Formal Languages and Automata, 2025 Fall semester

Lec 01. Intro & DFA

Eunjung Kim

Fundamental questions for CS

- What do we mean by "computation"?
  $\rightsquigarrow$ "Computation is to solve a problem by an effective manner."
  Vague.
- What is a computer? Why 'computers' are all called computers?
- What can it do and cannot?
- Computable vs not computable problems? 'Efficiently' computable problems and those which are not?
- Can anyone on earth devise a fundamentally more powerful computer? (An alien? In another universe?)

Two key features of Computation

What IS computed?

&

What COMPUTES it?

Throughout this course, we shall learn that these two features are intrinsically related.

How to express the object for computation: alphabet

Alphabet

- Alphabet, usually denoted as Σ, is a finite and nonempty set of symbols.
- Examples of alphabet: $\Sigma = \{0, 1\}$, $\{a, b, \ldots, z\}$, the set of all ASCII characters, etc.

How to express the object for computation: string

String

- String (a.k.a. word) is a finite sequence of symbols over Σ.
- Length of a string: number of symbols.
- Length-0 is a string itself, often denoted as ϵ.
- Σ^i: the set of all strings of length i.

How to express the object for computation: concatenation

Concatenation

- Operation on two strings.
- x, y are strings $\rightsquigarrow$ their concatenation xy is a string.
- $\epsilon x = x\epsilon =$?

How to express the object for computation: language

Language

- Language (over alphabet Σ) is a set of strings over Σ.
- Simply put, $L \subseteq \Sigma^*$.
- Here, Σ^* is a set of all strings of finite length, $\Sigma^* := \bigcup_{i \geq 0} \Sigma^i$.
- Examples of languages: ...
- Both $\emptyset$ and $\{\epsilon\} (= \Sigma^0)$ are languages.

Two key features of Computation

- Any well-formulated information can be represented as a string of 0 and 1, or any finite alphabet Σ.
- The object for computation can be stated as a function.

Compute WHAT

computational problem $\Leftrightarrow$ compute a function $f : \Sigma^* \to \Sigma^*$.

Decision problem and Language

Compute **WHAT**

a decision problem $\Leftrightarrow$ compute a function $f : \Sigma^* \to \{0, 1\}$.
$\Leftrightarrow$ given $x \in \Sigma^*$, decide if $x \in L$
where $L = \{s : f(s) = 1\} \subseteq \Sigma^*$.

- Decision problem, or equivalently "membership test for a language", is easier to handle.
- ...while capable of capturing the essence of important computational problems.

Two key features of Computation

WHAT computes a function / language?

- Let us agree: "computing a function f" means "there is an ~~effective method~~ algorithm which outputs $f(x)$ for each input x".
- The concept of "algorithm" is still vague.

Two key features of Computation

What do we expect for an algorithm, intuitively?

- ⇝ a finite number of finitely describable instructions.
- ⇝ each instruction and what to do next are unambiguous.
- ⇝ all the basic operation should be executable by the concerned executor.
- ⇝ terminates at some point (i.e. in finite number of steps)

Toward a rigorous notion of algorithm

A mathematically rigorous description of an executor (computing device/machine...) and instructions is needed.

(Our) Model of Computation

Exercutor(machine) constituents:

- an alphabet Σ it recognizes,
- a gadget to read an input $x \in \Sigma^*$,
- a finite set of states to recognize its status ("where am I?"),
- memory to write and read later.

Basic operation:

- read one alphabet from input tape (or from memory),
- update its internal state,
- move the header (only in one fixed direction, or both direction, or neither) on input tape or memory,
- write/change on memory tape.

Set-up

- Concatenation xy of x and y.
- Cartesian product $A \times B$
- Notations: Σ, Σ^i, ϵ, Σ^*.
- Computing a function $f : \Sigma^* \to \Gamma^*$ means ...
- Special function $f : \Sigma^* \to \{0, 1\} \rightsquigarrow$ language.
- Language: a subset A of Σ^*, indicator function f_A.
- Computing $f_A \Leftrightarrow$ membership test for A

Finite (state) automata

Example: automatic door

Model of computation mimicking a simple computing device

- no/limited memory,
- basic operations: read one symbol from the input, update the state, and move to the next position in input.

(State) Transition Diagram

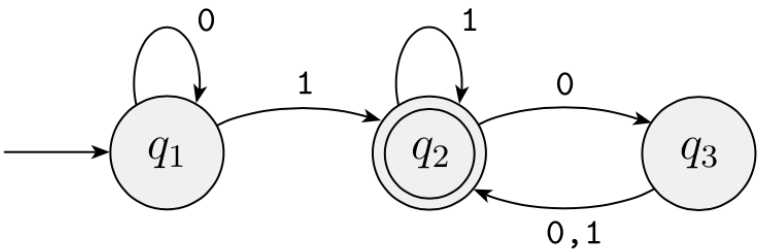


Figure 1.4, Sipser 2012.

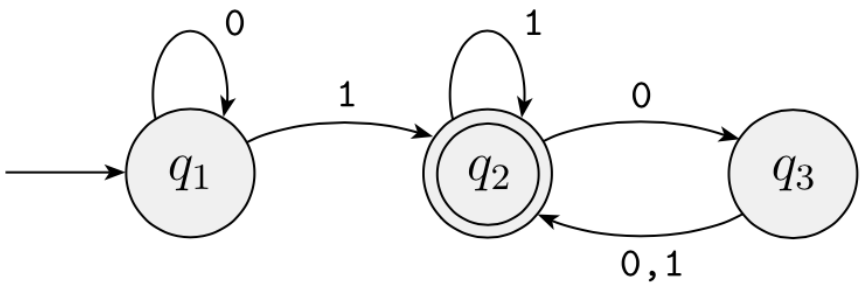
Strings accepted by M

The set of all $w \in \{0,1\}^*$ such that...

Formal definition

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

- Q a finite set called the states,
- Σ a finite set called the alphabet,
- δ a function from $Q \times \Sigma$ to Q called the (state) transition function,
- $q_0 \in Q$ the start state (a.k.a. initial states),
- $F \subseteq Q$ the set of accept states (a.k.a. final states).



Transition Diagram, Transition Table

(Other than listing the transition function) two common ways to express transition function.

Language recognized by FA

Definition

- Let M be a finite automaton.
- A string $w \in \Sigma^*$ is accepted by a finite automaton M if M ends in an accept state upon reading the entire w.
- $L(M)$ denotes the set of all strings accepted by M.
- A language A is said to be recognized by M if $A = L(M)$.

Examples of finite automata

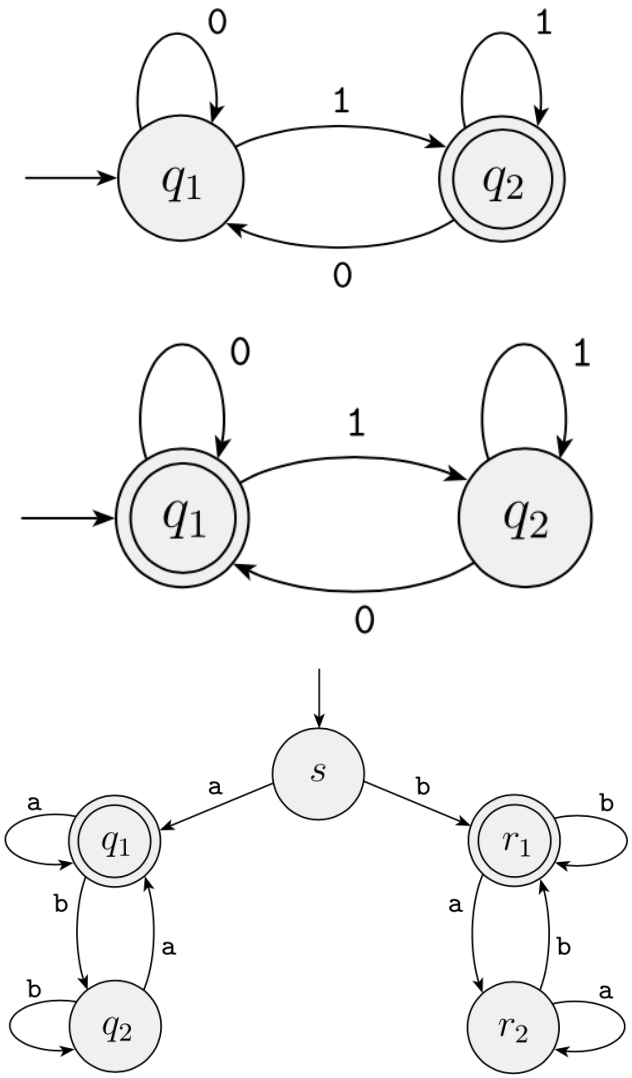


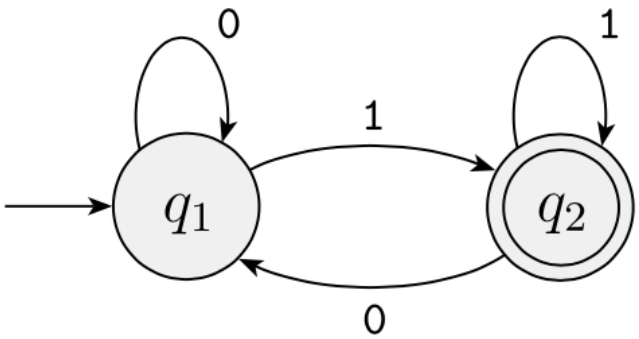
Figure 1.7, 9, 12 from Sipser 2012.

