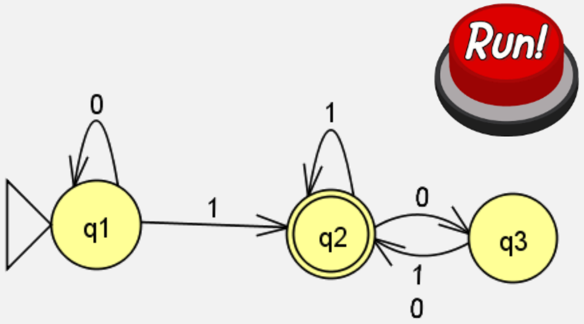
CS 420 / CS 620

Computing With DFAs

Monday, September 15, 2025

UMass Boston Computer Science



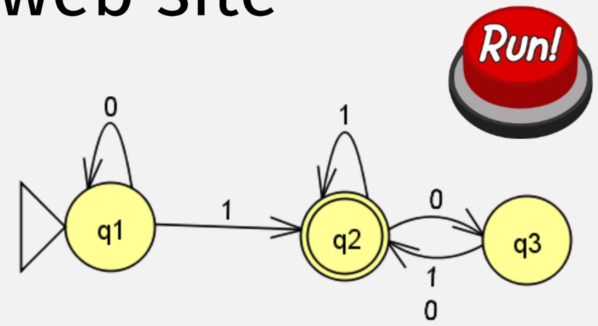
lecture04

Announcements

- HW 1
  - ~~Due: 9/15 12pm (noon) EDT~~

- HW 1
  - Out: 9/15 12pm (noon) EDT
  - Due: 9/22 12pm (noon) EDT

- Check office hour times and locations on course web site



A Computation Model is … (from lecture 1)

- Some **definitions** …

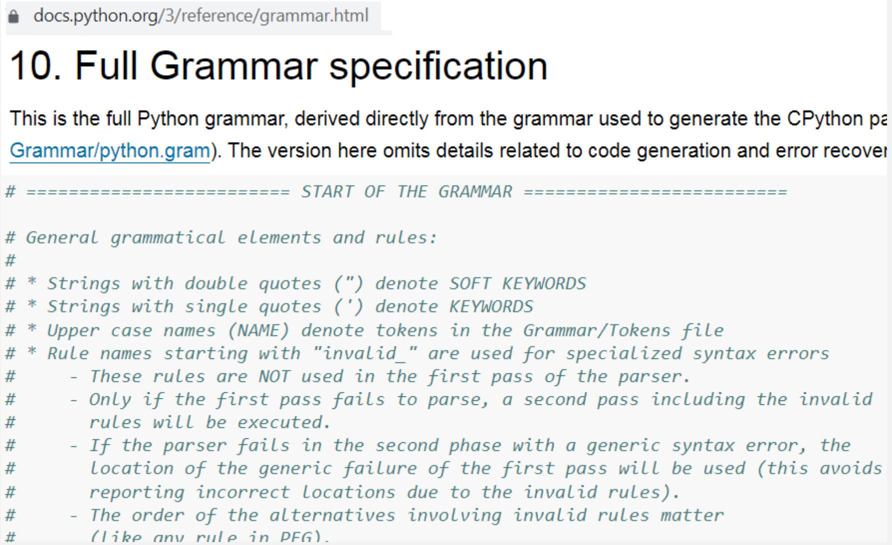
e.g., A **Natural Number** is either
- Zero
- a Natural Number + 1

