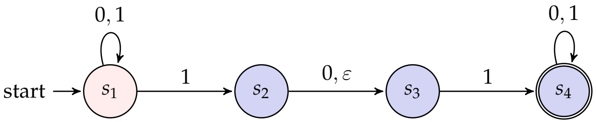
CSCI 3434: Theory of Computation

Lecture 3: Nondeterminism

Ashutosh Trivedi



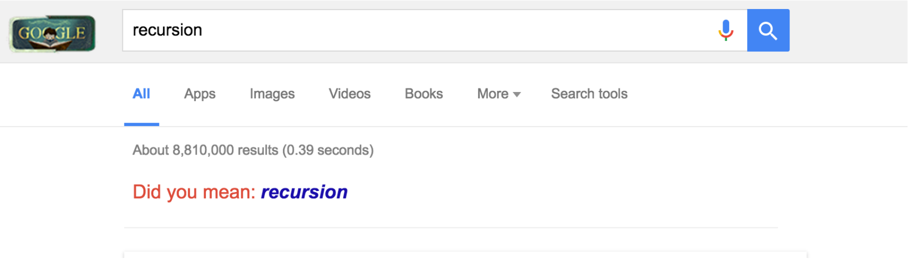
Department of Computer Science
UNIVERSITY OF COLORADO BOULDER

Recursive Definitions and Structural Induction

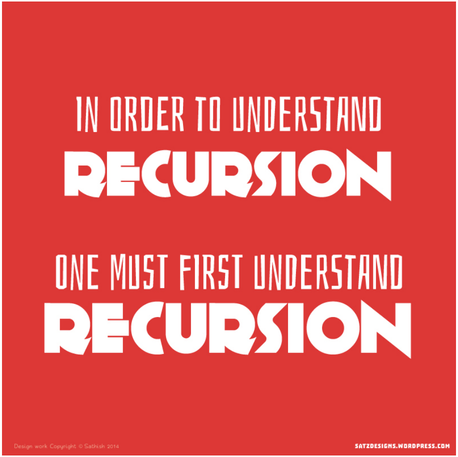
Regular Languages: Nondeterminism

Recursive Definitions

Recursive Definitions



Recursive Definitions



Recursive Definitions

Definition (Recursive Definitions.)

1. Defining an object using recursion.
2. Defining an object in terms of itself.

Recursive Definitions

Definition (Recursive Definitions.)

1. Defining an object using recursion.
2. Defining an object in terms of itself.

– Expressions over $+$ and $*$:
  – Base case: Any number of a variable is an expression.
  – Induction: If E and F are expressions then so are $E + F$, $E * F$, and (E).

Recursive Definitions

Definition (Recursive Definitions.)

1. Defining an object using recursion.
2. Defining an object in terms of itself.

- Expressions over $+$ and $*$:
  - Base case: Any number of a variable is an expression.
  - Induction: If E and F are expressions then so are $E + F$, $E * F$, and (E).
- Set of Natural numbers $\mathbb{N}$:
  - Base case: $0 \in \mathbb{N}$.
  - Induction: If $k \in \mathbb{N}$ then $k + 1 \in \mathbb{N}$.

Recursive Definitions

Definition (Recursive Definitions.)

1. Defining an object using recursion.
2. Defining an object in terms of itself.

- Expressions over $+$ and $*$:
  - Base case: Any number of a variable is an expression.
  - Induction: If E and F are expressions then so are $E + F$, $E * F$, and (E).
- Set of Natural numbers $\mathbb{N}$:
  - Base case: $0 \in \mathbb{N}$.
  - Induction: If $k \in \mathbb{N}$ then $k + 1 \in \mathbb{N}$.
- Definitions of the factorial function and Fibonacci sequence
- Definition of a singly-linked list or trees.

Structural Induction

Principle of Structural Induction

1. Let R be a recursive definition.
2. Let S be a statement about the elements defined by R.

Structural Induction

Principle of Structural Induction

1. Let R be a recursive definition.
2. Let S be a statement about the elements defined by R.
3. If the following hypotheses hold:
   - S is True for every element $b_1, \ldots, b_m$ in the base case of the definition R.

Structural Induction

Principle of Structural Induction

1. Let R be a recursive definition.
2. Let S be a statement about the elements defined by R.
3. If the following hypotheses hold:
   - S is True for every element $b_1, \ldots, b_m$ in the base case of the definition R.
   - For every element E constructed by the recursive definition from some elements $e_1, \ldots, e_n$

$$S \text{ is True for } e_1, \ldots e_n \implies S \text{ is true for } E$$

Structural Induction

Principle of Structural Induction

1. Let R be a recursive definition.
2. Let S be a statement about the elements defined by R.
3. If the following hypotheses hold:
   - S is True for every element $b_1, \ldots, b_m$ in the base case of the definition R.
   - For every element E constructed by the recursive definition from some elements $e_1, \ldots, e_n$

$$S \text{ is True for } e_1, \ldots e_n \implies S \text{ is true for } E$$

4. Then we can conclude that:
   S is True for Every Element E defined by the recursive definition R.

Structural Induction

Principle of Structural Induction

1. Let R be a recursive definition.
2. Let S be a statement about the elements defined by R.
3. If the following hypotheses hold:
   - S is True for every element $b_1, \ldots, b_m$ in the base case of the definition R.
   - For every element E constructed by the recursive definition from some elements $e_1, \ldots, e_n$

$$S \text{ is True for } e_1, \ldots e_n \implies S \text{ is true for } E$$

4. Then we can conclude that:
   S is True for Every Element E defined by the recursive definition R.

Examples:

- For all $n \geq 0$ we have that $\sum_{i=0}^{n} i = n(n+1)/2$.