Lec 02. More on DFA & Nondeterministic Finite Automata

Eunjung Kim

Formal definition of computation

- Let $w = w_1 w_2 \cdots w_n \in \Sigma^*$, where $w_i \in \Sigma$.
- The extended transition function $\hat{\delta}$ is a mapping from $Q \times \Sigma^*$ to Q defined as: $\hat{\delta}(q, w) = q'$ if there is a sequence of states $r_0, \ldots, r_n$ in Q such that
  - $r_0 = q$,
  - $r_i = \delta(r_{i-1}, w_i)$ for every $1 \leq i \leq n$,
  - $r_n = q'$
- Equivalently, there is a walk in the transition diagram of M from q to q' labelled by w.

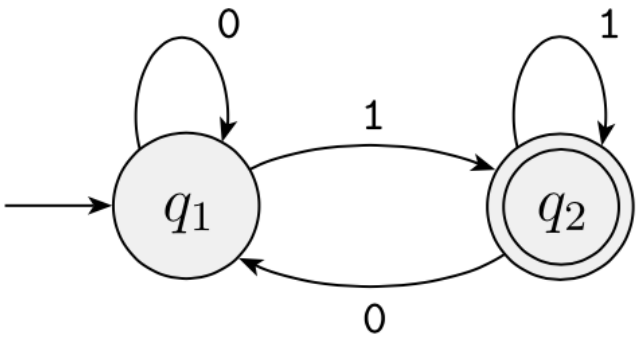


Computation history

- Configuration of a finite automata $M = (Q, \Sigma, \delta, q_0, F)$ is a pair $(q, w) \in Q \times \Sigma^*$.
- We interpret a configuration (q, w) as...
- $(q, w) \rightsquigarrow_M (q', w')$ if...
- $(q, w) \rightsquigarrow_M^* (q', w')$ if there is...
- A sequence of configuration is a computation history if the first configuration is in the form (q_0, w) for some $w \in \Sigma^*$, and each contiguous configurations are related by $\rightsquigarrow_M$ or $\rightsquigarrow_M^*$.
- A computation history is an accepting computation history if the last configuration is in the form ??????.

DFA M ACCEPTS A STRING

- Let $w_1 w_2 \cdots w_n$ be a string in Σ^* with $w_i \in \Sigma$ for each i.
- $M = (Q, \Sigma, \delta, q_0, F)$ accepts w if
  - $\hat{\delta}(q_0, w) \in F$, or equivalently
  - In the transition diagram of M, there is an walk from q_0 to an accept state labelled by w.



Language recognized by DFA

Definition: language recognized by DFA

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automata.
- A string $w \in \Sigma^*$ is accepted by M if
  - $\hat{\delta}(q_0, w) \in F$, or equivalently
  - in the transition diagram of M, there is an walk from q_0 to an accept state labelled by w.
- Let $L(M)$ be the set of all strings which are accepted by M.
- A language A is said to be recognized by M if $A = L(M)$.

Regular Language

Regular language = recognized by some DFA

- A language L over a finite alphabet is said to be regular if there is a (deterministic) finite-state automaton M which recognizes L.

From Languages to DFA: examples

Show that the following language is regular.

- $L = \{$all 0,1-strings containing 01$\}$
- $L = \{$ all 0,1-strings containing exactly even numbers of 0's and 1's respectively $\}$.
- $L = \{$ all strings containing at least two a's$\} \subseteq \{a, b\}^*$.
- $L = \{awa : w \in \{a, b\}^*\}$.

From Languages to DFA: examples

Suppose $L \subseteq \Sigma^*$ is regular. Is the complement of L, i.e. $\Sigma^* - L$, is regular?

From Languages to DFA: examples

$L = \{awa : w \in \{a, b\}^*\}$, $L^2 = \{aw_1aaw_2a : w_i \in \{a, b\}^*\}$

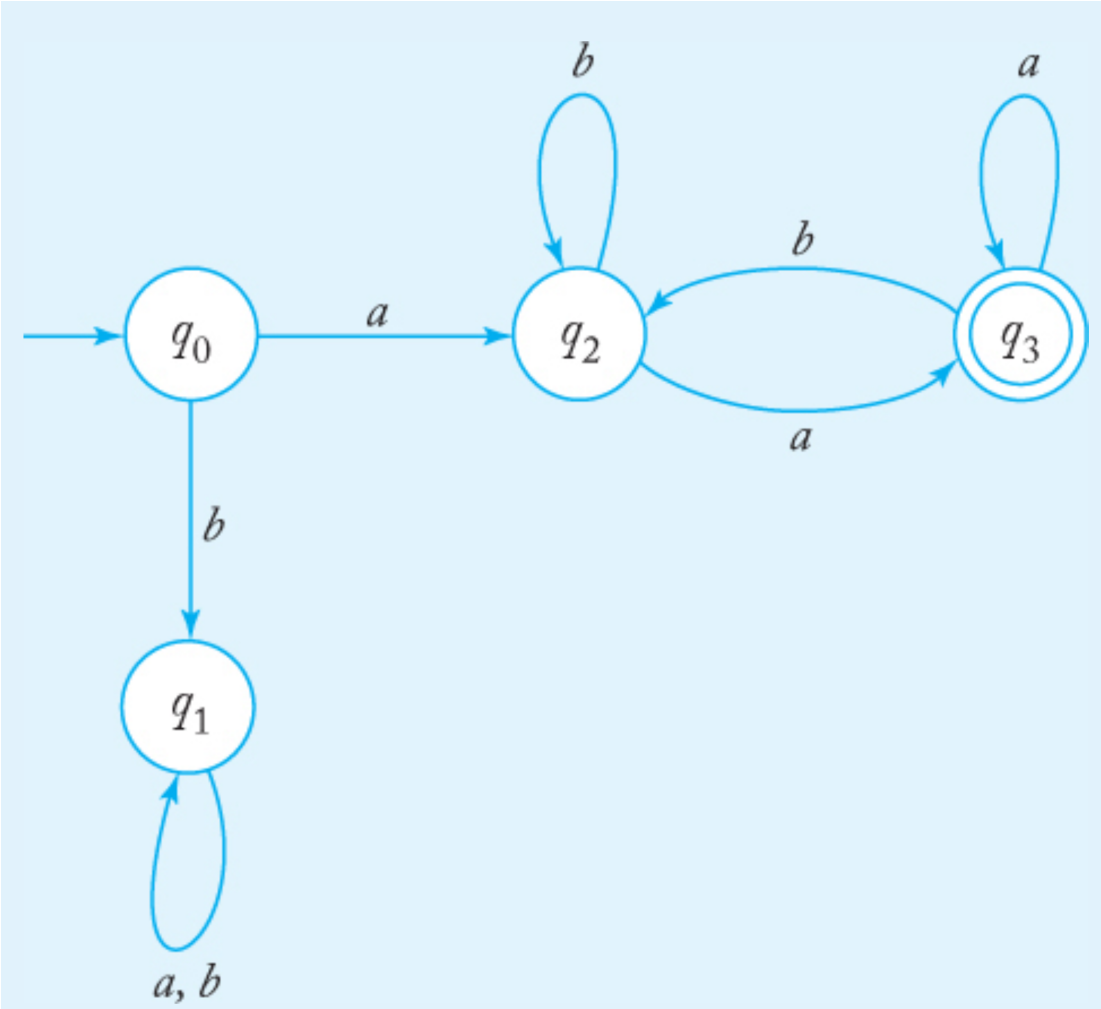


Figure 2.6 from Linz 2017.

