\{[1, 1, 1]\}$ is a language that only contains string $[1, 1, 1]$
- $\{w \mid w \in \Sigma^\star \wedge \text{ ends with } 1\}$ is a language whose strings' last character is 1
- $\{w \mid w \in \Sigma^\star \wedge |w| \text{ is even}\}$ is a language whose strings' sizes are even numbers

Operations on languages

- Union: $L \cup M = \{w \mid w \in L \vee w \in M\}$
- Intersection: $L \cap M = \{w \mid w \in L \wedge w \in M\}$
- Subtraction: $L - M = \{w \mid w \in L \wedge w \notin M\}$.
  In the book, $L - M = L \setminus M$. Note that $L - M = L \cap \overline{M}$
- Complementation: $\overline{L} = \{w \mid w \notin L\} = \Sigma^\star - L$