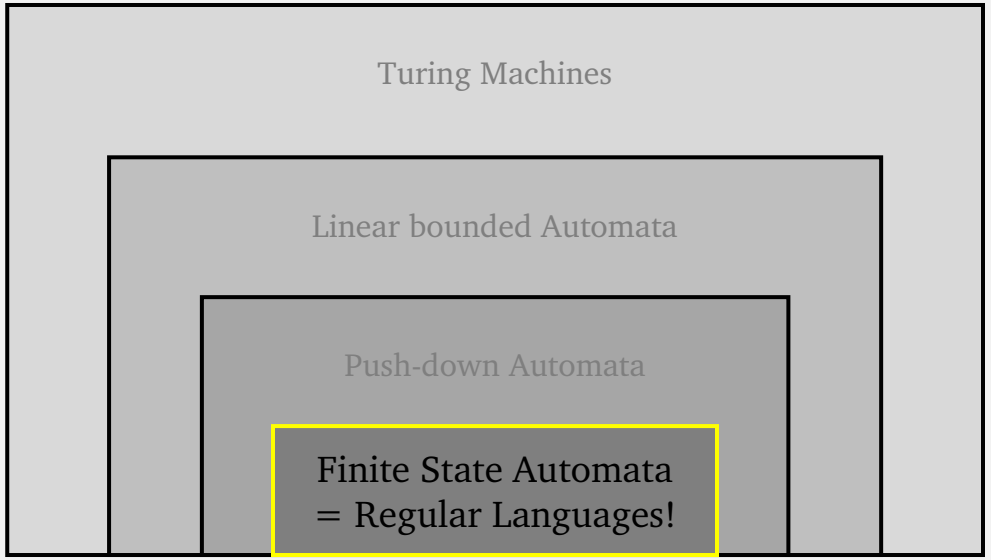
CS 420

Regular Languages

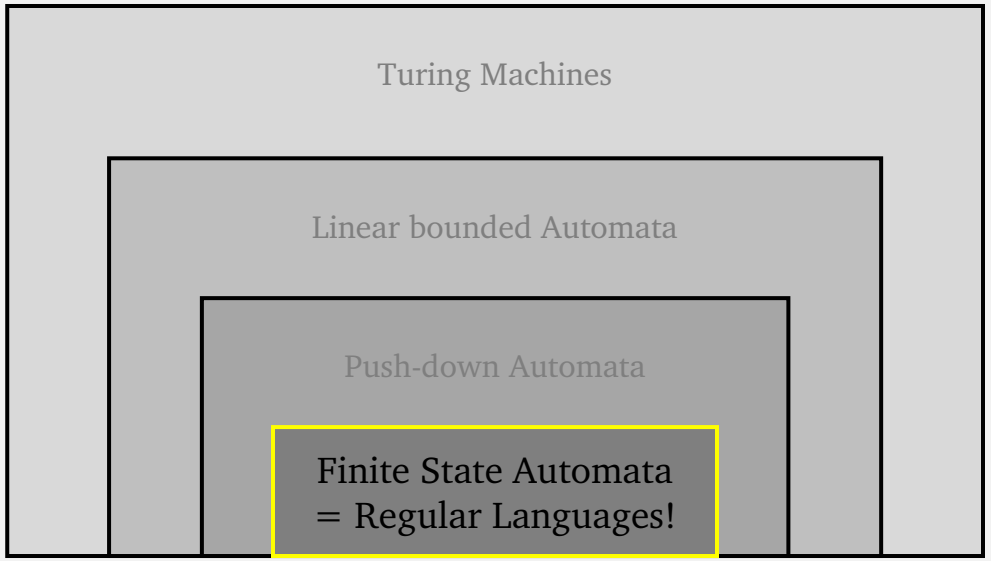
Tuesday, September 22, 2026

UMass Boston Computer Science



Announcements

- HW 1 in
  - ~~Due: Tue 9/22~~
- HW 2 out
  - Due: Tue 9/29 12:30pm



Last Time

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

DFA Computation Rules

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

Given

- A **DFA** (~ a "Program") → • $M = \texttt{mk-DFA}(\ Q,\ \Sigma,\ \delta,\ q_0,\ F\ )$ where …
- and **Input = string of chars,** e.g. "1101" → • $w = w_1 w_2 \cdots w_n$

