CS622
Computing With DFAs
Friday, February 2, 2024
UMass Boston Computer Science
Announcements

• HW 1
  • Due: Wed 2/7 12pm (noon)
A Computation Model is ... (from lecture 1)

• Some **definitions** ...
  
  e.g., A **Natural Number** is either
  - Zero
  - a **Natural Number** + 1

• And **rules** that describe how to **compute** with the **definitions** ...

  To **add** two **Natural Numbers**:
  1. Add the ones place of each num
  2. Carry anything over 10
  3. Repeat for each of remaining digits ...
A Computation Model is ... (from lecture 1)

• Some definitions ...

• And rules that describe how to compute with the definitions ...
A Computation Model is ... (from lecture 1)

• Some definitions ...

\[\text{DEFINITION}\]

A finite automaton is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where

1. \(Q\) is a finite set called the \textit{states},
2. \(\Sigma\) is a finite set called the \textit{alphabet},
3. \(\delta: Q \times \Sigma \to Q\) is the \textit{transition function},
4. \(q_0 \in Q\) is the \textit{start state}, and
5. \(F \subseteq Q\) is the \textit{set of accept states}.

• And rules that describe how to compute with the definitions ...

???
Computation with DFAs (JFLAP demo)

• DFA:

• Input: “1101”

HINT: always work out concrete examples to understand how a machine works
DFA Computation Rules

Informally

Given
• A DFA (~ a “Program”)
• and `Input` = string of chars, e.g. “1101”

To run the automata / “program”:
• `Start` in “start state”

• Repeat:
  • `Read 1 char from input;`
  • `Change state` according to the `transition` table

• Result of computation =
  • `Accept` if last state is `Accept state`
  • `Reject` otherwise
DFA Computation Rules

**Informally**

Given

- A DFA (~ a “Program”)
- and **Input** = string of chars, e.g. “1101”

To run the automata / “program”:

- **Start** in “start state”

**Formally (i.e., mathematically)**

- \[ M = \]
- \[ w = \]

- **Repeat:**
  - Read 1 char from input;
  - Change state according to the **transition** table

- **Result** of computation =
  - Accept if last state is **Accept state**
  - Reject otherwise
DFA Computation Rules

**Informally**

Given
- A DFA (~ a “Program”)
- and Input = string of chars, e.g. “1101”

To run the automata / “program”:
- Start in “start state”

- Repeat:
  - Read 1 char from input;
  - Change state according to the transition table

- Result of computation =
  - Accept if last state is Accept state
  - Reject otherwise

**Formally (i.e., mathematically)**

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1w_2 \cdots w_n$

A run is represented by variables $r_0, \ldots, r_n$, the sequence of states in the computation, where:

- $r_0 = q_0$

- $M$ accepts $w$ if
  sequence of states $r_0, r_1, \ldots, r_n$ in $Q$ exists 
  with $r_n \in F$
DFA Computation Rules

Informally

Given

- A DFA (~ a “Program”)
- and Input = string of chars, e.g. “1101”

To run the automata / “program”:

- Start in “start state”

- Repeat:
  - Read 1 char from input;
  - Change state according to the transition table

Result of computation =

- Accept if last state is Accept state
- Reject otherwise

Formally (i.e., mathematically)

- \( M = (Q, \Sigma, \delta, q_0, F) \)
- \( w = w_1 w_2 \cdots w_n \)
  - A run is represented by variables \( r_0, \ldots, r_n \), the sequence of states in the computation, where:
    - \( r_0 = q_0 \)
    - \( r_i = \)
      - if \( i=1 \), \( r_1 = \delta(r_0, w_1) \)
      - if \( i=2 \), \( r_2 = \delta(r_1, w_2) \)
      - if \( i=3 \), \( r_3 = \delta(r_2, w_3) \)

- \( M \) accepts \( w \) if
  - sequence of states \( r_0, r_1, \ldots, r_n \) in \( Q \) exists . . .
  - with \( r_n \in F \)
DFA Computation Rules

Informally

Given
• A DFA (~ a “Program”)
• and Input = string of chars, e.g. “1101”

To run the automata / “program”:
• Start in “start state”

• Repeat:
  • Read 1 char from input;
  • Change state according to the transition table

• Result of computation =
  • Accept if last state is Accept state
  • Reject otherwise

Formally (i.e., mathematically)

• \( M = (Q, \Sigma, \delta, q_0, F) \)
• \( w = w_1 w_2 \cdots w_n \)
  A run is represented by variables \( r_0, \ldots, r_n \),
  the sequence of states in the computation, where:
  • \( r_0 = q_0 \)
  • \( r_i = \delta(r_{i-1}, w_i) \), for \( i = 1, \ldots, n \)

• \( M \) accepts \( w \) if
  the sequence of states \( r_0, r_1, \ldots, r_n \) in \( Q \) exists …
  with \( r_n \in F \)
DFA Computation Rules

**Informally**

Given
- A DFA (~ a “Program”)
- and **Input** = string of chars, e.g. “1101”

To **run** the automata / “program”:
- **Start** in “start state”