NONDETERMINISM

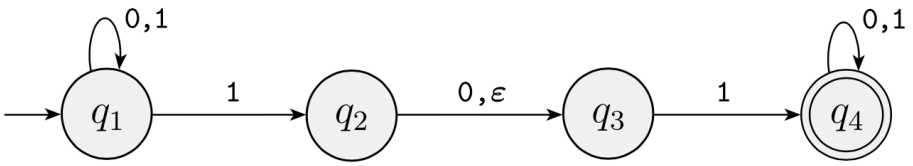


Figure 1.27, Sipser 2012.

	Deterministic FA	Nondeterministic FA
each state & symbol	one leaving arc	multiple arcs or none
labels	Σ	$\Sigma \cup \{\epsilon\}$
computation history	single path	multiple paths (tree)

Nondeterminism: computation tree and ϵ

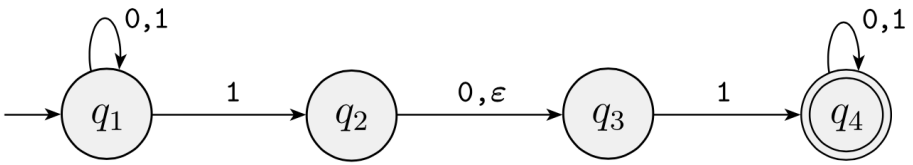
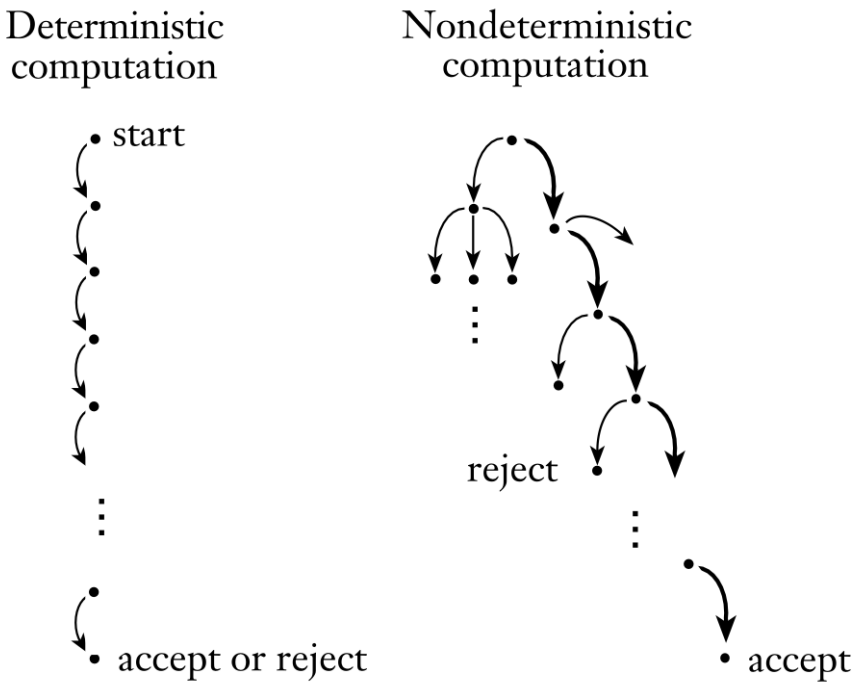


Figure 1.27, Sipser 2012.



Examples of NFA

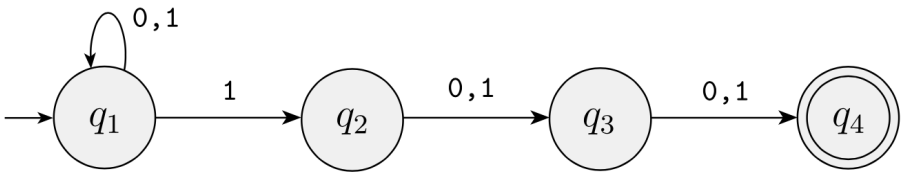


Figure 1.31, Sipser 2012.

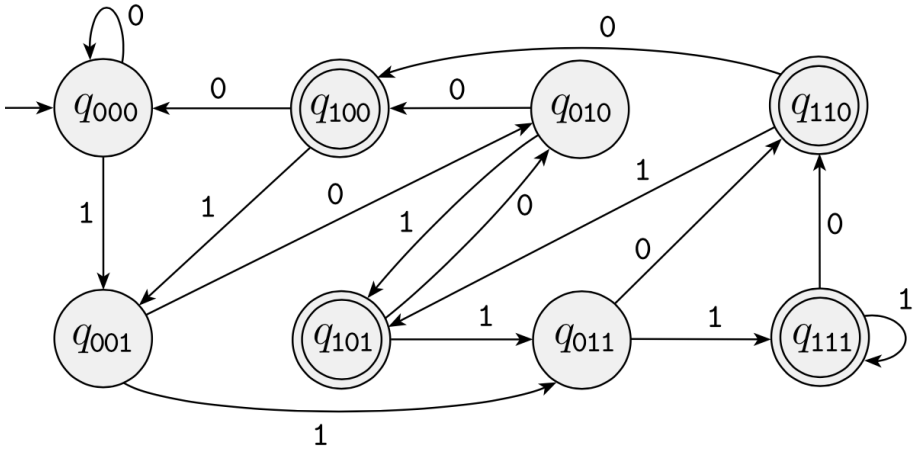


Figure 1.32, Sipser 2012.

Examples of NFA

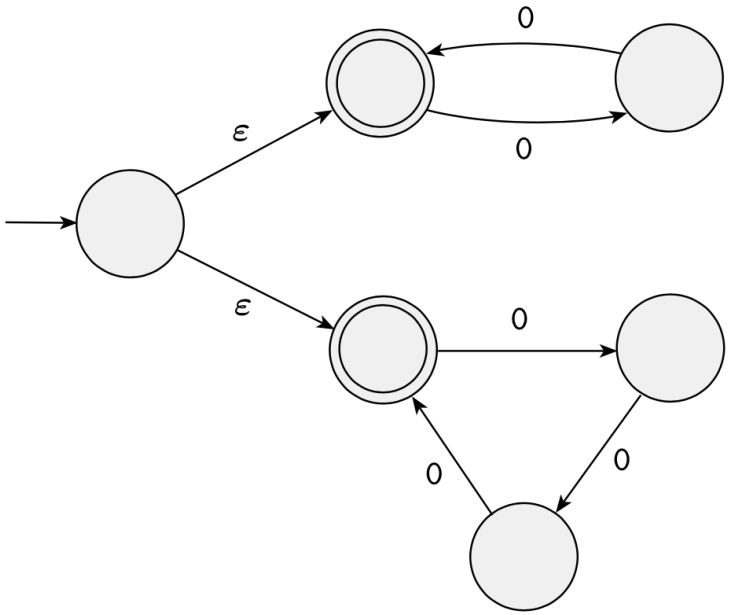


Figure 1.33, Sipser 2012.

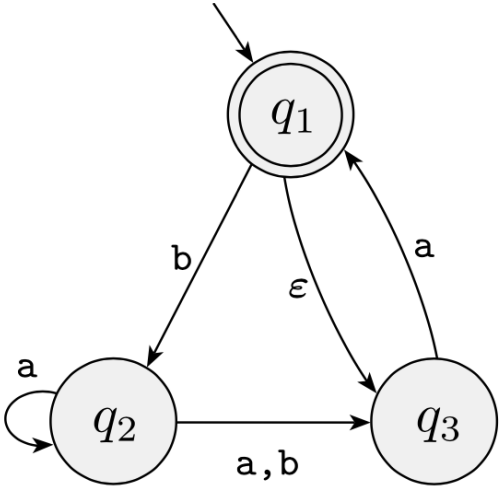
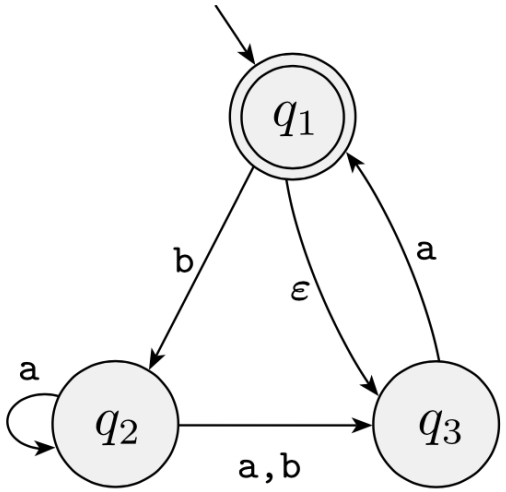


Figure 1.36, Sipser 2012.