A **DFA** computation (~ "Program run"):

A DFA **computation** is a sequence of states $r_0, \ldots, r_n \in Q$ where:

- Starts in ***start state*** → • $r_0 = q_0$

- Repeats: → • $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$
  - Read **1 char** from **Input,** and
  - Change state according to ***transition rules***

Can we express this even more concisely …

Result of **computation:**

- Accept if **last state** is ***Accept state*** → • M **accepts** w if $r_n \in F$
- Reject otherwise → • M **rejects** w if $r_n \notin F$

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

(one-step)

A Multi-Step Transition Function

Define a **multi-step transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

set of pairs

* = “0 or more”

Σ^* = set of all possible strings!

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

(one-step)

A Multi-Step Transition Function

Define a **multi-step transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

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

Recursive Functions operate on Recursive Input Data!

(Defined recursively)

- Base case: …

Strings Defined Recursively

Base case

Recursive case

A **String** is either:

- the **empty string** (ε), or
- xa (non-empty string) where
  - x is a **String** ← "smaller" argument
  - a is a "**char**" in Σ

Remember: **strings** are **relative** to some **alphabet** set Σ

Σ^* = set of all possible strings!

Recursive Data needs Recursive Functions

A **Natural Number** is either:

- **Zero**, or
- the **Successor** of a **Natural Number**

Base case

Recursive case

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

Recursive case must have “smaller” argument

(The structure of) **Recursive functions** … **match** the **recursively defined input data!**

(Most) **Recursive functions** are **recursive** because … its **input data** is **recursively defined!**

A Multi-Step Transition Function

Define a **multi-step transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

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

(Defined recursively)

Recursive Input Data needs Recursive Function

Base case

A **String** is either:

- the **empty string** (ε), or
- $w'w_n$ (non-empty string) where
  - w' is a **String**
  - w_n is a "**char**" in Σ

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

"**starting in** q, and **reading zero chars,** the **machine will wind up in state …**"

A Multi-Step Transition Function

Define a **multi-step transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

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

Recursive Input Data needs Recursive Function

A **String** is either:

- the **empty string** (ε), or
- $w'w_n$ (**non-empty string**) where
  - w' is a **String**
  - w_n is a "**char**" in Σ

(Defined recursively)

- Base case $\hat{\delta}(q, \varepsilon) = q$
- Recursive Case $\hat{\delta}(q, w'w_n) = \delta(\hat{\delta}(q, w'), w_n)$

Recursive case

Recursive call ...

... is on the "smaller" argument

string

char

where $w' = w_1 \cdots w_{n-1}$

How to go from 2nd to last, to last state?

"starting in q, and reading w' chars, the machine ends in state ..." $\hat{\delta}(q, w')$

(also called "extended transition function" --- HMU 2.2.4)

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

A Multi-Step Transition Function

Define a **multi-step transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

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

Recursive Input Data
needs
Recursive Function

A **String** is either:

- the **empty string** (ε), or
- $w'w_n$ (non-empty string) where
  - w' is a **String**
  - w_n is a "**char**" in Σ

(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 $w' = w_1 \cdots w_{n-1}$

Previously

DFA Computation Rules

Informally

Given

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

A **DFA** computation (~ "Program run"):

- Starts in *start state*

- Repeats:
  - Read **1 char** from **Input**, and
  - Change state according to *transition rules*

Result of **computation**:

- Accept if **last state** is *Accept state*
- Reject otherwise

Formally (i.e., mathematically)

- M = **mk-DFA**(Q, Σ, δ, q_0, F) ...
- $w = w_1 w_2 \cdots w_n$

A **DFA computation** is a sequence of states $r_0, \ldots, r_n \in Q$ where:

- $r_0 = q_0$

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

Can we express this even more concisely ...