**Repeat:**
- Read 1 char from input;
- **Change state** according to the **transition** table

**Result** of computation =
- **Accept** if last state is **Accept state**
- **Reject** otherwise

---

**Formally (i.e., mathematically)**

\[ M = (Q, \Sigma, \delta, q_0, F) \]

- \( w = w_1 w_2 \cdots w_n \)
- A run is represented by variables \( r_0, \ldots, r_n \), the **sequence of states** in the computation, where:
  - \( r_0 = q_0 \)
  - \( r_i = \delta(r_{i-1}, w_i) \), for \( i = 1, \ldots, n \)

**M accepts** \( w \) if the sequence of states \( r_0, r_1, \ldots, r_n \) in \( Q \) exists . . . with \( r_n \in F \)
An Extended Transition Function

Define **extended transition function:**

- **Domain:**
  - Input state \( q \in Q \) (doesn’t have to be start state)
  - Input string \( w = w_1 w_2 \cdots w_n \) where \( w_i \in \Sigma \)
- **Range:**
  - Output state (doesn’t have to be an accept state)

(Defined recursively)

- **Base case:** ...
Interlude: Recursive Definitions

```javascript
function factorial( n )
{
    if ( n == 0 )
        return 1;
    else
        return n * factorial( n - 1 );
}
```

- Why is this allowed?
  - It's a “feature” (i.e., an axiom!) of the programming language
- Why does this “work”? (Why doesn’t it loop forever?)
  - Because the recursive call always has a “smaller” argument...
  - ... and so eventually reaches the base case and stops
Recursive Definitions

A **Natural Number** is either:
- **Zero**, or
- the **Successor** of a **Natural Number**

**Examples**
- Zero
- **Successor of Zero** ( = “one” )
- **Successor of Successor of Zero** ( = “two” )
- **Successor of Successor of Successor of Zero** ( = “three” ) ...
Recursive Definitions

Recursive definitions have:
- base case and
- recursive case
  (with a “smaller” object)

This is a recursive definition:
* Node is used before it is fully defined (but must be “smaller”)

```
/* Linked list Node*/

class Node {
  int data;
  Node next;
}
```
Strings Are Defined Recursively

A String is either:
- the **empty string** (\(\varepsilon\)), or
- \(xa\) (non-empty string) where
  - \(x\) is a **string**
  - \(a\) is a “char” in \(\Sigma\)

Remember: all strings are formed with “chars” from some **alphabet** set \(\Sigma\)

\[\Sigma^* = \text{set of all possible strings!}\]
Recursive Functions ⇔ Recursive Data

A **Natural Number** is either:
- **Zero**, or
- the **Successor** of a **Natural Number**

```java
function factorial( n )
{
    if ( n == 0 )
        return 1;
    else
        return n * factorial( n - 1 );
}
```

- **Base case**
- **Recursive case**

**Recursive case** must have “smaller” argument

The “**shape**” of recursive function definitions is based on ...
The recursive definition of its input data
An Extended Transition Function

Define **extended transition function**: \( \hat{\delta} : Q \times \Sigma^* \rightarrow Q \)

- **Domain:**
  - Input state \( q \in Q \) (doesn’t have to be start state)
  - Input string \( w = w_1w_2 \cdots w_n \) where \( w_i \in \Sigma \)
- **Range:**
  - Output state (doesn’t have to be an accept state)

(Defined recursively)

- **Base case**
  \[ \hat{\delta}(q, \varepsilon) = \]
An Extended Transition Function

Define **extended transition function**:

- **Domain**:
  - Input state $q \in Q$ (doesn’t have to be start state)
  - Input string $w = w_1w_2 \cdots w_n$ where $w_i \in \Sigma$

- **Range**:
  - Output state (doesn’t have to be an accept state)

(Defined recursively)

- **Base case** $\hat{\delta}(q, \varepsilon) = q$

- **Recursive Case**
  \[
  \hat{\delta}(q, w'w_n) = \delta(\hat{\delta}(q, w'), w_n)
  \]
  where $w' = w_1 \cdots w_{n-1}$

\[\hat{\delta} : Q \times \Sigma^* \rightarrow Q\]
An Extended Transition Function

Define extended transition function:
- **Domain:**
  - Input state \( q \in Q \) (doesn’t have to be start state)
  - Input string \( w = w_1 w_2 \cdots w_n \) where \( w_i \in \Sigma \)
- **Range:**
  - Output state (doesn’t have to be an accept state)

(Defined recursively)

- **Base case**
  \[ \hat{\delta}(q, \varepsilon) = q \]

- **Recursive Case**
  \[ \hat{\delta}(q, w'w_n) = \delta(\hat{\delta}(q, w'), w_n) \]
  where \( w' = w_1 \cdots w_{n-1} \)

\[ \delta : Q \times \Sigma \rightarrow Q \text{ is the transition function} \]

---

A String is either:
- the **empty string** \( \varepsilon \), or
- \( xa \) (non-empty string) where
  - \( x \) is a **string**
  - \( a \) is a “char” in \( \Sigma \)
DFA Computation Rules

Informally

Given

• A DFA (~ a “Program”)
• and Input = string of chars, e.g. “1101”