Formal definition of NFA

Nondeterministic FA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$

- Q a finite set called the states,
- Σ a finite set called the alphabet,
- δ a function from $Q \times \Sigma_\epsilon$ to 2^Q called the transition function,
- $q_0 \in Q$ the start state,
- $F \subseteq Q$ the set of accept states.

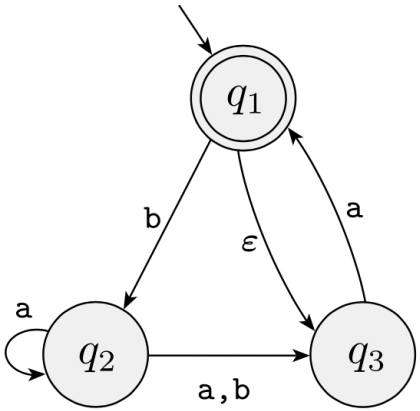


Write a formal description of this NFA

Language recognized by NFA

NFA N accepts w if

1. w can be written as $y_1, \ldots, y_m$ with $y_i \in \Sigma_\epsilon = \Sigma \cup \{\epsilon\}$,
2. there exists a sequence of states $r_0, \ldots, r_m$ s.t.
   - $r_0 = q_0$,
   - r_{i+1} (??) $\delta(q_i, y_i)$,
   - $r_m \in F$.



Write a computation tree for $w = baabaaa$. How many accepting paths?

Closure under regular operation

Union operation

Let A_1 and A_2 be two languages recognized by NFA N_1 and N_2 respectively. Then $A_1 \cup A_2$ is recognized by some NFA.

Closure under regular operation

Concatenation operation

Let A_1 and A_2 be two languages recognized by NFA N_1 and N_2 respectively. Then $A_1 \circ A_2$ is recognized by some NFA.

Closure under regular operation

Kleene star operation

Let A a languages recognized by NFA N. Then A^* is recognized by some NFA.

Closure under complementation

Complementation operation

Let A a languages recognized by NFA N. Then $\bar{A}$, that is, $\Sigma^* - A$ is recognized by some NFA.

- For a regular language L, we can obtain a DFA recognizing the complement of L.
- ...using the trick...
- Can we use the same trick for NFA in general?

Closure under intersection

Intersection operation

Let A_1 and A_2 be two languages recognized by NFA N_1 and N_2 respectively. Then $A_1 \cap A_2$ is recognized by some NFA.

- Use the expression that $A_1 \cap A_2 =$??????.
- Combine the above (which ones?) operations on NFAs...
- Direct way with two DFAs M_1 and M_2 by simulating both automata simultaneously.

Closure under intersection

- Direct way with two DFAs M_1 and M_2 by simulating both automata simultaneously.

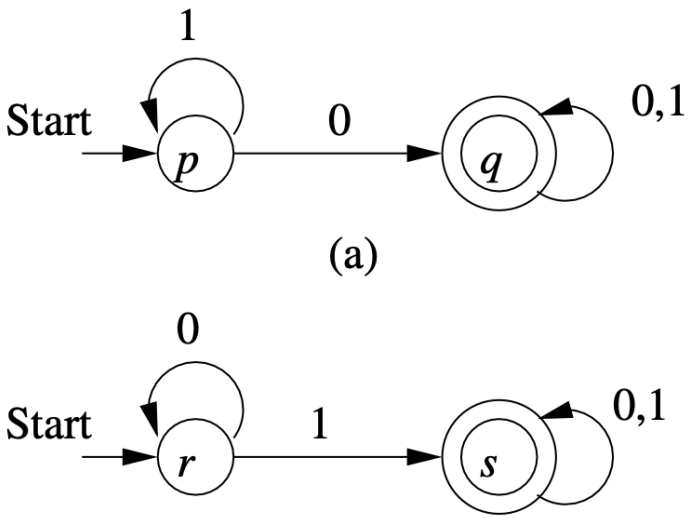


Figure 4.4 (a)-(b), Hopcroft et al. 2014.

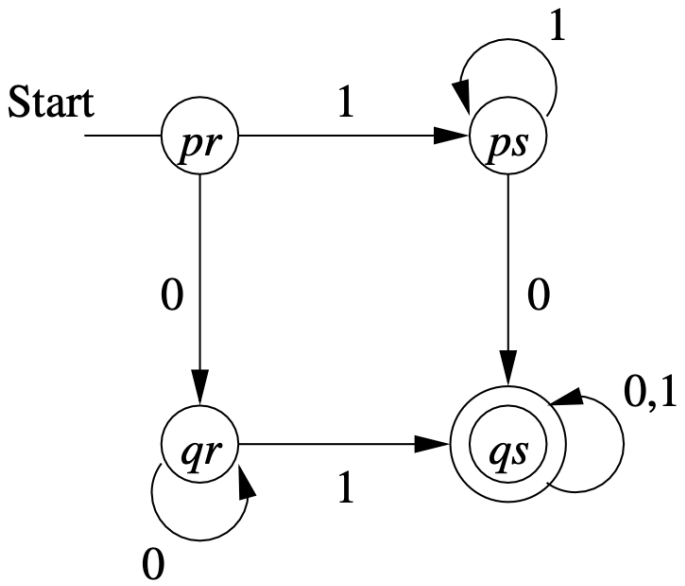


Figure 4.4 (c), Hopcroft et al. 2014.

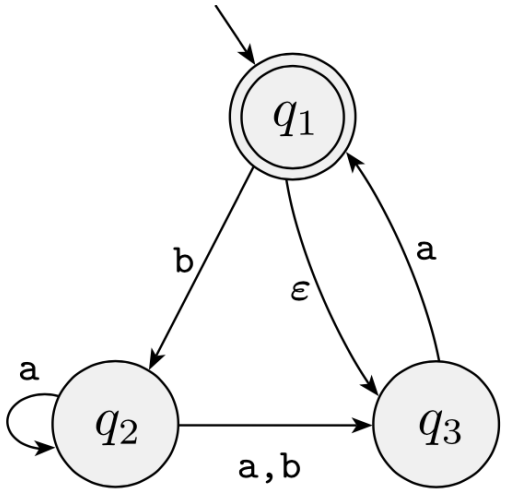
Lec 03. Equivalence of DFA and NFA

Eunjung Kim

Formal definition of NFA

Nondeterministic FA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$

- Q a finite set called the states,
- Σ a finite set called the alphabet,
- δ a function from $Q \times \Sigma_\epsilon$ to 2^Q called the transition function,
- $q_0 \in Q$ the start state,
- $F \subseteq Q$ the set of accept states.

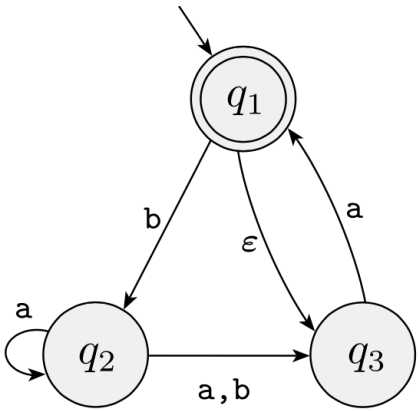


Write a transition table of this NFA.

Language recognized by NFA

NFA N accepts w if

1. w can be written as $y_1, \ldots, y_m$ with $y_i \in \Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ such that there exists a sequence of states $r_0, \ldots, r_m$ satisfying the following:
   - $r_0 = q_0$,
   - $r_{i+1} \quad (??) \quad \delta(q_i, y_i)$,
   - $r_m \in F$.
2. Equivalently, there exists an accepting computation history starting with the (initial) configuration (q_0, w).



Write a computation tree for $w = baabbaa$. How many accepting paths?

Language recognized by NFA

Definition

- Let M be a nondeterministic finite automaton.
- A string $w \in \Sigma^*$ is accepted by M if there exists an accepting computation history.
- $L(M)$ denotes the set of all strings accepted by M.
- A language A is said to be recognized by M if $A = L(M)$.

Equivalence of NFA and DFA

NFA and DFA own the same computational power

For every NFA, there exists a deterministic finite automaton which recognizes the same language (a.k.a. equivalent DFA).

Proof outline.

- Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA.
- We want to construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ such that $L(N) = L(M)$.
- Define $Q' := 2^Q$, i.e. the collection of all subsets of Q.
- Let us define δ', $q_0' \in 2^Q$ and $F' \subseteq 2^Q$,

Proof: constructing DFA M, when no ϵ-transition

- $q_0' = \{q_0\}$.
- transition function δ' from $2^Q \times \Sigma$ to 2^Q:
  for every $R \in 2^Q$ (R is a subset of Q) and every symbol $a \in \Sigma$,

$$\delta'(R, a) := \bigcup_{r \in R} \delta(r, a)$$

- Define $F' \subseteq 2^Q$ as the collection of all subsets of Q containing at least one accept state of N.

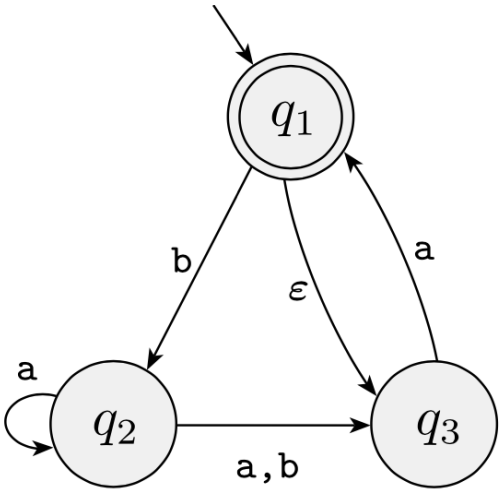
Proof: constructing DFA M with ϵ-transition

- How to define the initial state for DFA: from a state $q \in Q$ of NFA N, any other state q' that can be reached by reading a string ϵ, can be aggregated with q to form a single state in DFA.
- Define $ext(q) \subseteq Q$ as the set of all states q' of N such that there is a directed path from q to q' in (the state diagram of) N each of whose arcs carries the label ϵ. Extend the definition $ext(X) := \bigcup_{q \in X} ext(q)$.
- transition function δ' from $2^Q \times \Sigma_\epsilon$ to 2^Q:
  for every $R \in 2^Q$ (R is a subset of Q) and every symbol $a \in \Sigma$,

$$\delta'(R, a) := ext(\bigcup_{r \in R} \delta(r, a))$$

- Define $q_0' := ext(q_0) \in 2^Q$. Note that q_0' corresponds to a subset of Q.
- Define $F' \subseteq 2^Q$ as the family of all subsets of Q containing at least one accept state of N.

Constructing DFA M from NFA N, example



Proof: $L(N) \subseteq L(M)$

Strategy: from an accepting computation history of N on w, build an accepting computation history of M on w.

Let $w = y_1 y_2 \cdots y_s$ for $y_i \in \Sigma$.

- Let $\pi = (q_0, w = w_0), (r_0, w_0), \ldots, (r_i, w_i), \ldots, (r_s, w_s = \epsilon)$ be an accepting computation history of N for w such that

  r_i is reachable from r_{i-1} via a walk (in the transition diagram of N) labelled by $y_i \circ \epsilon^*$.

- Observe: $r_0 \in ext(\{q_0\})$ and $r_i \in ext(\delta(r_{i-1}, y_i))$ for every $i \in [s]$ and $r_s \in F$.

Proof: $L(N) \subseteq L(M)$

Strategy: from an accepting computation history of N on w, build an accepting computation history of M on w.

Let $w = y_1 y_2 \cdots y_s$ for $y_i \in \Sigma$.

- Let $\pi = (q_0, w = w_0), (r_0, w_0), \ldots, (r_i, w_i), \ldots, (r_s, w_s = \epsilon)$ be an accepting computation history of N for w such that

  r_i is reachable from r_{i-1} via a walk (in the transition diagram of N) labelled by $y_i \circ \epsilon^*$.

- Observe: $r_0 \in ext(\{q_0\})$ and $r_i \in ext(\delta(r_{i-1}, y_i))$ for every $i \in [s]$ and $r_s \in F$.

- Let $Q_0 = q_0'$. Inductively for each $i \in [s-1]$, let

$$Q_i := \delta'(Q_{i-1}, y_i).$$

- Now we have a computation history for $w = y_1 y_2 \cdots y_s$ in DFA M

$$\pi' = (Q_0, w_0 = w), (Q_1, w_1), \cdots (Q_t, w_s = \epsilon).$$

Proof: $L(N) \subseteq L(M)$

Strategy: from an accepting computation history of N on w, build an accepting computation history of M on w.

Let $w = y_1 y_2 \cdots y_s$ for $y_i \in \Sigma$.

- Let $\pi = (q_0, w = w_0), (r_0, w_0), \ldots, (r_i, w_i), \ldots, (r_s, w_s = \epsilon)$ be an accepting computation history of N for w such that

  r_i is reachable from r_{i-1} via a walk (in the transition diagram of N) labelled by $y_i \circ \epsilon^*$.

- Observe: $r_0 \in ext(\{q_0\})$ and $r_i \in ext(\delta(r_{i-1}, y_i))$ for every $i \in [s]$ and $r_s \in F$.

- Let $Q_0 = q_0'$. Inductively for each $i \in [s-1]$, let

$$Q_i := \delta'(Q_{i-1}, y_i).$$

- Now we have a computation history for $w = y_1 y_2 \cdots y_s$ in DFA M

$$\pi' = (Q_0, w_0 = w), (Q_1, w_1), \cdots (Q_t, w_s = \epsilon).$$

Proof: $L(N) \subseteq L(M)$

- It remains to see Q_s is an accept state of M, i.e. the subset $Q_s \subseteq Q$ contains at least one accept state of N.
- This is because $r_0 \in ext(q_0) = Q_0$ and inductively $r_i \in ext(\delta(r_{i-1}, y_i)) \subseteq ext(\delta(Q_{i-1}, y_i) = \delta'(Q_{i-1})$.

Proof: $L(M) \subseteq L(N)$

- Let $\pi' = (R_0, w = w_0), \ldots, (R_i, w_i), \ldots, (R_s, w_s = \epsilon)$ be an accepting computation history of M on w. By definition of computation history $R_i = \delta'(R_{i-1}, y_i)$, where $y_i \in \Sigma$ is the leading symbol of w_{i-1}.
- We construct an accepting computation history of N by following the sequence π' backwardly.
- Observe: for each $q \in R_i \subseteq Q$, there exists a state $q' \in R_{i-1}$ such that from q' to q there is a computation history via a string consisting of y_i followed by ϵ's.
- Now starting from $q_f \in R_s \subseteq Q$, we concatenate computation histories witnessed by the previous observation.

Closure under regular operation

Union operation

Let A_1 and A_2 be two languages recognized by NFA N_1 and N_2 respectively. Then $A_1 \cup A_2$ is recognized by some NFA.

Closure under regular operation

Concatenation operation

Let A_1 and A_2 be two languages recognized by NFA N_1 and N_2 respectively. Then $A_1 \circ A_2$ is recognized by some NFA.

Closure under regular operation

Kleene star operation

Let A a languages recognized by NFA N. Then A^* is recognized by some NFA.

Closure under complementation

Complementation operation

Let A a languages recognized by NFA N. Then $\bar{A}$, that is, $\Sigma^* - A$ is recognized by some NFA.

- For a regular language L, we can obtain a DFA recognizing the complement of L.
- ...using the trick...
- Can we use the same trick for NFA in general?

Closure under intersection

Intersection operation

Let A_1 and A_2 be two languages recognized by NFA N_1 and N_2 respectively. Then $A_1 \cap A_2$ is recognized by some NFA.

- Use the expression that $A_1 \cap A_2 =$??????.
- Combine the above (which ones?) operations on NFAs...
- Direct way with two DFAs M_1 and M_2 by simulating both automata simultaneously.

Closure under intersection

- Direct way with two DFAs M_1 and M_2 by simulating both automata simultaneously.

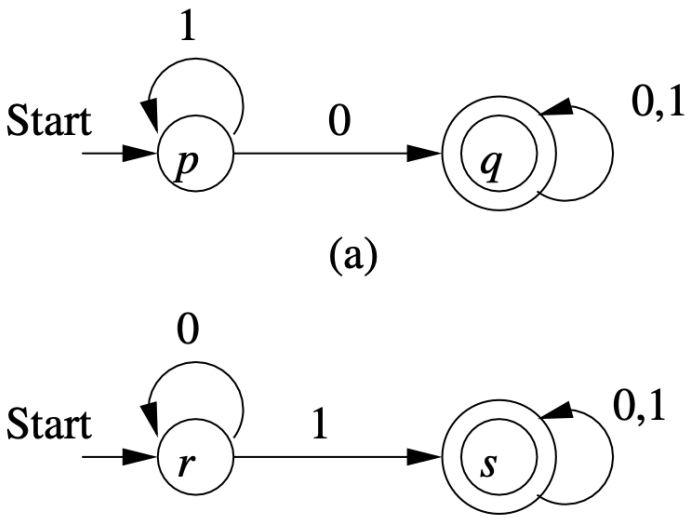


Figure 4.4 (a)-(b), Hopcroft et al. 2014.