- M **accepts** w if $r_n \in F$
- M **rejects** w if $r_n \notin F$

DFA Computation Rules

Informally

Given

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

A **DFA** computation (~ "Program run"):

- Starts in *start state*

- Repeats:
  - Read **1 char** from **Input**, and
  - Change state according to *transition rules*

Result of **computation**:

- Accept if **last state** is *Accept state*
- Reject otherwise

Formally (i.e., mathematically)

- M = **mk-DFA**(Q, Σ, δ, q_0, F) ...
- $w = w_1 w_2 \cdots w_n$

A DFA **computation** is a sequence of states $r_0, \ldots, r_n \in Q$ where:

- $r_0 = q_0$

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

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

Machine and Language Terminology

- The **language** of a **machine** = set of strings that it **accepts**

Alphabets, Strings, Languages

An **alphabet** defines **"all possible strings"**

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

(strings with non-alphabet symbols are impossible)

$$\Sigma_1 = \{0,1\}$$

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

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

01001 abracadabra ε ← **Empty string** (length 0)

(ε symbol is not in the alphabet!)

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

$$A = \{\mathtt{good}, \mathtt{bad}\}$$

$$\emptyset \quad \{\ \}$$

The **Empty set** is a **language**

Languages can be **infinite**

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

"the set of all ..." "such that ..."

Machine and Language Terminology

- The **language** of a **machine** = <u>set of strings</u> that it **accepts**

- E.g., A **DFA** M ***accepts*** w ← string

M ***recognizes language*** A ← Set of strings

$$\text{if } A = \{w | \ M \text{ accepts } w\}$$

"the set of all …"

"such that …"

Machine and Language Terminology

- The **language** of a **machine** = set of strings that it **accepts**

- E.g., A **DFA** M ***accepts*** w

M ***recognizes language*** $\textsc{LangOf}(M)$

$$\textsc{LangOf}(M) = \{w | \ M \text{ accepts } w\}$$

Definition:
LangOf is function mapping
Machine → Language

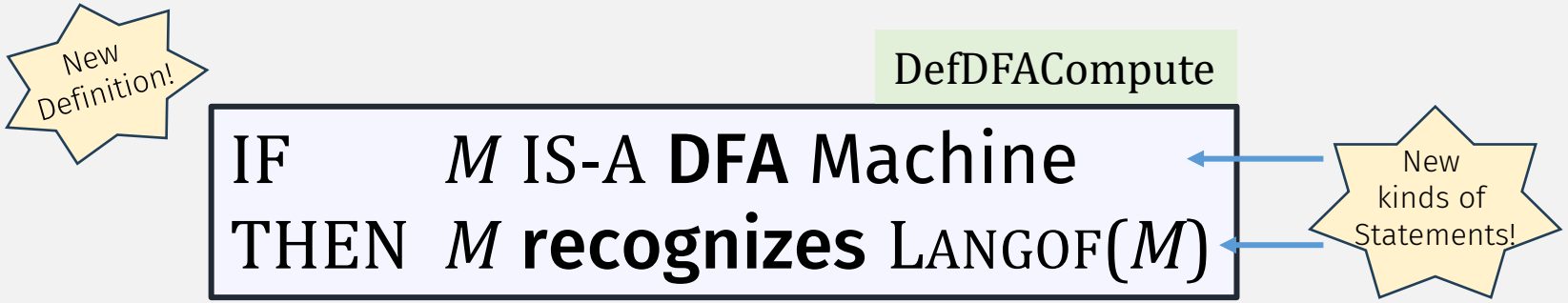
Machine and Language Terminology

- The **language** of a **machine** = set of strings that it **accepts**

- E.g., A **DFA** M ***accepts*** w

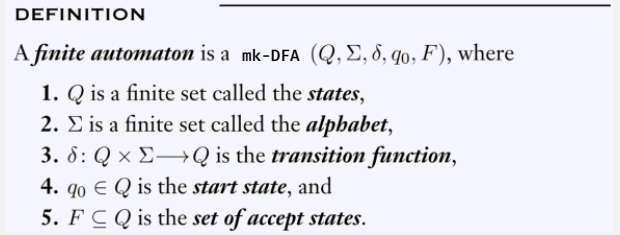
M ***recognizes language*** $\textsc{LangOf}(M)$