- 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** …



docs.python.org/3/reference/grammar.html

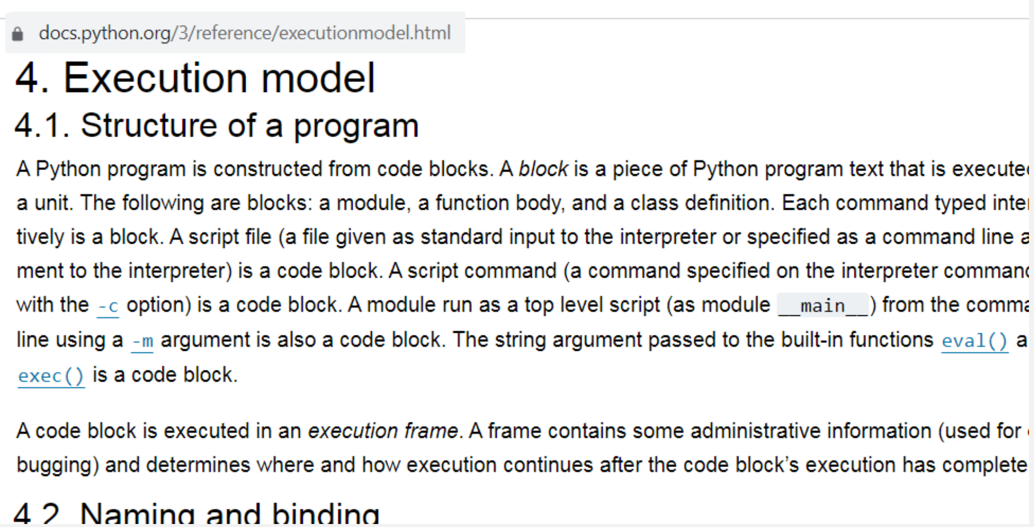
10. Full Grammar specification

This is the full Python grammar, derived directly from the grammar used to generate the CPython pa
Grammar/python.gram). The version here omits details related to code generation and error recover

```
# ======================== START OF THE GRAMMAR =========================

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
#     - These rules are NOT used in the first pass of the parser.
#     - Only if the first pass fails to parse, a second pass including the invalid
#       rules will be executed.
#     - If the parser fails in the second phase with a generic syntax error, the
#       location of the generic failure of the first pass will be used (this avoids
#       reporting incorrect locations due to the invalid rules).
#     - The order of the alternatives involving invalid rules matter
#       (like any rule in PEG).
```

- And **rules** that describe how to **compute** with the **definitions** …



docs.python.org/3/reference/executionmodel.html

4. Execution model

4.1. Structure of a program

A Python program is constructed from code blocks. A *block* is a piece of Python program text that is execute
a unit. The following are blocks: a module, a function body, and a class definition. Each command typed inte
tively is a block. A script file (a file given as standard input to the interpreter or specified as a command line a
ment to the interpreter) is a code block. A script command (a command specified on the interpreter comman
with the `-c` option) is a code block. A module run as a top level script (as module `__main__`) from the comma
line using a `-m` argument is also a code block. The string argument passed to the built-in functions `eval()` a
`exec()` is a code block.

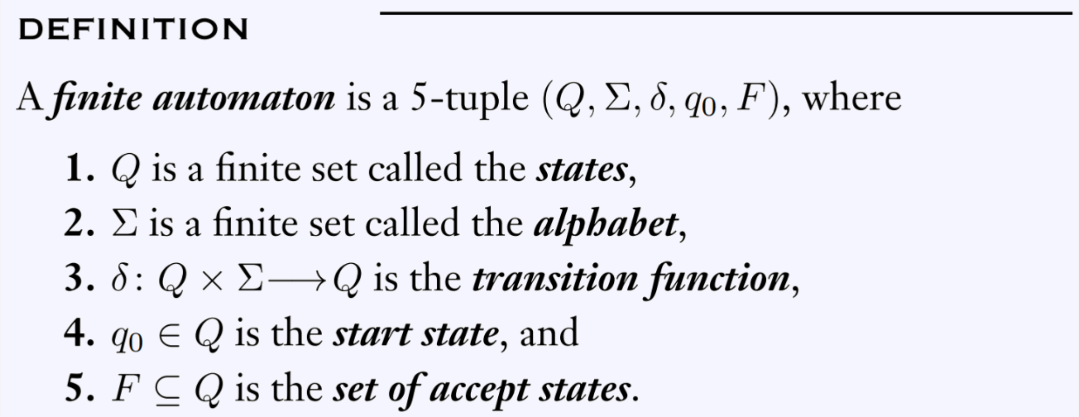
A code block is executed in an *execution frame*. A frame contains some administrative information (used for
bugging) and determines where and how execution continues after the code block's execution has complete

4.2. Naming and binding

Last Time

A Computation Model is … (from lecture 1)

- Some **definitions** …



DEFINITION

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

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

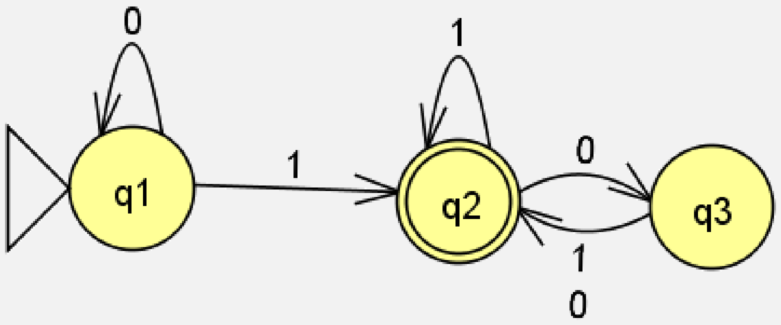
- And **rules** that describe how to **compute** with the **definitions** …

???