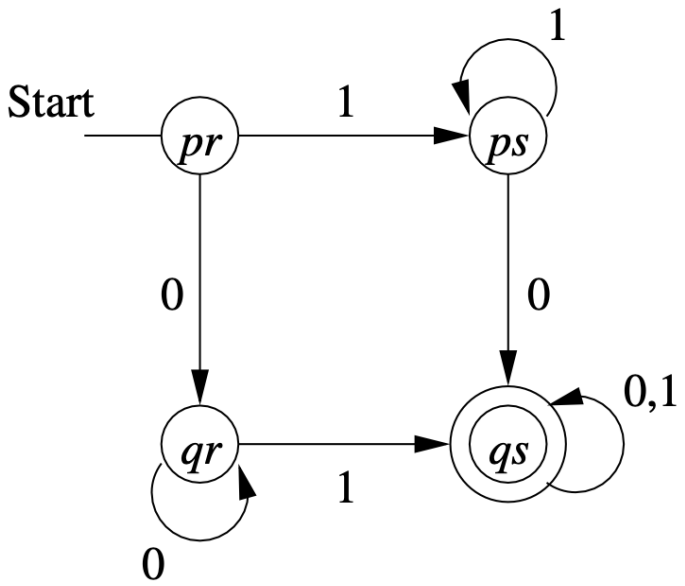


Figure 4.4 (c), Hopcroft et al. 2014.

Formal Languages and Automata, 2025 Fall semester

Lec 04. Regular expression

Eunjung Kim

Regular expression

We want to compactly describe the 'pattern' of the following languages using union, concatenation and Kleene star operations.

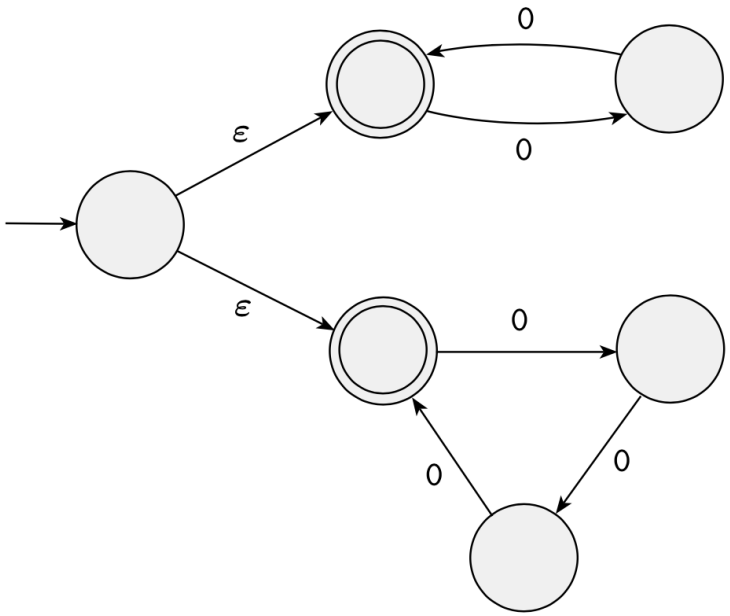


Figure 1.33, Sipser 2012.

1. The set of all $0, 1$ strings with exactly one 1's.
2. The set of all $0, 1$ strings with at least one 1's.
3. The set of strings over Σ of even length.

Formal definition of Regular expression

Regular expression over a finite alphabet Σ

Regular expression over Σ is a string consisting of symbols of Σ, parenthesis ( ), and the operators $\cup, \circ, ^*$ that can be generated as follows.

- Each symbol $x \in \Sigma \cup \{\epsilon\}$ is a regular expression.
- $\emptyset$ is a regular expression.
- $(R_1 \cup R_2)$ is a regular expression if R_1 and R_2 are regular expressions.
- $(R_1 \circ R_2)$ is a regular expression if R_1 and R_2 are regular expressions.
- R^* is a regular expression if R is a regular expression

Examples of Regular expression

Assume $\Sigma = \{0, 1\}$. Which language does the regular expression describe?

1. 0^*10^*.
2. $\Sigma^*1\Sigma^*$.
3. $1^*(01^+)^*$.
4. $(\Sigma\Sigma)^*$.
5. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1$.
6. $1^*\emptyset = \emptyset$.
7. $\emptyset^* = \{\epsilon\}$.

Regular language

Value of a regular expression

For a regular expression R, the set of all strings which can be generated following the expression is denoted by $\mathcal{L}(R)$, said to be the language of R.

$\mathcal{L}(R)$ is also called the value of R, or the language described by R.

For two regular expressions R_1 and R_2, the value of union / concatenation / kleene star is...

- $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$.
- $L(R_1 \circ R_2) = L(R_1) \circ L(R_2)$.
- $L(R_1^\star) = L(R_1)^\star$.

Equivalence of regular expression and finite automata

Regular expression=NFA

A language $A \subseteq \Sigma^*$ is described by a regular expression if and only it is a regular language, i.e. recognized by some NFA.

One direction can be proved easily using what we learnt in the previous lecture. Which direction is it?

Equivalence proof: easy direction

Equivalence Theorem, Part I

If a language $A \subseteq \Sigma^*$ is described by a regular expression, then there is a (nondeterminist) finite automata M such that $L(M) = A$.

Proof idea: inductively build an NFA from NFAs accepting each symbol, $\{\epsilon\}$ or $\emptyset$ by applying each regular operations (union, concatenation, Kleene star).

Equivalence proof: easy direction

Constructing NFA recognizing the language $(\{\epsilon\} \cup \{a\} \cup \{ab\})^\star$.

Constructing NFA recognizing the language $(0 \cup 1)^\star 10$.

Equiv proof: the other direction

Equivalence Theorem, Part II

If a language $A \subseteq \Sigma^*$ is recognized by a finite automata A, then there is a regular expression R such that $\mathcal{L}(R) = A$.

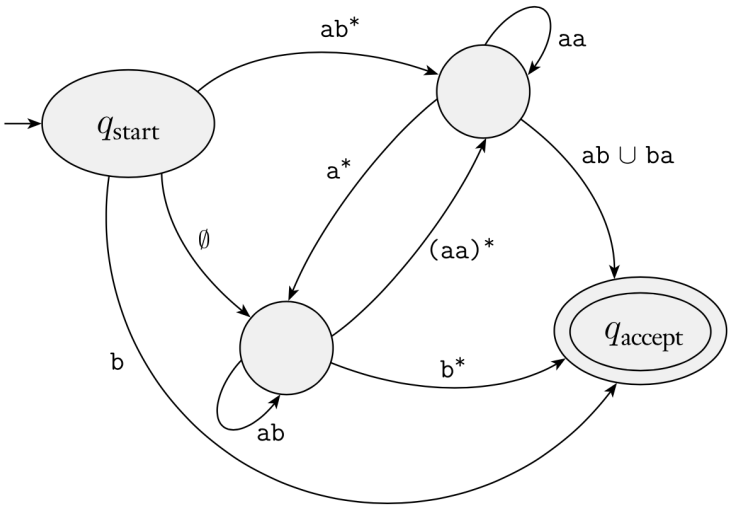
Proof idea: we shall see a procedure which converts DFA (or NFA) recognizing A into a regular expression.

Equiv proof: the other direction

Generalized Nondeterministic Finite Automata

- Generalized NFA is a NFA in which each arc carries a regular expression as a label.
- There are a unique source q_{start} as an initial state and a unique sink q_{accept} as an accept state.
- Between any other states there are arcs in both ways, including loops.

Figure 1.61, Sipser 2012.