$$\textsc{LangOf}(M) = \{w | \ M \text{ accepts } w\}$$



Languages Are Computation Models

- The **language** of a **machine** = set of strings that it **accepts**
  - E.g., a **DFA recognizes** a **language**
- A **computation model** = set of machines it **defines**
  - E.g., **all possible DFAs** are a **computation model**



DEFINITION

A *finite automaton* is a `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the *states*,
2. Σ is a finite set called the *alphabet*,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the *transition function*,
4. $q_0 \in Q$ is the *start state*, and
5. $F \subseteq Q$ is the *set of accept states*.

= set of set of strings

Thus: a **computation model** equivalently = a set of languages

This class is really about studying **sets of languages!**

Regular Languages

- first **set of languages** we will study: **regular languages**

This class is <u>really</u> about studying **sets of languages!**

Regular Languages: Definition

DefRegLang

IF M IS-A **DFA** Machine (some defined machine)

AND

M **recognizes** L (some defined language)

THEN L is IS-A **Regular** Language

A Language, Regular or Not?

Sometimes, a proof doesn't need a proof constructor.

A proof can be inferred from other known proofs / facts

If we know: $proof_M$ = M IS-A **DFA** Machine

Then (we can prove) LANGOF(M) IS-A **Regular** Language

Statements

1. M **recognizes** LANGOF(M)
2. LANGOF(M) IS-A **Regular** Language

Justification

DefDFACompute

1. **APPLY** IF M IS-A **DFA** Machine
   THEN M **recognizes** LANGOF(M)

   TO $proof_M$

DefRegLang

2. **APPLY** IF M IS-A **DFA** Machine AND M **recognizes** L
   THEN L IS IS-A **Regular** Language (some language)

   TO `mk-AND-proof` $proof_M$ $Proof_1$

Here, L = LANGOF(M)

NOTE:
You don't need to write out previously proven statements in every proof.
You can refer to facts in our "library" by name

A Language, Regular or Not?

If we know: $proof_M$ = M IS-A **DFA** Machine

Then (we can prove) LANGOF(M) IS-A **Regular** Language

Statements	**Justification**
1. M **recognizes** LANGOF(M)	1. **APPLY** DefDFACompute **TO** $proof_M$
2. LANGOF(M) IS-A **Regular** Language	2. **APPLY** DefRegLang **TO** **mk-AND-proof** $proof_M$ $Proof_1$

NOTE:
You should write the proof this way

A Language, Regular or Not?

If we know: $proof_M$ = M IS-A **DFA** Machine

Then (we can prove) $\textsc{LangOf}(M)$ IS-A **Regular** Language

If we have: L is some **Language** (i.e., a set of strings)

How can we prove: L IS-A **Regular** Language

Proving That a Language is Regular

Prove: L IS-A **Regular** Language, where L = some **Language** { ... }

Proof: (no constructor)

Statements	**Justifications**
1. M IS-A **DFA** Machine	1.
(TODO 1: design M)	
2. M **recognizes** L	2.
3. L IS-A **Regular** Language	3. APPLY IF M IS-A **DFA** Machine AND M **recognizes** L THEN L is IS-A **Regular** Language
	TO **mk-AND-proof** $Proof_{1?}$ $Proof_{2?}$

DefRegLang

A Language: strings with odd # of **1**s (2 min)

Come up with string examples (in a table), **both**
- **in the language**
- and **not in the language**

- **In-class exercise** (submit to gradescope):

String	In the language?
1	Yes
0	No
01	Yes
11	No
1101	Yes
ε	no

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

IF M IS-A **DFA** Machine AND M **recognizes** L
THEN L is IS-A **Regular** Language

How to prove the language is regular?