Sipser Fig 1.4

Computation with DFAs (JFLAP demo)

- DFA:



- Input: “1101”

HINT: always work out concrete examples to understand how a machine (i.e., “program”) works

DFA Computation Rules

Informally

Given

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

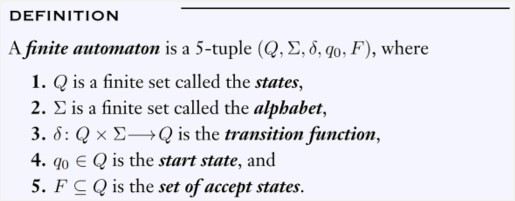
A **DFA** <u>**computation**</u> (~ “Program run”):

- <u>Starts</u> in *start state*

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

<u>**Result**</u> of **computation:**

- <u>Accept</u> if **last state** is *Accept state*
- <u>Reject</u> otherwise



DEFINITION

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

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

DFA Computation Rules

Informally

Given

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

A **DFA** <u>computation</u> (~ "Program run"):

- <u>Starts</u> in *start state*
- <u>Repeats</u>:
  - <u>Read</u> **1 char** from **Input**, and
  - <u>Change state</u> according to *transition rules*

<u>Result</u> of **computation:**

- <u>Accept</u> if **last state** is *Accept state*
- <u>Reject</u> otherwise

Formally (i.e., mathematically)

- $M =$
- $w =$

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)

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

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

- $r_0 = q_0$

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

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)

- $M = (Q, \Sigma, \boxed{\delta}, q_0, F)$
- $w = \boxed{w_1} w_2 \cdots w_n$

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

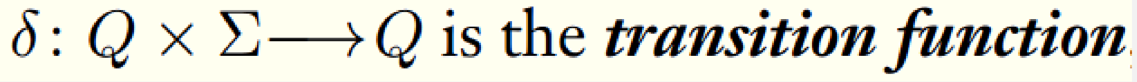
- $\boxed{r_0} = q_0$

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

if i=1, $r_1 = \delta(r_0, w_1)$

if i=2, $r_2 = \delta(r_1, w_2)$

if i=3, $r_3 = \delta(r_2, w_3)$



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

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:**

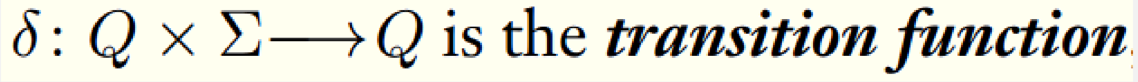
- __Accept__ if **last state** is ***Accept state***
- __Reject__ otherwise

Formally (i.e., mathematically)

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

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 $i = 1, \ldots, n$



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

DFA Computation Rules

Informally

Given

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

A **DFA** <u>computation</u> (~ "Program run"):

- <u>Starts</u> in *start state*

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

<u>Result</u> of **computation:**

- <u>Accept</u> if last state is *Accept state*
- <u>Reject</u> otherwise

Formally (i.e., mathematically)

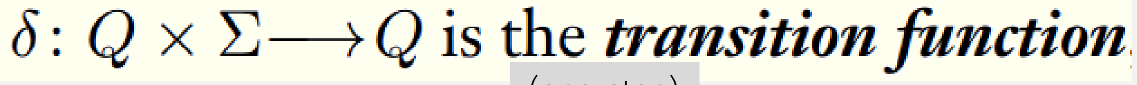
- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

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

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$

This is still pretty verbose …

- M **accepts** w if $r_n \in F$
- 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:** $\hat{\delta} : Q \times \Sigma^* \to Q$

set of pairs

* = “0 or more”

Σ^* = set of all possible strings!

Alphabets, Strings, Languages

An **alphabet** defines **“all possible strings”**

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

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

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

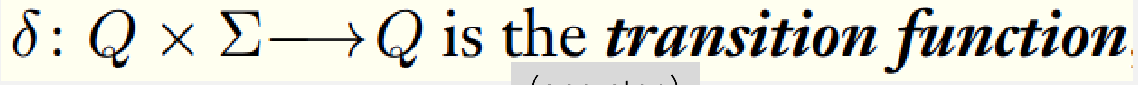
(strings with non-alphabet symbols are impossible)

- 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!)