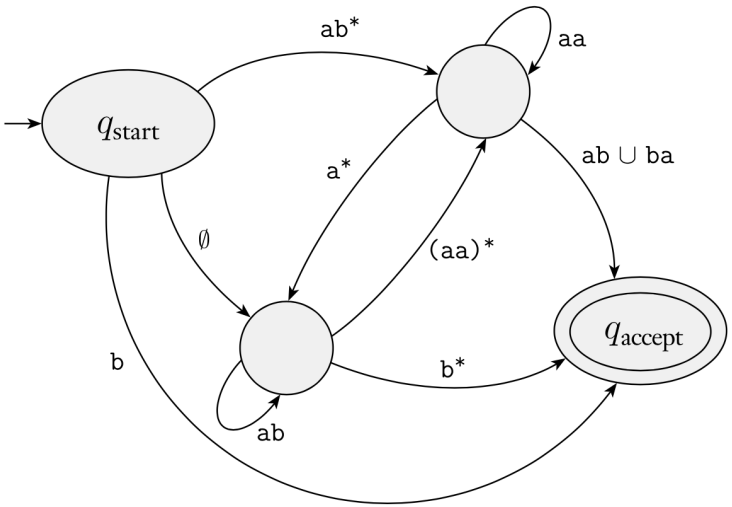
Equiv proof: the other direction

A formal description of GNFA

- Generalized NFA is a 5-tuple $(Q, \Sigma, \delta, q_{start}, q_{accept})$, where
- the transition function maps $(Q \setminus q_{accept}) \times (Q \setminus q_{start})$ to the set of all regular expressions over Σ.

In the transition diagram, we often omit an arc which carries $\emptyset$.

Figure 1.61, Sipser 2012.



Equiv proof: the other direction

A GNFA accepts a string w

If w can be written as $w_1 w_2 \cdots w_\ell$, $w_i \in \Sigma^*$, and there is a sequence of states $r_0, \ldots, r_\ell$ such that

- r_0 is the initial state of GNFA,
- w_i is in the value of $\delta(r_{i-1}, r_i)$, i.e. $w_i \in L(\delta(r_{i-1}, r_i))$, and
- r_ℓ is the accept state of GNFA.

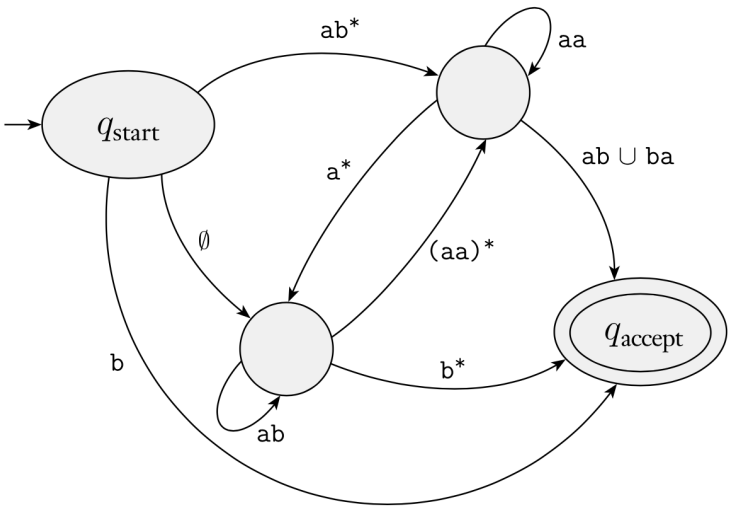
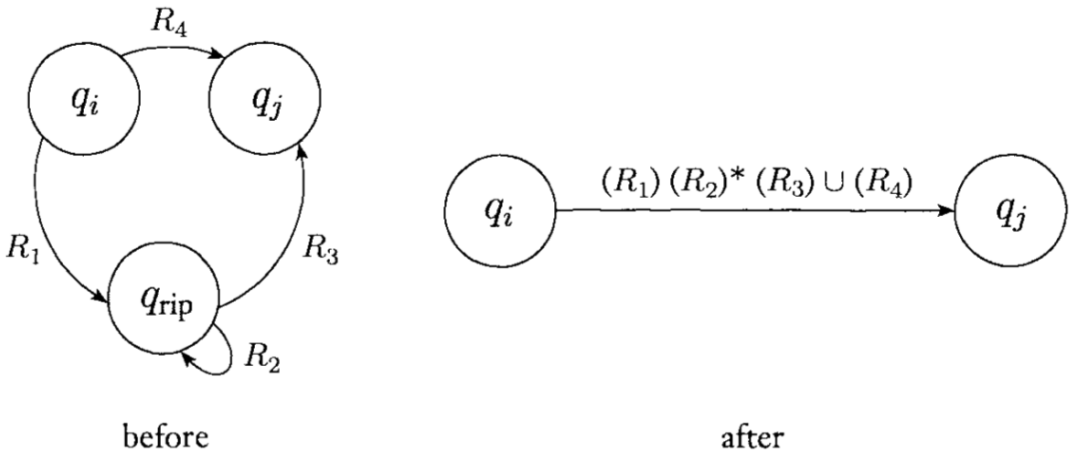


Figure 1.61, Sipser 2012.

Equiv proof: the other direction

Proof idea.

- Initial NFA can be seen as GNFA, possibly after a simple modification to ensure a unique accept state, no incoming/outgoing arc from the unique initial/accept state.
- Reduce the number of states of GNFA inductively by eliminating a state of $Q - \{q_{start}, q_{accet}\}$ one by one, each elimination yielding an equivalent GNFA.
- The final GNFA with two states q_{start} and q_{accept} carries a single regular expression, a desired end product.
- How to eliminate a state q_k and update the label on (q_i, q_j):



Example: from NFA to regular expression

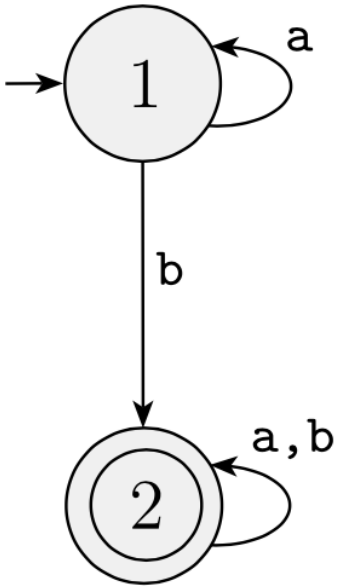


Figure 1.67 (a), Sipser 2012.

Equiv proof: the other direction

Proof by induction. Let G' be a GNFA obtained by eliminating state q_k from GNFA G. It suffices to prove that G accepts a string w if and only if G' does.

- Let $r_0, \ldots, r_\ell$ be a sequence of states appearing in the accepting computation history for $w = w_1 \cdots w_\ell$ in G.
- If q_k does not appear in this sequence, done.

Equiv proof: the other direction

Proof by induction. Let G' be a GNFA obtained by eliminating state q_k from GNFA G. It suffices to prove that G accepts a string w if and only if G' does.

- Let $r_0, \ldots, r_\ell$ be a sequence of states appearing in the accepting computation history for $w = w_1 \cdots w_\ell$ in G.
- If q_k does not appear in this sequence, done.
- If not, consider a maximal subsequence $r_a, \ldots, r_b$ of contiguous occurrences of q_k. Note that $1 \leq a \leq b < \ell$.

Equiv proof: the other direction

Proof by induction. Let G' be a GNFA obtained by eliminating state q_k from GNFA G. It suffices to prove that G accepts a string w if and only if G' does.

- Let $r_0, \ldots, r_\ell$ be a sequence of states appearing in the accepting computation history for $w = w_1 \cdots w_\ell$ in G.
- If q_k does not appear in this sequence, done.
- If not, consider a maximal subsequence $r_a, \ldots, r_b$ of contiguous occurrences of q_k. Note that $1 \leq a \leq b < \ell$.
- Observe: $w_a \in L(\delta(r_{a-1}, r_a))$, $w_{i+1} \in L(\delta(r_i, r_{i+1})) = L(\delta(q_k, q_k))$ for every $i \in [a, b-1]$, and $w_{b+1} \in L(\delta(r_b, r_{b+1}))$.

Equiv proof: the other direction

Proof by induction. Let G' be a GNFA obtained by eliminating state q_k from GNFA G. It suffices to prove that G accepts a string w if and only if G' does.