Prove there's a DFA recognizing it!

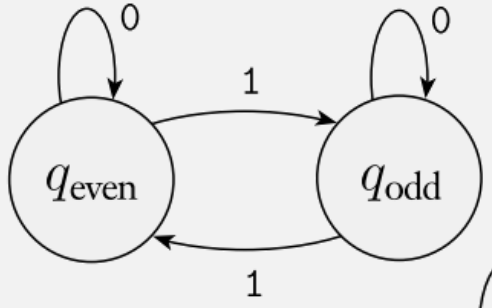
Designing Finite Automata: Tips

- Input is read only once, one char at a time (can't go back!)
- Must decide accept/reject after that
- States = the machine's "memory"!
  - # states must be decided in advance
  - Think about what information must be "remembered".
- Every state/symbol pair must have a defined transition (for DFAs)
- Come up with examples to help you!

Design a DFA: accept strs with odd # **1**s

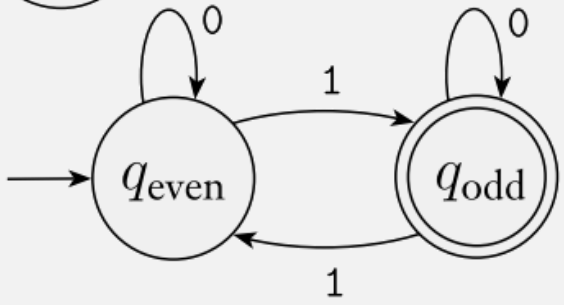
- States:
  - 2 states:
    - seen even 1s so far
    - seen odds 1s so far

- Alphabet: `0` and `1`
- Transitions:



let StateDiag_1 =

- Start / Accept states:



(from now on: **must use** “**let**” or “**where**” to **define variable names**)

Proving That a Language is Regular

Prove: L IS-A **Regular** Language, where L = some **Language** { ... }

Proof:

Statements	**Justifications**
1. M IS-A **DFA** Machine	1.
(TODO 1: design M)	
2. M **recognizes** L	2.
3. L IS-A **Regular** Language	3. APPLY IF M IS-A **DFA** Machine AND M **recognizes** L THEN L is IS-A **Regular** Language

DefRegLang

TO **mk-AND-proof** $Proof_1$ $Proof_2$

Proving That a Language is Regular

Prove: L IS-A **Regular** Language, where L = some **Language** { ... }

DefDFA

DEFINITION

A ***finite automaton*** is a `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

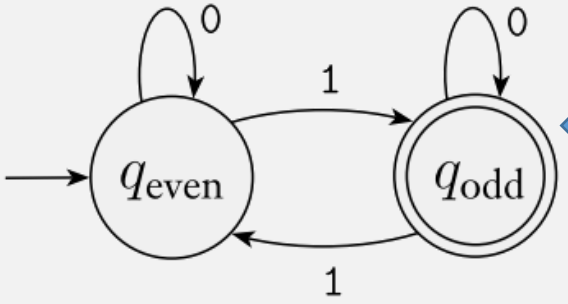
Proof:

	Statements	**Justifications**
☑ 1.	StateDiag$_1$ IS-A **DFA** Machine	1. **BY** ~~DefDFA~~ DefDFAStateDiagram
	(only if problem allows it!)	(only valid if there is a valid state diagram!)
2.	StateDiag$_1$ **recognizes** L	2. (TODO 2: ???)
3.	L IS-A **Regular** Language	3. **APPLY** IF M IS-A **DFA** Machine AND M **recognizes** L THEN L is IS-A **Regular** Language

DefRegLang

TO **`mk-AND-proof`** $Proof_1$ $Proof_2$

“Prove” that DFA recognizes a language



???