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

(one-step)

A Multi-Step Transition Function

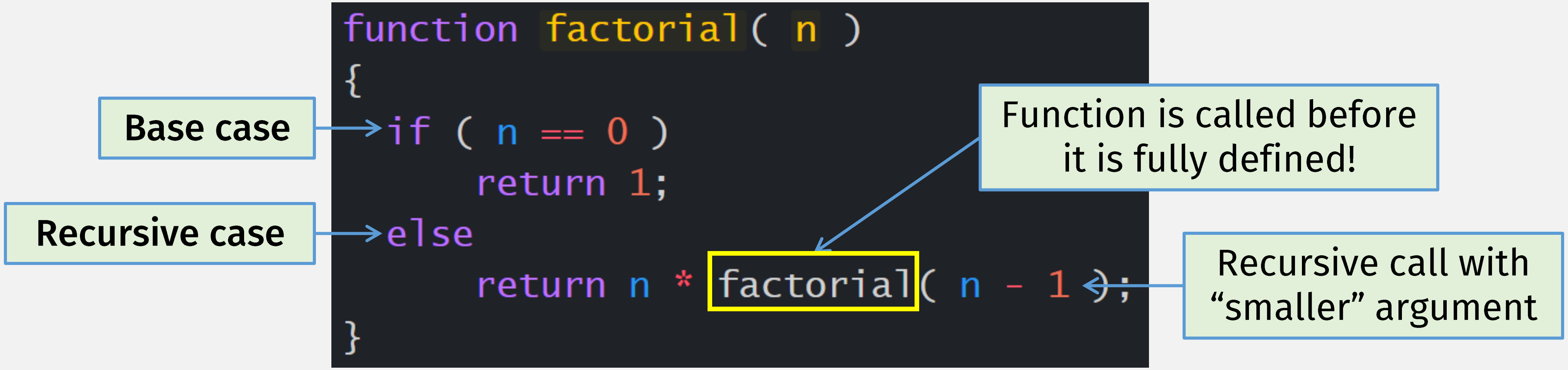
Define a **multi-step transition function:** $\hat{\delta} : Q \times \Sigma^* \to Q$

- 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)

- Base case: …

Interlude: Recursive Definitions



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

Base case

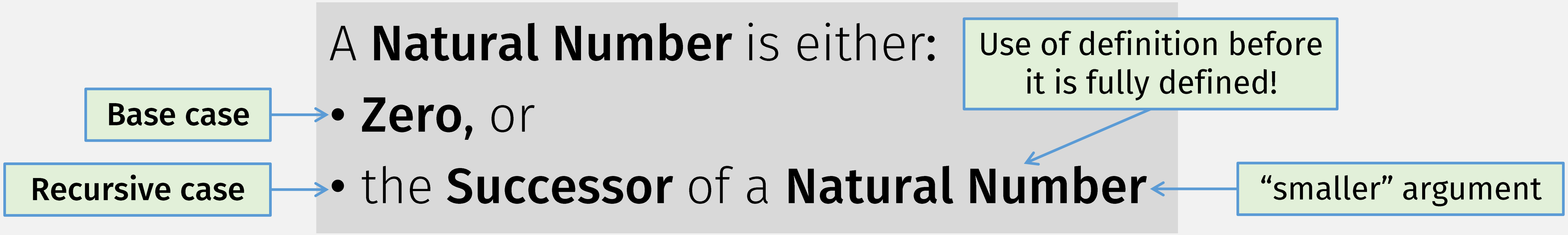
Recursive case

Function is called before it is fully defined!

Recursive call with “smaller” argument

- 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**

Use of definition before it is fully defined!