Structural Induction

Principle of Structural Induction

1. Let R be a recursive definition.
2. Let S be a statement about the elements defined by R.
3. If the following hypotheses hold:
   - S is True for every element $b_1, \ldots, b_m$ in the base case of the definition R.
   - For every element E constructed by the recursive definition from some elements $e_1, \ldots, e_n$

$$S \text{ is True for } e_1, \ldots e_n \implies S \text{ is true for } E$$

4. Then we can conclude that:
   S is True for Every Element E defined by the recursive definition R.

Examples:

- For all $n \geq 0$ we have that $\sum_{i=0}^{n} i = n(n+1)/2$.
- Every expression defined has an equal number of left and right parenthesis.

Structural Induction

Principle of Structural Induction

1. Let R be a recursive definition.
2. Let S be a statement about the elements defined by R.
3. If the following hypotheses hold:
   - S is True for every element $b_1, \ldots, b_m$ in the base case of the definition R.
   - For every element E constructed by the recursive definition from some elements $e_1, \ldots, e_n$

$$S \text{ is True for } e_1, \ldots e_n \implies S \text{ is true for } E$$

4. Then we can conclude that:
   S is True for Every Element E defined by the recursive definition R.

Examples:
- For all $n \geq 0$ we have that $\sum_{i=0}^{n} i = n(n+1)/2$.
- Every expression defined has an equal number of left and right parenthesis.
- Every tree has one more node than the edges.

Structural Induction

Principle of Structural Induction

1. Let R be a recursive definition.
2. Let S be a statement about the elements defined by R.
3. If the following hypotheses hold:
   - S is True for every element $b_1, \ldots, b_m$ in the base case of the definition R.
   - For every element E constructed by the recursive definition from some elements $e_1, \ldots, e_n$

$$S \text{ is True for } e_1, \ldots e_n \implies S \text{ is true for } E$$

4. Then we can conclude that:
   S is True for Every Element E defined by the recursive definition R.

Examples:

- For all $n \geq 0$ we have that $\sum_{i=0}^{n} i = n(n+1)/2$.
- Every expression defined has an equal number of left and right parenthesis.
- Every tree has one more node than the edges.
- Other examples

Recursive Definitions and Structural Induction

Regular Languages: Nondeterminism

What are Regular Languages?

– An alphabet $\Sigma = \{a, b, c\}$ is a finite set of letters,

What are Regular Languages?

- An alphabet $\Sigma = \{a, b, c\}$ is a finite set of letters,
- The set of all strings (aka, words) Σ^* over an alphabet Σ can be recursively defined as: as :
  - Base case: $\varepsilon \in \Sigma^*$ (empty string),
  - Induction: If $w \in \Sigma^*$ then $wa \in \Sigma^*$ for all $a \in \Sigma$.

What are Regular Languages?

- An alphabet $\Sigma = \{a, b, c\}$ is a finite set of letters,
- The set of all strings (aka, words) Σ^* over an alphabet Σ can be recursively defined as: as :
  - Base case: $\varepsilon \in \Sigma^*$ (empty string),
  - Induction: If $w \in \Sigma^*$ then $wa \in \Sigma^*$ for all $a \in \Sigma$.
- A language L over some alphabet Σ is a set of strings, i.e. $L \subseteq \Sigma^*$.
- Some examples:
  - $L_{\text{even}} = \{w \in \Sigma^* : w \text{ is of even length}\}$
  - $L_{a^*b^*} = \{w \in \Sigma^* : w \text{ is of the form } a^n b^m \text{ for } n, m \geq 0\}$
  - $L_{a^n b^n} = \{w \in \Sigma^* : w \text{ is of the form } a^n b^n \text{ for } n \geq 0\}$
  - $L_{\text{prime}} = \{w \in \Sigma^* : w \text{ has a prime number of } a'\text{s}\}$
- Deterministic finite state automata define languages that require finite resources (states) to recognize.

What are Regular Languages?

- An alphabet $\Sigma = \{a, b, c\}$ is a finite set of letters,
- The set of all strings (aka, words) Σ^* over an alphabet Σ can be recursively defined as: as :
  - Base case: $\varepsilon \in \Sigma^*$ (empty string),
  - Induction: If $w \in \Sigma^*$ then $wa \in \Sigma^*$ for all $a \in \Sigma$.
- A language L over some alphabet Σ is a set of strings, i.e. $L \subseteq \Sigma^*$.
- Some examples:
  - $L_{\text{even}} = \{w \in \Sigma^* : w \text{ is of even length}\}$
  - $L_{a^*b^*} = \{w \in \Sigma^* : w \text{ is of the form } a^n b^m \text{ for } n, m \geq 0\}$
  - $L_{a^n b^n} = \{w \in \Sigma^* : w \text{ is of the form } a^n b^n \text{ for } n \geq 0\}$
  - $L_{\text{prime}} = \{w \in \Sigma^* : w \text{ has a prime number of } a'\text{s}\}$
- Deterministic finite state automata define languages that require finite resources (states) to recognize.