Language = { w | w has odd # of **1**s}

String	In the language?
1	Yes
0	No
01	Yes
11	No
1101	Yes
ε	no

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

Q1: How can we **“prove”** that a **computation** does what it’s **“supposed to do”?**

These kinds of proofs (e.g., HMU 2.3.4) are usually **hard,** sometimes impossible!

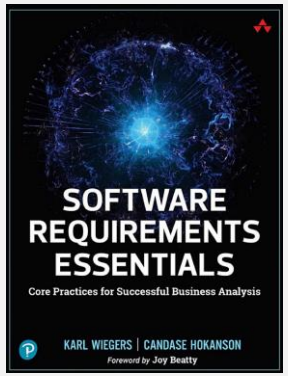
… so we will not do them in this course

But we still need to do something …

Q2: How do **programmers “prove”** that a **computation** does what it’s **“supposed to do”?**

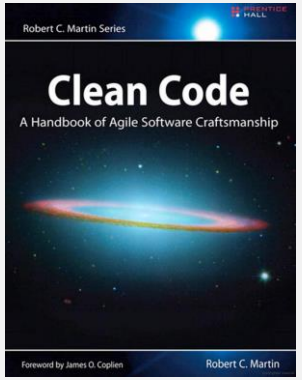
i.e., that the program is **“correct”**?

Interlude: Software Dev 101



Specification (i.e., **requirements**):

- What the **computation**, i.e., **program**, **should do**

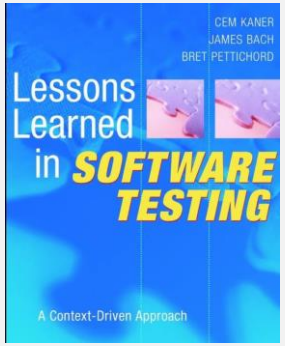


Implementation (i.e., the **code**):

- What the **program does**

Verification (i.e., **testing**):

- Does the **program do what it's supposed to?**
- (i.e., is the **program "correct"**?)



let ExampleTable_1 =

"Prove" that DFA recognizes a language

Verification
(does the **program do what it's supposed to do?**)

These **columns must match** for the **DFA** to **be "correct"!**

Confirm **the DFA:**
- **Accepts strings** in the **language**
- **Rejects strings** <u>not</u> in the **language**

- **In-class exercise** (part 2):

(2 min)

String	In the language?
1	Yes
0	No
01	Yes
11	No
1101	Yes
ε	no

Not a <u>real</u> proof, but …

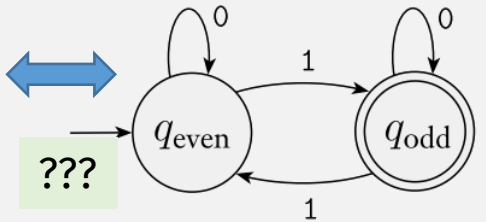
In this class, **a table like this is sufficient to "prove" that a DFA recognizes a language**

Analogous to **what programmers do** (write tests) to **"prove" their computation** (code) **"works"**

Specification
(what the **program should do**)

Implementation
(what the **program does**)

Language = { *w* | *w* has odd # of **1**s}



???

Proving That a Language is Regular

Prove: L IS-A **Regular** Language, where L = some **Language** { ... }

Proof:

	Statements	**Justifications**
1.	StateDiag_1 IS-A **DFA** Machine	1. **BY** DefDFAStateDiagram
2.	StateDiag_1 **recognizes** L	2. (TODO 2: ???)
3.	L IS-A **Regular** Language	3. **APPLY** DefRegLang IF M IS-A **DFA** Machine AND M recognizes L THEN L is IS-A **Regular** Language **TO** **mk-AND-proof** $Proof_1$ $Proof_2$

Proving That a Language is Regular

Prove: L IS-A **Regular** Language, where L = some **Language** { ... }

Proof:

	Statements	**Justifications**
☑ 1.	StateDiag_1 IS-A **DFA** Machine	1. **BY** DefDFAStateDiagram
☑ 2.	StateDiag_1 **recognizes** L	2. **BY** ExamplesTable_1 **COMPARING** StateDiag_1, L
☑ 3.	L IS-A **Regular** Language	3. **APPLY** IF M IS-A **DFA** Machine AND M **recognizes** L THEN L is IS-A **Regular** Language

Not a real proof, but ...