Base case

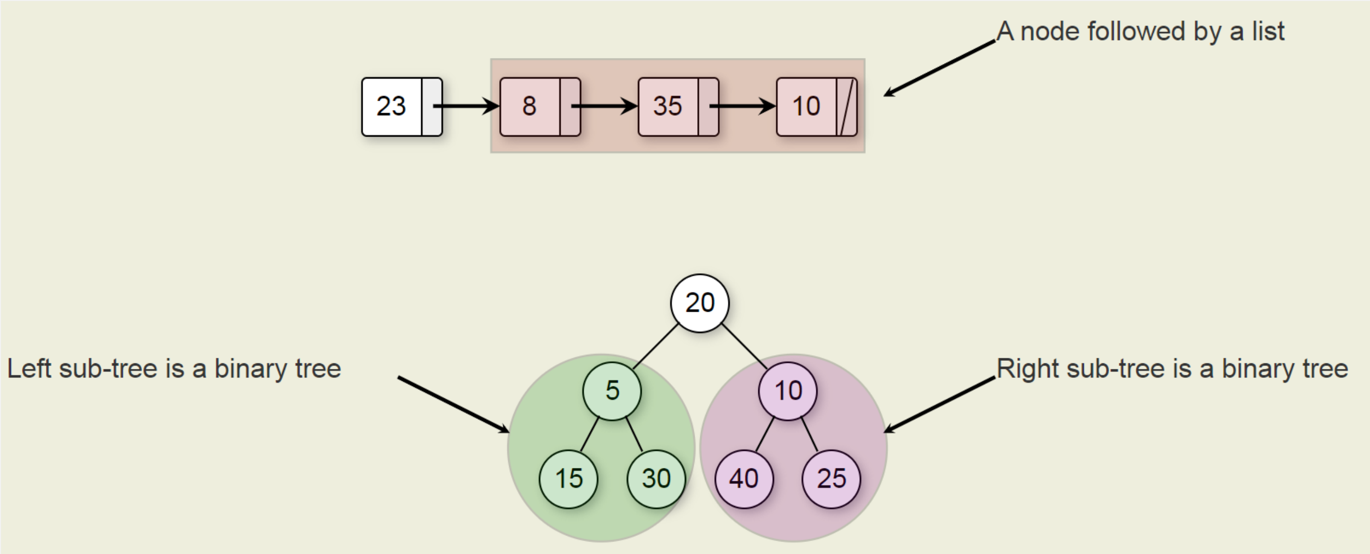
Recursive case

"smaller" argument

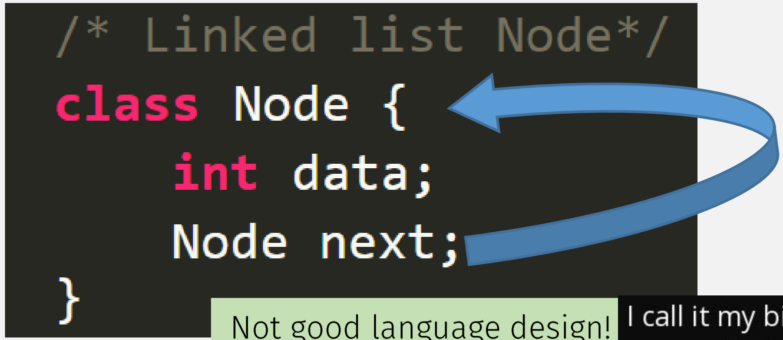
<u>Examples</u>

- **Zero**
- **Successor** of **Zero** (= "one")
- **Successor** of **Successor** of **Zero** (= "two")
- **Successor** of **Successor** of **Successor** of **Zero** (= "three") …

Recursive Data Definitions


A node followed by a list
23
8
35
10
20
5
15
30
10
40
25
Left sub-tree is a binary tree
Right sub-tree is a binary tree

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



```
/* Linked list Node*/
class Node {
    int data;
    Node next;
}
```

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

Note: **Where’s the base case???**

Not good language design!

I call it my billion-dollar mistake. It was the invention of the null reference in 1965.

— Tony Hoare —

Tony Hoare introduced Null references in ALGOL W back in 1965 "simply because it was so easy to implement", says Mr. Hoare. He talks about that decision considering it "my billion-dollar mistake".

Strings Are Defined Recursively

A **String** is either:

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

Base case

Recursive case

"smaller" argument

Remember: **all strings** are **formed with "chars"** from some **alphabet** set Σ

Σ^* = set of all possible strings!