Definition (Regular Languages)

We call a language regular if it can be accepted by a deterministic finite state automaton.

Why they are "Regular"

- A number of widely different and equi-expressive formalisms precisely capture the same class of languages:
  - Deterministic finite state automata
  - Nondeterministic finite state automata (also with ε-transitions)
  - Kleene's regular expressions, also appeared as Type-3 languages in Chomsky's hierarchy [Cho59].
  - Monadic second-order logic definable languages [B60̈, Elg61, Tra62]
  - Certain Algebraic connection (acceptability via finite semi-group) [RS59]

Why they are "Regular"

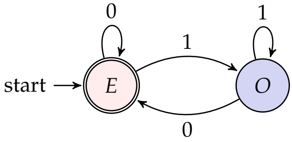
- A number of widely different and equi-expressive formalisms precisely capture the same class of languages:
  - Deterministic finite state automata
  - Nondeterministic finite state automata (also with ε-transitions)
  - Kleene's regular expressions, also appeared as Type-3 languages in Chomsky's hierarchy [Cho59].
  - Monadic second-order logic definable languages [B6̈0, Elg61, Tra62]
  - Certain Algebraic connection (acceptability via finite semi-group) [RS59]

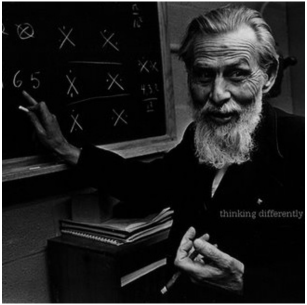
Today we show that:

Theorem (DFA=NFA=ε-NFA)

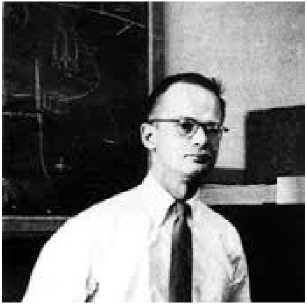
A language is accepted by a deterministic finite automaton if and only if it is accepted by a non-deterministic finite automaton.

Finite State Automata



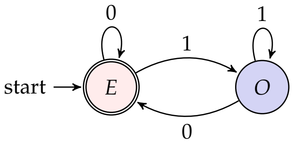


Warren S. McCullough



Walter Pitts

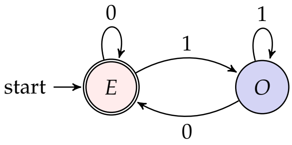
Deterministic Finite State Automata (DFA)



A finite state automaton is a tuple $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$, where:

- S is a finite set called the states;
- Σ is a finite set called the alphabet;
- $\delta : S \times \Sigma \to S$ is the transition function;
- $s_0 \in S$ is the start state; and
- $F \subseteq S$ is the set of accept states.

Deterministic Finite State Automata (DFA)

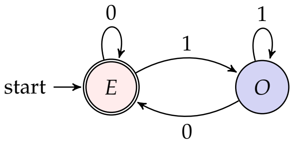


A finite state automaton is a tuple $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$, where:

- S is a finite set called the states;
- Σ is a finite set called the alphabet;
- $\delta : S \times \Sigma \to S$ is the transition function;
- $s_0 \in S$ is the start state; and
- $F \subseteq S$ is the set of accept states.

For a function $\delta : S \times \Sigma \to S$ we define extended transition function $\hat{\delta} : S \times \Sigma^* \to S$ using the following inductive definition:

Deterministic Finite State Automata (DFA)



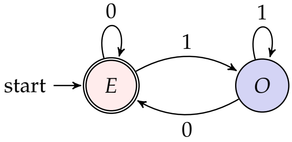
A finite state automaton is a tuple $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$, where:

- S is a finite set called the states;
- Σ is a finite set called the alphabet;
- $\delta : S \times \Sigma \to S$ is the transition function;
- $s_0 \in S$ is the start state; and
- $F \subseteq S$ is the set of accept states.

For a function $\delta : S \times \Sigma \to S$ we define extended transition function $\hat{\delta} : S \times \Sigma^* \to S$ using the following inductive definition:

$$\hat{\delta}(q, w) = \begin{cases} q & \text{if } w = \varepsilon \\ \delta(\hat{\delta}(q, x), a) & \text{if } w = xa \text{ s.t. } x \in \Sigma^* \text{ and } a \in \Sigma. \end{cases}$$

Deterministic Finite State Automata (DFA)



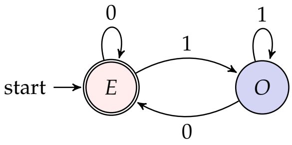
A finite state automaton is a tuple $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$, where:

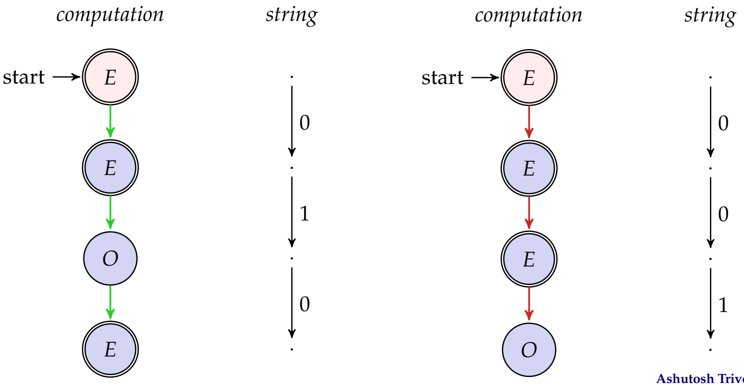
- S is a finite set called the states;
- Σ is a finite set called the alphabet;
- $\delta : S \times \Sigma \to S$ is the transition function;
- $s_0 \in S$ is the start state; and
- $F \subseteq S$ is the set of accept states.