In this class, an "Examples Table" is sufficient to "prove" that a Machine recognizes a language

Remember:
You can refer to facts in our "library" by name

DefRegLang

TO **mk-AND-proof** $Proof_1$ $Proof_2$

Proving That a Language is Regular

Prove: L IS-A **Regular** Language, where L = some **Language** { ... }

Proof:

Statements	**Justifications**
1. StateDiag_1 IS-A **DFA** Machine	1. **BY** DefDFAStateDiagram
2. StateDiag_1 **recognizes** L	2. **BY** ExamplesTable_1 **COMPARING** $\text{StateDiag}_1, L$
3. L IS-A **Regular** Language	3. **APPLY** DefRegLang **TO** **mk-AND-proof** $Proof_1$ $Proof_2$

NOTE:
You should write the proof this way

Another In-class exercise

let $A = \{ w \mid w \text{ has exactly three 1's} \}$

- Prove: A IS-A **Regular** Language

- Where **Σ= {0, 1},**

Remember:
To understand the language, always come up with string examples first (in a table)! Both:
- in the language
- and not in the language

You will need this later in the proof anyways!

DEFINITION

A ***finite automaton*** is a `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

Proving That a Language is Regular

Prove: A IS-A **Regular** Language, where $A = \{ w \mid w \text{ has exactly three 1's} \}$

Proof:

Statements	**Justifications**
1. M IS-A **DFA** Machine	1.
(TODO 1: design M)	
2. M **recognizes** A	2.
3. A IS-A **Regular** Language	3. APPLY IF M IS-A **DFA** Machine AND M **recognizes** L THEN L is IS-A **Regular** Language

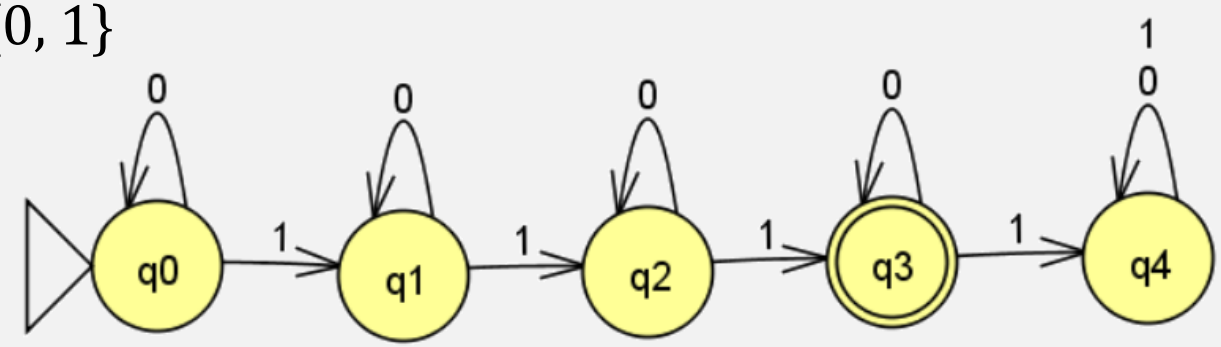
DefRegLang

TO `mk-AND-proof` $Proof_1$ $Proof_2$

In-class exercise Solution

- Design finite automata recognizing:
  - $\{w \mid w$ has exactly three 1's$\}$

- *States*:
  - Need one state to represent how many 1's seen so far
  - Q = {q0, q1, q2, q3, q4+}
- *Alphabet*: $\Sigma = \{0, 1\}$
- *Transitions*:



- *Start state*:
  - q0
- *Accept states*:
  - {q3}

So: a **DFA's computation** recognizes simple string patterns?

Yes!

Have you ever used **a programming language feature** for recognizing simple string patterns?

Regular Expressions! (stay tuned!)