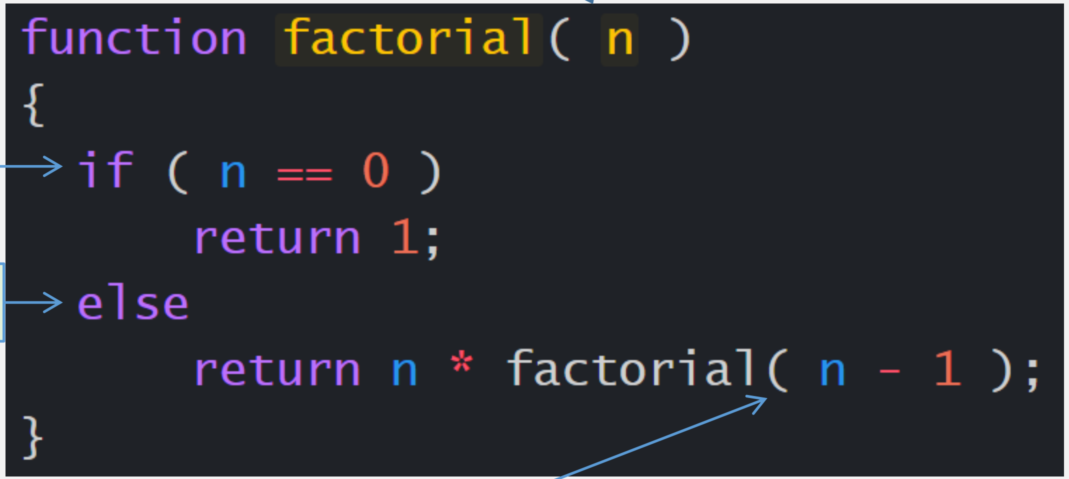
Recursive Data ⇒ Recursive Functions

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:** $\hat{\delta} : Q \times \Sigma^* \to Q$

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

Recursive Input Data
needs
Recursive Function

(Defined recursively)

Base case

A **String** is either:

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

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

A Multi-Step Transition Function

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

- 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
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "**char**" in Σ

Recursive case

"smaller" argument

(Defined recursively)

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

Recursive call

string

char

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

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

(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
- *xa* (**non-empty string**) where
  - *x* is a **string**
  - *a* 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)

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

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 $i = 1, \ldots, n$

This is still pretty verbose …

- 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)

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

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 $r_n \notin F$

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**

Languages can be **infinite**

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

$$\emptyset \qquad \{\ \}$$

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

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

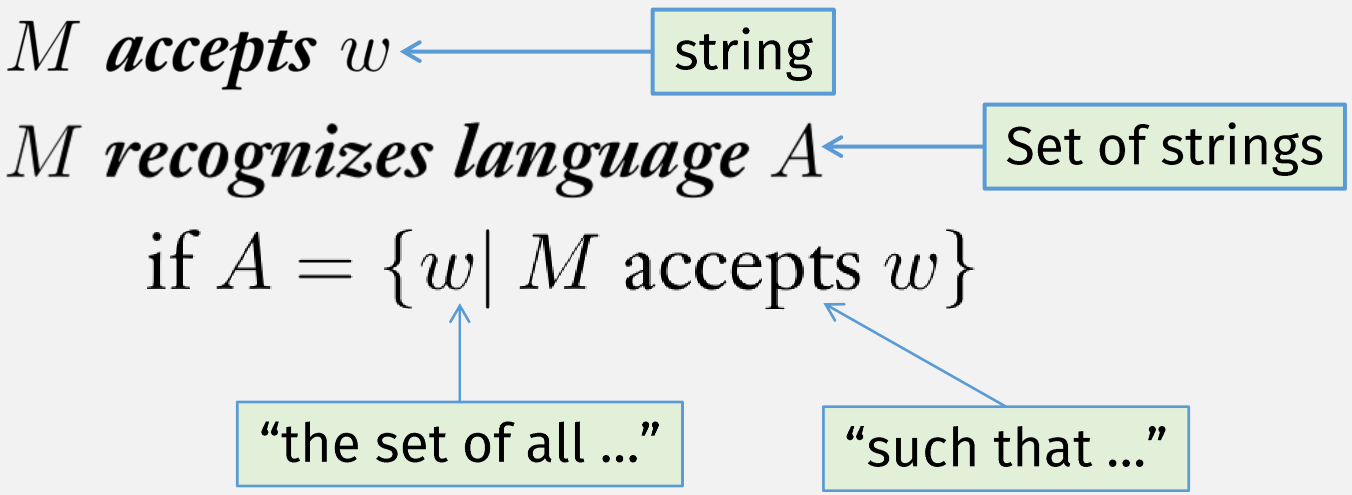
"the set of all …"

"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



M ***recognizes language*** A

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

string

Set of strings

“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*** $L(M)$

  $L(M) = \{w \mid M \text{ accepts } w\}$

Using L as function mapping **Machine → Language** is common notation

Machine and Language Terminology

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

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

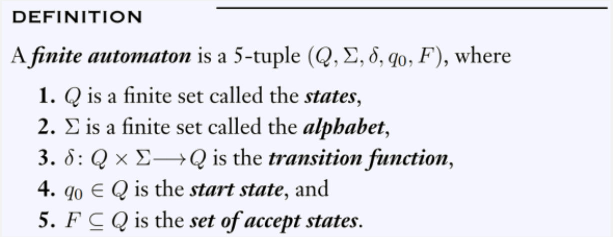
  M ***recognizes language*** $L(M)$

- **Language** of $M = L(M) = \{w \mid M \text{ accepts } w\}$

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 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta \colon Q \times \Sigma \longrightarrow Q$ 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 really about studying **sets of languages!**

Regular Languages: Definition

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

<u>Proof :</u> If a **DFA** <u>**recognizes**</u> a **language,**
then **that language** is called a **regular language**.