The language $L(\mathcal{A})$ accepted by a DFA $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$ is defined as:

$$L(\mathcal{A}) \stackrel{\text{def}}{=} \{w \ : \ \hat{\delta}(w) \in F\}.$$

Computation or Run of a DFA





Deterministic Finite State Automata

Semantics using extended transition function:

- The language $L(\mathcal{A})$ accepted by a DFA $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$ is defined as:

$$L(\mathcal{A}) \stackrel{\text{def}}{=} \{w \,:\, \hat{\delta}(w) \in F\}.$$

Semantics using accepting computation:

- A computation or a run of a DFA $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$ on a string $w = a_0 a_1 \ldots a_{n-1}$ is the finite sequence

$$s_0, a_1 s_1, a_2, \ldots, a_{n-1}, s_n$$

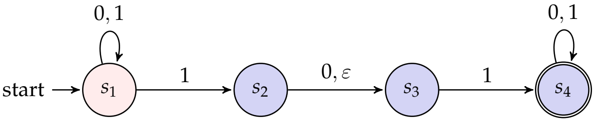
  where s_0 is the starting state, and $\delta(s_{i-1}, a_i) = s_{i+1}$.
- A string w is accepted by a DFA $\mathcal{A}$ if the last state of the unique computation of $\mathcal{A}$ on w is an accept state, i.e. $s_n \in F$.
- Language of a DFA $\mathcal{A}$

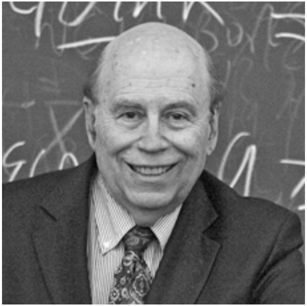
$$L(\mathcal{A}) = \{w \,:\, \text{string } w \text{ is accepted by DFA } \mathcal{A}\}.$$

Proposition

Both semantics define the same language. *Proof by induction.*

Nondeterministic Finite State Automata

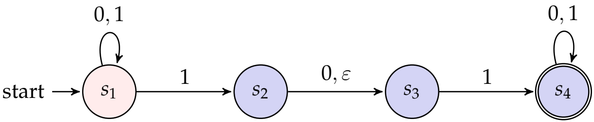




Michael O. Rabin

Dana Scott

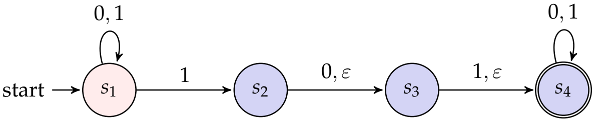
Non-deterministic Finite State Automata



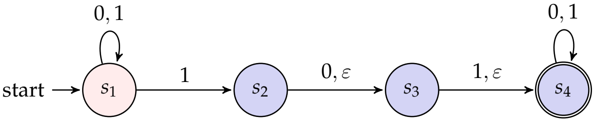
A non-deterministic finite state automaton (NFA) is a tuple $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$, where:

- S is a finite set called the states;
- Σ is a finite set called the alphabet;
- $\delta : S \times (\Sigma \cup \{\varepsilon\}) \to 2^S$ is the transition function;
- $s_0 \in S$ is the start state; and
- $F \subseteq S$ is the set of accept states.

ε-closure ECLOS

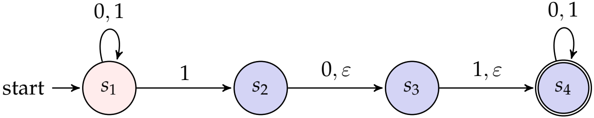


ε-closure ECLOS



– ε-closure ECLOS(s) of a state s is the set of states that can be reached from s (including itself) via ε-transitions. E.g.

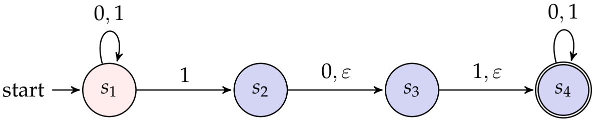
ε-closure ECLOS



– ε-closure ECLOS(s) of a state s is the set of states that can be reached from s (including itself) via ε-transitions. E.g.

$$\text{ECLOS}(s_2) = \{s_2, s_3, s_4\} \text{ and } \text{ECLOS}(s_3) = \{s_3, s_4\}$$

ε-closure ECLOS

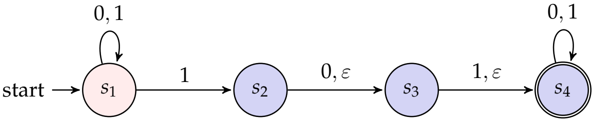


- ε-closure $\text{ECLOS}(s)$ of a state s is the set of states that can be reached from s (including itself) via ε-transitions. E.g.

$$\text{ECLOS}(s_2) = \{s_2, s_3, s_4\} \text{ and } \text{ECLOS}(s_3) = \{s_3, s_4\}$$

- $\text{ECLOS}(R) = \cup_{s \in R} \text{ECLOS}(R)$. E.g.