To run the automata / “program”:

• Start in “start state”

• Repeat:
  • Read 1 char from input;
  • Change state according to the transition table

• Result of computation =
  • Accept if last state is Accept state
  • Reject otherwise

Formally (i.e., mathematically)

\[ M = (Q, \Sigma, \delta, q_0, F) \]
\[ w = w_1 w_2 \cdots w_n \]

A run is represented by variables \( r_0, \ldots, r_n \),
the sequence of states in the computation, where:

• \( r_0 = q_0 \)

• \( r_i = \delta(r_{i-1}, w_i), \text{ for } i = 1, \ldots, n \)

\[ M \text{ accepts } w \text{ if } \text{ sequence of states } r_0, r_1, \ldots, r_n \text{ in } Q \text{ exists } \]
\[ \text{with } r_n \in F \]

This is still a little “informal”
DFA Computation Rules

**Informally**

Given
- A DFA (~ a “Program”)
- and Input = string of chars, e.g. “1101”

To run the automata / “program”:
- Start in “start state”

- Repeat:
  - Read 1 char from input;
  - Change state according to the transition table

- Result of computation =
  - Accept if last state is Accept state
  - Reject otherwise

---

**Formally (i.e., mathematically)**

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1w_2 \cdots w_n$

A run is represented by variables $r_0, \ldots, r_n$, the sequence of states in the computation, where:

- $r_0 = q_0$

- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$

- $M$ accepts $w$ if $\hat{\delta}(q_0, w) \in F$

  sequence of states $r_0, r_1, \ldots, r_n$ in $Q$ exists … with $r_n \in F$
Definition of Accepting Computations

An **accepting computation**, for DFA $M = (Q, \Sigma, \delta, q_0, F)$ and string $w$:

1. *starts* in the start state $q_0$

2. *goes through* a valid sequence of states according to $\delta$

3. *ends* in an accept state

All 3 must be true for a computation to be an **accepting computation**!

$M$ accepts $w$ if $\delta(q_0, w) \in F$
Accepting Computation or Not?

DFA:

- \( \hat{\delta}(q1, 1101) \)
  - Yes
- \( \hat{\delta}(q1, 110) \)
  - No (doesn’t end in accept state)
- \( \hat{\delta}(q2, 101) \)
  - No (doesn’t start in start state)
Alphabets, Strings, Languages

• An **alphabet** is a **non-empty finite set** of symbols

  \[ \Sigma_1 = \{0,1\} \]

  \[ \Sigma_2 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\} \]

• A **string** is a **finite sequence** of symbols from an alphabet

  01001    abracadabra    \(\varepsilon\)  

  *Empty string (length 0)*

• A **language** is a **set** of strings

  \[ A = \{\text{good, bad}\} \]

  \[ \emptyset \]

  *Empty set is a language*

  \[ A = \{w \mid w \text{ contains at least one } 1 \text{ and an even number of } 0s \text{ follow the last } 1\} \]

  *Languages can be infinite*

  *“the set of all ...”*

  *“such that ...”*
Computation and Languages

• The **language** of a machine is the **set** of all strings that it **accepts**

• E.g., A DFA $M$ **accepts** $w$ if $\hat{\delta}(q_0, w) \in F$

• Language of $M = L(M) = \{w \mid M$ accepts $w\}$

"the set of all ..."  "such that ..."
Machine and Language Terminology

DFA $M$ accepts $w$<sup>string</sup>

$M$ recognizes language $A$<sup>Set of strings</sup>

if $A = \{ w \mid M$ accepts $w \}$
Computation and Classes of Languages

• The **language** of a machine = **set of all strings** that it accepts
  
  • E.g., every DFA is associated with a language

• A **computation model** = **set of machines** it defines
  
  • E.g., all possible DFAs are a computation model

• Thus: a **computation model** = **set of languages**
Regular Languages: Definition

If a **deterministic finite automata (DFA)** recognizes a language, then that language is called a **regular language**.

* A **language** is a set of strings.

\[
M \text{ recognizes language } A \text{ if } A = \{w | M \text{ accepts } w\}
\]
A Language, Regular or Not?

- If given: a DFA $M$
  - We know: $L(M)$, the language recognized by $M$, is a **regular language**

  If a DFA recognizes a language, then that language is called a **regular language**.

- If given: a Language $A$
  - Is $A$ a regular language?
    - Not necessarily!
  - How do we determine, i.e., prove, that $A$ is a regular language?
An Inference Rule: Modus Ponens

Premises
- If $P$ then $Q$
- $P$ is true

Conclusion
- $Q$ must also be true

Example Premises
- If there is an DFA recognizing language $A$, then $A$ is a regular language
- There is an DFA $M$ where $L(M) = A$

Conclusion
- $A$ is a regular language!
A Language, Regular or Not?

• If given: a DFA $M$
  • We know: $L(M)$, the language recognized by $M$, is a regular language

If a DFA recognizes a language, then that language is called a regular language.

• If given: a Language $A$
  • Is $A$ a regular language?
    • Not necessarily!
  • How do we determine, i.e., prove, that $A$ is a regular language?

Create an DFA recognizing $A$!