(modus ponens)

- If given: **a Language** A
  - Is A a **regular language?**
    - **Not necessarily!**

<u>Proof :</u> ??????

Proving That a Language is Regular

Prove: A **language** L = { … } is a **regular language**

Proof:

Statements	**Justifications**
1. DFA $M = (Q, \Sigma, \delta, q_0, F)$ (TODO: actually define ***M***) (no unbound variables!)	1. Definition of a DFA
2. **DFA M recognizes L**	2. TODO: ???
3. **If** a **DFA recognizes L, then L is a regular language**	3. **Definition of a regular language**
4. **Language L is a regular language**	4. **Stmts** 2 and 3 (and **modus ponens**)

Modus Ponens

If we can prove these:

- If P then Q
- P

Then we've proved:

- Q

A Language: strings with odd # of **1**s

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

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

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

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

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

How to prove the language is regular?

Prove there's a DFA recognizing it!

Proving That a Language is Regular

Prove: A **language** L = { … } is a **regular language**

Proof:

Statements	**Justifications**
1. DFA $M = (Q, \Sigma, \delta, q_0, F)$ (TODO: actually define *M*) (no unbound variables!)	1. Definition of a DFA
2. DFA M recognizes L	2. TODO: ???
3. If a DFA recognizes L, then L is a regular language	3. Definition of a regular language
4. Language L is a regular language	4. Stmts 2 and 3 (and **modus ponens**)

Designing Finite Automata: Tips

- Input is read only once, one char at a time (can’t go back!)

- Must decide accept/reject after that

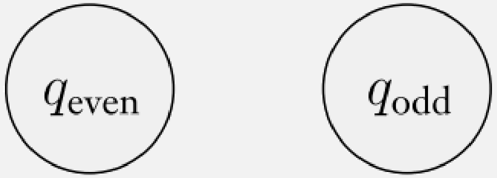
- States = the machine’s “memory”!
  - # states must be decided in advance
  - Think about what information must be “remembered”.

- Every state/symbol pair must have a defined transition (for DFAs)

- Come up with examples to help you!

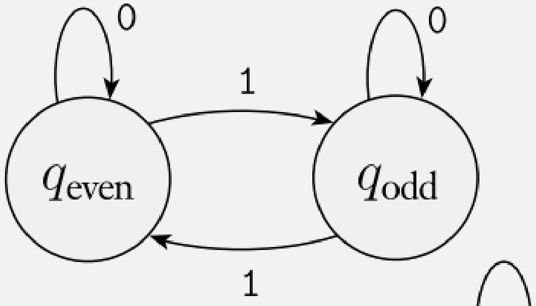
Design a DFA: accept strs with odd # **1**s

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

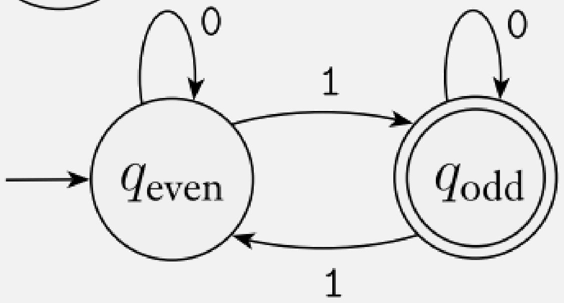


- Alphabet: **0** and **1**

- Transitions:



- Start / Accept states:



Proving That a Language is Regular

Prove: A **language** L = { … } is a **regular language**

Proof:

Statements	**Justifications**
☑ 1. DFA $M =$ See state diagram (only if problem allows!)	1. Definition of a DFA
2. DFA M recognizes L	2. TODO: ???
3. If a DFA recognizes L, then L is a regular language	3. Definition of a regular language
4. Language L is a regular language	4. Stmts 2 and 3 (and **modus ponens**)