ε-closure ECLOS



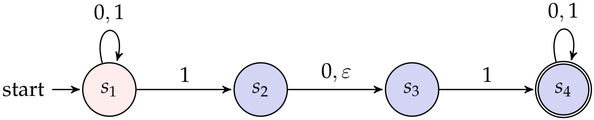
- ε-closure $\textsc{Eclos}(s)$ of a state s is the set of states that can be reached from s (including itself) via ε-transitions. E.g.

$$\textsc{Eclos}(s_2) = \{s_2, s_3, s_4\} \text{ and } \textsc{Eclos}(s_3) = \{s_3, s_4\}$$

- $\textsc{Eclos}(R) = \cup_{s \in R} \textsc{Eclos}(R)$. E.g.

$$\textsc{Eclos}(\{s_1, s_2\}) = \{s_1, s_2, s_3, s_4\}$$

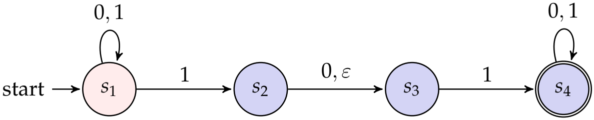
Non-deterministic Finite State Automata



A non-deterministic finite state automaton (NFA) is a tuple $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$, where:

- S is a finite set called the states;
- Σ is a finite set called the alphabet;
- $\delta : S \times (\Sigma \cup \{\varepsilon\}) \to 2^S$ is the transition function;
- $s_0 \in S$ is the start state; and
- $F \subseteq S$ is the set of accept states.

Non-deterministic Finite State Automata



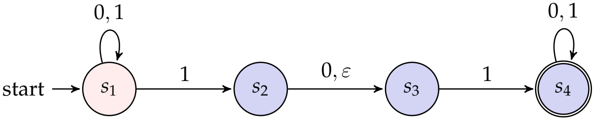
A non-deterministic finite state automaton (NFA) is a tuple $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$, where:

- S is a finite set called the states;
- Σ is a finite set called the alphabet;
- $\delta : S \times (\Sigma \cup \{\varepsilon\}) \to 2^S$ is the transition function;
- $s_0 \in S$ is the start state; and
- $F \subseteq S$ is the set of accept states.

For a function $\delta : S \times \Sigma \to 2^S$ we define extended transition function $\hat{\delta} : S \times \Sigma^* \to 2^S$ using the following inductive definition:

$$\hat{\delta}(q, w) = \begin{cases} \textsc{Eclos}(\{q\}) & \text{if } w = \varepsilon \\ \bigcup_{p \in \hat{\delta}(q,x)} \textsc{Eclos}(\delta(p, a)) & \text{if } w = xa \text{ s.t. } x \in \Sigma^* \text{ and } a \in \Sigma. \end{cases}$$

Non-deterministic Finite State Automata



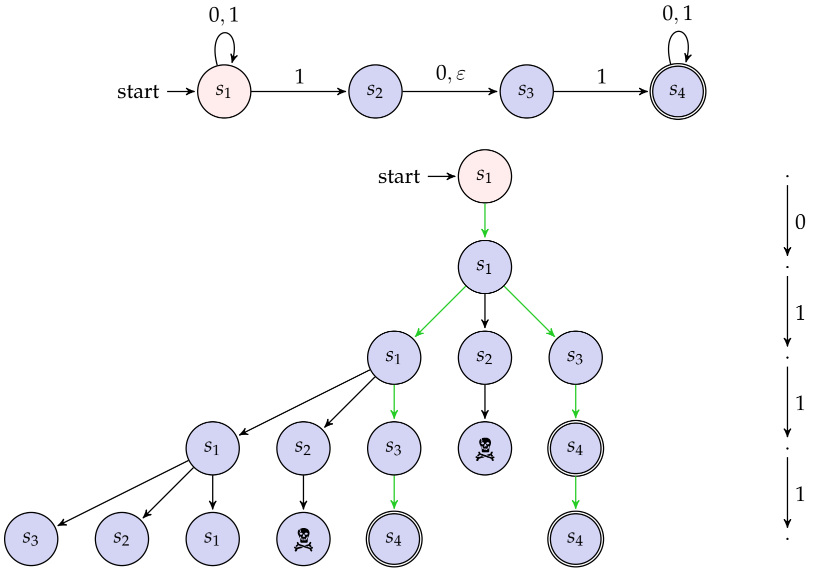
A non-deterministic finite state automaton (NFA) is a tuple $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$, where:

- S is a finite set called the states;
- Σ is a finite set called the alphabet;
- $\delta : S \times (\Sigma \cup \{\varepsilon\}) \to 2^S$ is the transition function;
- $s_0 \in S$ is the start state; and
- $F \subseteq S$ is the set of accept states.