“Prove” that DFA recognizes a language

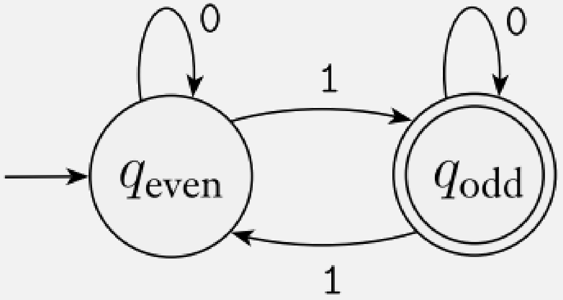
- **In-class exercise** (part 2):

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

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

Confirm the DFA:
- Accepts strings in the language
- Rejects strings not in the language

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



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”**

Proving That a Language is Regular

Prove: A **language** L = { … } is a **regular language**

Proof:

Statements	**Justifications**
1. DFA $M =$ See state diagram (only if problem allows!)	1. Definition of a DFA
2. DFA M recognizes L	☑ 2. See examples table
3. If a DFA recognizes L, then L is a regular language	3. Definition of a regular language
4. Language L is a regular language	4. Stmts 2 and 3 (and **modus ponens**)

Not a real proof, but …

In this class, **an “examples table” is sufficient to “prove” that a DFA recognizes a language**

■

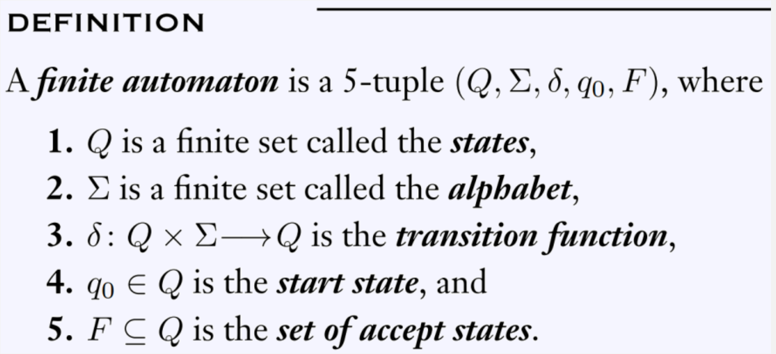
In-class exercise 2

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

- Prove: the following language is a regular language:
  - $A = \{ w \mid w$ has exactly three 1's $\}$

You will need this later in the proof anyways!

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



DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, 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 **language** L = { … } is a **regular language**

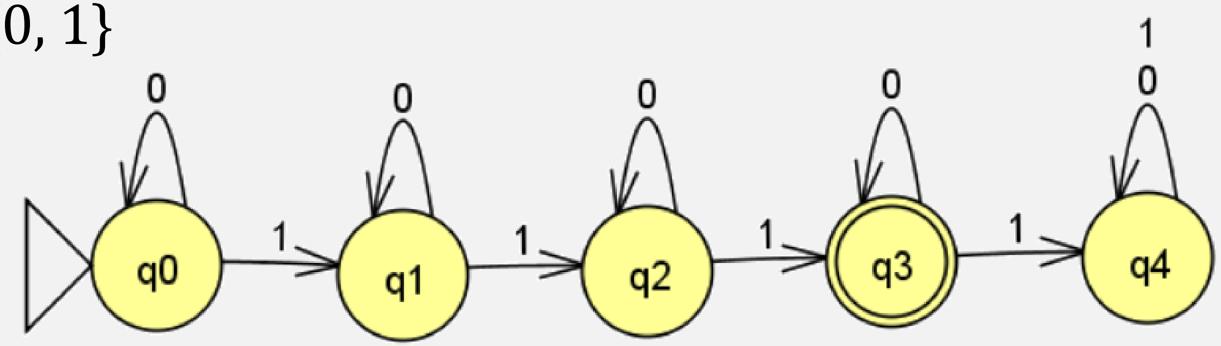
Proof:

Statements	Justifications
1. DFA $M = (Q, \Sigma, \delta, q_0, F)$ (TODO: actually define M) (no unbound variables!)	1. Definition of a DFA
2. DFA M recognizes L	2. TODO: ???
3. If a DFA recognizes L, then L is a regular language	3. Definition of a regular language
4. Language L is a regular language	4. Stmts 2 and 3 (and **modus ponens**)

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
  - $Q = \{q_0, q_1, q_2, q_3, q_{4+}\}$
- *Alphabet*: $\Sigma = \{0, 1\}$
- *Transitions*:



- *Start state*:
  - q_0
- *Accept states*:
  - $\{q_3\}$

So a DFA's computation recognizes simple string patterns?

Yes!

Have you ever used a programming language feature to recognize simple string patterns?

Submit 9/17 in-class work to gradescope