The language $L(\mathcal{A})$ accepted by an NFA $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$ is defined as:

$$L(\mathcal{A}) \stackrel{\text{def}}{=} \{w \ : \ \hat{\delta}(w) \cap F \neq \emptyset\}.$$

Computation or Run of an NFA



Non-deterministic Finite State Automata

Semantics using extended transition function:

- The language $L(\mathcal{A})$ accepted by an NFA $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$ is defined:

$$L(\mathcal{A}) \stackrel{\text{def}}{=} \{w \,:\, \hat{\delta}(w) \cap F \neq \emptyset\}.$$

Semantics using accepting computation:

- A computation or a run of a NFA on a string $w = a_0 a_1 \dots a_{n-1}$ is a finite sequence

$$s_0, r_1, s_1, r_2, \dots, r_{k-1}, s_n$$

  where s_0 is the starting state, and $s_{i+1} \in \delta(s_{i-1}, r_i)$ and $r_0 r_1 \dots r_{k-1} = a_0 a_1 \dots a_{n-1}$.
- A string w is accepted by an NFA $\mathcal{A}$ if the last state of some computation of $\mathcal{A}$ on w is an accept state $s_n \in F$.
- Language of an NFA $\mathcal{A}$

$$L(\mathcal{A}) = \{w \,:\, \text{string } w \text{ is accepted by NFA } \mathcal{A}\}.$$

Proposition

Both semantics define the same language. *Proof by induction.*

Why study NFA?

NFA are often more convenient to design than DFA, e.g.:

- $\{w : w$ contains 1 in the third last position$\}$.
- $\{w :: w$ is a multiple of 2 or a multiple of 3$\}$.
- Union and intersection of two DFAs as an NFA
- Exponentially succinct than DFA
  - Consider the language of strings having n-th symbol from the end is 1.
  - DFA has to remember last n symbols, and
  - hence any DFA needs at least 2^n states to accept this language.

Why study NFA?

NFA are often more convenient to design than DFA, e.g.:

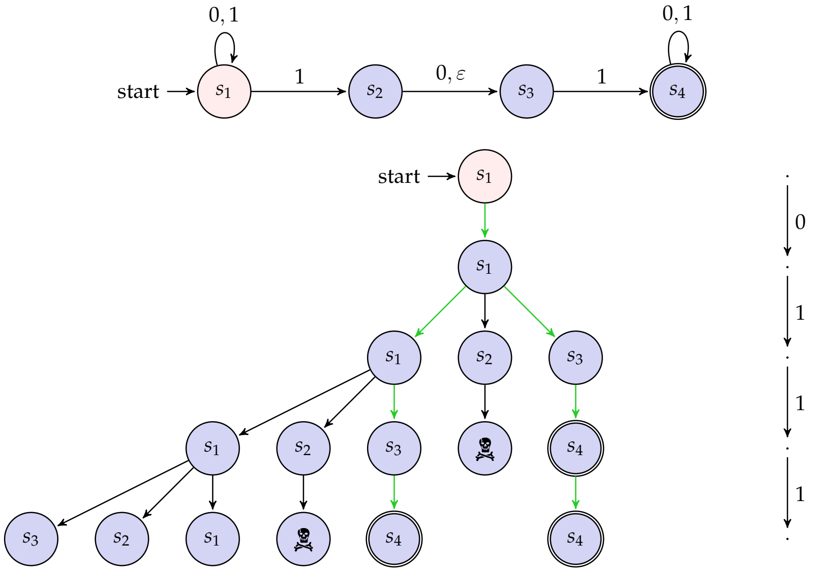
- $\{w : w$ contains 1 in the third last position$\}$.
- $\{w :: w$ is a multiple of 2 or a multiple of 3$\}$.
- Union and intersection of two DFAs as an NFA
- Exponentially succinct than DFA
  - Consider the language of strings having n-th symbol from the end is 1.
  - DFA has to remember last n symbols, and
  - hence any DFA needs at least 2^n states to accept this language.

And, surprisingly perhaps:

Theorem (DFA=NFA)

Every non-deterministic finite automaton has an equivalent (accepting the same language) deterministic finite automaton. *Subset construction.*

Computation of an NFA: An observation



ε-free NFA = DFA

Let $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$ be an ε-free NFA. Consider the DFA $Det(\mathcal{A}) = (S', \Sigma', \delta', s_0', F')$ where

- $S' = 2^S$,
- $\Sigma' = \Sigma$,
- $\delta' : 2^S \times \Sigma \to 2^S$ such that $\delta'(P, a) = \bigcup_{s \in P} \delta(s, a)$,
- $s_0' = \{s_0\}$, and
- $F' \subseteq S'$ is such that $F' = \{P \ : \ P \cap F \neq \emptyset\}$.

Theorem (ε-free NFA = DFA)

$L(\mathcal{A}) = L(Det(\mathcal{A}))$.

By induction, hint $\hat{\delta}(s_0, w) = \hat{\delta}'(\{s_0\}, w)$.

Proof of correctness: $L(\mathcal{A}) = L(Det(\mathcal{A}))$.

The proof follows from the observation that $\hat{\delta}(s_0, w) = \hat{\delta}'(\{s_0\}, w)$.

Proof of correctness: $L(\mathcal{A}) = L(Det(\mathcal{A}))$.

The proof follows from the observation that $\hat{\delta}(s_0, w) = \hat{\delta}'(\{s_0\}, w)$. We prove it by induction on the length of w.

– Base case: Let w be ε. The base case follows immediately from the definition of extended transition functions:

$$\hat{\delta}(s_0, \varepsilon) = s_0 \text{ and } \hat{\delta}'(\{s_0\}, \varepsilon) = \{s_0\}.$$

Proof of correctness: $L(\mathcal{A}) = L(Det(\mathcal{A}))$.

The proof follows from the observation that $\hat{\delta}(s_0, w) = \hat{\delta}'(\{s_0\}, w)$. We prove it by induction on the length of w.

- Base case: Let w be ε. The base case follows immediately from the definition of extended transition functions:

$$\hat{\delta}(s_0, \varepsilon) = s_0 \text{ and } \hat{\delta}'(\{s_0\}, \varepsilon) = \{s_0\}.$$

- Induction Step: Let $w = xa$ where $x \in \Sigma^*$ and $a \in \Sigma$. Now observe,

$$\hat{\delta}(s_0, xa) = \bigcup_{s \in \hat{\delta}(s_0, x)} \delta(s, a), \text{by definition of } \hat{\delta}.$$

Proof of correctness: $L(\mathcal{A}) = L(Det(\mathcal{A}))$.

The proof follows from the observation that $\hat{\delta}(s_0, w) = \hat{\delta}'(\{s_0\}, w)$. We prove it by induction on the length of w.

- Base case: Let w be ε. The base case follows immediately from the definition of extended transition functions:

$$\hat{\delta}(s_0, \varepsilon) = s_0 \text{ and } \hat{\delta}'(\{s_0\}, \varepsilon) = \{s_0\}.$$

- Induction Step: Let $w = xa$ where $x \in \Sigma^*$ and $a \in \Sigma$. Now observe,

$$\begin{aligned} \hat{\delta}(s_0, xa) &= \bigcup_{s \in \hat{\delta}(s_0, x)} \delta(s, a), \text{by definition of } \hat{\delta}. \\ &= \bigcup_{s \in \hat{\delta}'(\{s_0\}, x)} \delta(s, a), \text{from inductive hypothesis.} \end{aligned}$$

Proof of correctness: $L(\mathcal{A}) = L(Det(\mathcal{A}))$.

The proof follows from the observation that $\hat{\delta}(s_0, w) = \hat{\delta}'(\{s_0\}, w)$. We prove it by induction on the length of w.

- Base case: Let w be ε. The base case follows immediately from the definition of extended transition functions:

$$\hat{\delta}(s_0, \varepsilon) = s_0 \text{ and } \hat{\delta}'(\{s_0\}, \varepsilon) = \{s_0\}.$$

- Induction Step: Let $w = xa$ where $x \in \Sigma^*$ and $a \in \Sigma$. Now observe,

$$\begin{aligned}
\hat{\delta}(s_0, xa) &= \bigcup_{s \in \hat{\delta}(s_0, x)} \delta(s, a), \text{by definition of } \hat{\delta}. \\
&= \bigcup_{s \in \hat{\delta}'(\{s_0\}, x)} \delta(s, a), \text{from inductive hypothesis.} \\
&= \delta'(\hat{\delta}'(\{s_0\}, x), a), \text{ from definition } \delta'(P, a) = \bigcup_{s \in P} \delta(s, a).
\end{aligned}$$

Proof of correctness: $L(\mathcal{A}) = L(Det(\mathcal{A}))$.

The proof follows from the observation that $\hat{\delta}(s_0, w) = \hat{\delta}'(\{s_0\}, w)$. We prove it by induction on the length of w.

- Base case: Let w be ε. The base case follows immediately from the definition of extended transition functions:

$$\hat{\delta}(s_0, \varepsilon) = s_0 \text{ and } \hat{\delta}'(\{s_0\}, \varepsilon) = \{s_0\}.$$

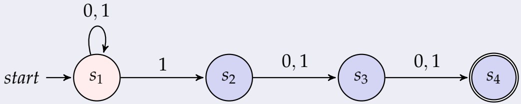
- Induction Step: Let $w = xa$ where $x \in \Sigma^*$ and $a \in \Sigma$. Now observe,

$$\begin{aligned}
\hat{\delta}(s_0, xa) &= \bigcup_{s \in \hat{\delta}(s_0, x)} \delta(s, a), \text{by definition of } \hat{\delta}. \\
&= \bigcup_{s \in \hat{\delta}'(\{s_0\}, x)} \delta(s, a), \text{from inductive hypothesis.} \\
&= \delta'(\hat{\delta}'(\{s_0\}, x), a), \text{ from definition } \delta'(P, a) = \bigcup_{s \in P} \delta(s, a). \\
&= \hat{\delta}'(\{s_0\}, xa), \text{ by definition of } \hat{\delta}'.
\end{aligned}$$

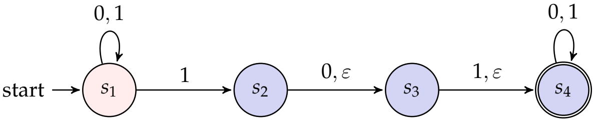
Equivalence of NFA and DFA

Exercise (In class)

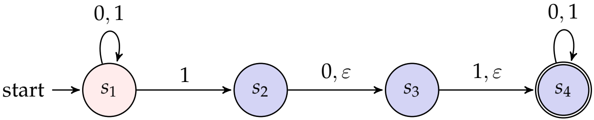
Determinize the following automaton:



NFA with ε transitions = DFA

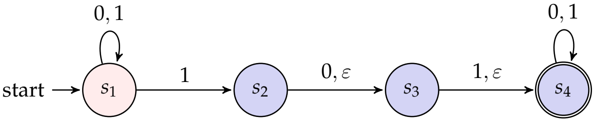


NFA with ε transitions = DFA



– ε-closure ECLOS(s) of a state s is the set of states that can be reached from s (including itself) via ε-transitions. E.g.

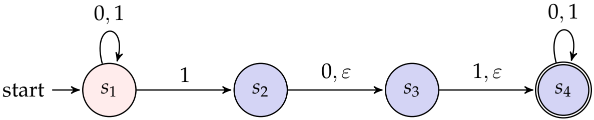
NFA with ε transitions = DFA



– ε-closure $\textsc{Eclos}(s)$ of a state s is the set of states that can be reached from s (including itself) via ε-transitions. E.g.

$$\textsc{Eclos}(s_2) = \{s_2, s_3, s_4\} \text{ and } \textsc{Eclos}(s_3) = \{s_3, s_4\}$$

NFA with ε transitions = DFA

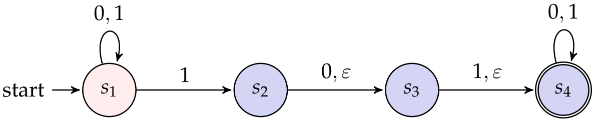


- ε-closure $\textsc{Eclos}(s)$ of a state s is the set of states that can be reached from s (including itself) via ε-transitions. E.g.

$$\textsc{Eclos}(s_2) = \{s_2, s_3, s_4\} \text{ and } \textsc{Eclos}(s_3) = \{s_3, s_4\}$$

- $\textsc{Eclos}(R) = \cup_{s \in R} \textsc{Eclos}(R)$. E.g.

NFA with ε transitions = DFA



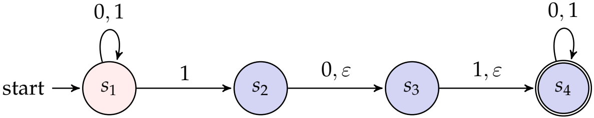
- ε-closure $\text{ECLOS}(s)$ of a state s is the set of states that can be reached from s (including itself) via ε-transitions. E.g.

$$\text{ECLOS}(s_2) = \{s_2, s_3, s_4\} \text{ and } \text{ECLOS}(s_3) = \{s_3, s_4\}$$

- $\text{ECLOS}(R) = \cup_{s \in R} \text{ECLOS}(R)$. E.g.

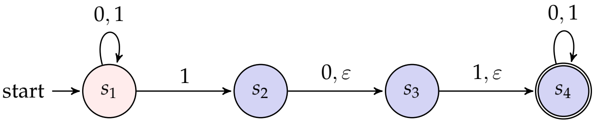
$$\text{ECLOS}(\{s_1, s_2\}) = \{s_1, s_2, s_3, s_4\}$$

NFA with ε transitions = DFA



- Let $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$ be an ε-free NFA. Consider the DFA $Det(\mathcal{A}) = (S', \Sigma', \delta', s'_0, F')$ where
  - $S' = 2^S$,
  - $\Sigma' = \Sigma$,
  - $\delta' : 2^S \times \Sigma \to 2^S$ such that $\delta'(P, a) = \bigcup_{s \in P} \textsc{Eclos}(\delta(s, a))$,
  - $s'_0 = \textsc{Eclos}(\{s_0\})$, and
  - $F' \subseteq S'$ is such that $F' = \{P \ : \ P \cap F \neq \emptyset\}$.

NFA with ε transitions = DFA



- Let $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$ be an ε-free NFA. Consider the DFA $Det(\mathcal{A}) = (S', \Sigma', \delta', s'_0, F')$ where
  - $S' = 2^S$,
  - $\Sigma' = \Sigma$,
  - $\delta' : 2^S \times \Sigma \to 2^S$ such that $\delta'(P, a) = \bigcup_{s \in P} \text{ECLOS}(\delta(s, a))$,
  - $s'_0 = \text{ECLOS}(\{s_0\})$, and
  - $F' \subseteq S'$ is such that $F' = \{P : P \cap F \neq \emptyset\}$.

Theorem (NFA with ε-transitions = DFA)

$L(\mathcal{A}) = L(Det(\mathcal{A}))$. *By induction, hint $\hat{\delta}(s_0, w) = \hat{\delta}'(\{s_0\}, w)$.*

J. R. Büchi.
Weak second-order arithmetic and finite automata.
Zeitschrift für Mathematische Logik und Grundlagen der Mathematik, 6(1–6):66–92, 1960.

Noam Chomsky.
On certain formal properties of grammars.
Information and Control, 2(2):137 – 167, 1959.

C. C. Elgot.
Decision problems of finite automata design and related arithmetics.
In Transactions of the American Mathematical Society, 98(1):21–51, 1961.

M. O. Rabin and D. Scott.
Finite automata and their decision problems.
IBM Journal of Research and Developmen, 3(2):114–125, 1959.

B. A. Trakhtenbrot.
Finite automata and monadic second order logic.
Siberian Mathematical Journal, 3:101–131, 1962.