- Let $r_0, \ldots, r_\ell$ be a sequence of states appearing in the accepting computation history for $w = w_1 \cdots w_\ell$ in G.
- If q_k does not appear in this sequence, done.
- If not, consider a maximal subsequence $r_a, \ldots, r_b$ of contiguous occurrences of q_k. Note that $1 \leq a \leq b < \ell$.
- Observe: $w_a \in L(\delta(r_{a-1}, r_a))$, $w_{i+1} \in L(\delta(r_i, r_{i+1})) = L(\delta(q_k, q_k))$ for every $i \in [a, b-1]$, and $w_{b+1} \in L(\delta(r_b, r_{b+1}))$.
- That is: $w_a \cdots w_{b+1}$ is described by

$$\delta(r_{a-1}, r_a) \cdot \delta(q_k, q_k)^\star \cdot \delta(q_k, r_{b+1})$$

  which is in the union expression $\delta'(r_{a-1}, r_{b+1})$ in the G'.

Equiv proof: the other direction

Conversely,

- Let $r_0, \ldots, r_\ell$ be a sequence of states appearing in the accepting computation history for $w = w_1 \cdots w_\ell$ in G'.
- Each w_i in is the language of $\delta'(r_{i-1}, r_i)$.
- $\delta'(r_{i-1}, r_i) = \delta(r_{i-1}, r_i) \cup \delta(r_{i-1}, q_k) \cdot \delta(q_k, q_k)^\star \cdot \delta(q_k, r_i)$.
- Therefore, $w_i \in \delta(r_{i-1}, r_i)$ or
  w_i can be written as $x_1 \cdots, x_m$ such that
  1. $x_1 \in \delta(r_{i-1}, q_k)$
  2. $x_2 \in \delta(q_k, r_i)$

  or
  1. $x_1 \in \delta(r_{i-1}, q_k)$
  2. $x_j \in \delta(q_k, q_k)$ for each $2 \leq j < m$
  3. $x_m \in \delta(q_k, r_i)$
- Now replace the computation history in G' by..... to obtain a computation history in G.

Example: from NFA to regular expression

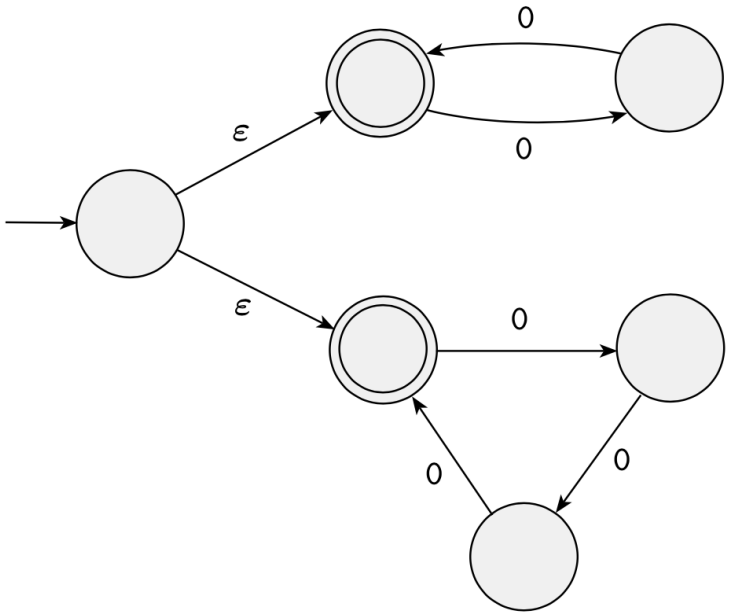


Figure 1.33, Sipser 2012.

Lec 05. Pumping Lemma & Minimal DFA

Eunjung Kim

Limit of finite automata and tools for investigation

Which of the following languages are regular?

1. $B = \{0^n 1^n : n \geq 0\}$.
2. $C = \{w : w \text{ has equal number of 0's and 1's}\}$.
3. $D = \{w : w \text{ has equal number of 01's and 10's}\}$.
4. For a DFA D, the set of strings in $L(D)$ accepted via a computation history visiting all states.

Pumping lemma

Pumping lemma: tool to prove nonregularity

Let A be a regular language. Then there exists a number p (called the pumping length) such that any string $w \in A$ of length at least p, w can be written as $w = xyz$ such that the following holds:

1. $|y| \geq 1$,
2. $|xy| \leq p$,
3. $xy^i z \in A$ for every $i \geq 0$.

Proof idea: DFA for A has a finite (constant) number of states.

Pumping lemma, proof

There exists DFA M with $L(M) = A$.

1. Let p be the number of states of this DFA.
2. Consider the accepting computation history $r_0 = q_0, r_1, \ldots, r_s$ for w (with $r_s \in F$) such that $r_{i+1} = \delta(r_i, w_{i+1})$ for all $i = 0, \ldots, s-1$, where w_i is the i-th symbol of w.
3. In the first $p+1$ states $r_0, \ldots, r_p$, there exist two identical states, say r_a and r_b, with $a \neq b$.
4. Take $x = w_1 \cdots w_a$, $y = w_{a+1} \cdots w_b$ and $z = w_{b+1} \cdots w_s$.
5. It remains to observe that
   - $r_{b+1} = \delta(r_b, w_{b+1}) = \delta(r_a, w_{b+1})$, and thus $w_1 \cdots w_a \cdot w_{b+1} \cdots w_s = x \cdot z = x \cdot y^0 \cdot z$ is accepted with the sequence of states $r_0, \ldots, r_a, r_{b+1}, \ldots, r_s$.
   - Any $x \cdot y^i \cdot z$ is accepted with the sequence

$$r_0, \ldots, r_a, (r_{a+1}, \ldots, r_b)^i, r_{b+1}, \ldots, r_s.$$

Pumping lemma for nonregularity

Pumping lemma

Let A be a regular language. Then there exists a number p such that any string $w \in A$ of length at least p, w can be written as $w = xyz$ such that ....

Recipe: assume that A is regular and p is an unknown (arbitrary) pumping length. Choose a good string s, and show that rewriting $s = xyz$ as required is impossible.

Pumping lemma for nonregularity

Pumping lemma

Let A be a regular language. Then there exists a number p such that any string $w \in A$ of length at least p, w can be written as $w = xyz$ such that ....

Recipe: assume that A is regular and p is an unknown (arbitrary) pumping length. Choose a good string s, and show that rewriting $s = xyz$ as required is impossible.
That is, we use the contraposition of Pumping lemma for proving nonregularity of A

Syntax for showing non-regularity

1. For every positive number p, ("$\forall p$")
2. there exists $w \in A$ of length at least p such that ("$\exists w \in A$")
3. for every split $w = xyz$ with $|y| \geq 1$ and $|xy| \leq p$ ("$\forall$ splits xyz")
4. there exists $i \geq 0$ with $xy^i z \notin A$ ("$\exists i$").

Nonregularity of $B = \{0^n 1^n : n \geq 0\}$.

Syntax

1. For every positive number p, ("$\forall p$")
2. there exists $w \in A$ of length at least p such that ("$\exists w \in A$")
3. for every split $w = xyz$ with $|y| \geq 1$ and $|xy| \leq p$ ("$\forall$ splits xyz")
4. there exists $i \geq 0$ with $xy^i z \notin A$ ("$\exists i$").

$\{w : w$ HAS EQUAL # OF 0'S AND 1'S$\}$

SYNTAX

1. For every positive number p, ("$\forall p$")
2. there exists $w \in A$ of length at least p such that ("$\exists w \in A$")
3. for every split $w = xyz$ with $|y| \geq 1$ and $|xy| \leq p$ ("$\forall$ splits xyz meeting the conditions")
4. there exists $i \geq 0$ with $xy^i z \notin A$ ("$\exists i$").

Alternative way to show the non-regularity?

$$D = \{1^{n^2} : n \geq 0\}$$

Syntax

1. For every positive number p, ("$\forall p$")
2. there exists $w \in A$ of length at least p such that ("$\exists w \in A$")
3. for every split $w = xyz$ with $|y| \geq 1$ and $|xy| \leq p$ ("$\forall$ splits xyz")
4. there exists $i \geq 0$ with $xy^i z \notin A$ ("$\exists i$").

$$D = \{0^i \cdot 1^j : i > j\}$$

Syntax

1. For every positive number p, ("$\forall p$")
2. there exists $w \in A$ of length at least p such that ("$\exists w \in A$")
3. for every split $w = xyz$ with $|y| \geq 1$ and $|xy| \leq p$ ("$\forall$ splits xyz")
4. there exists $i \geq 0$ with $xy^i z \notin A$ ("$\exists i$").

$$F = \{ww : w \in \{0,1\}^*\}$$

Syntax

1. For every positive number p, ("$\forall p$")
2. there exists $w \in A$ of length at least p such that ("$\exists w \in A$")
3. for every split $w = xyz$ with $|y| \geq 1$ and $|xy| \leq p$ ("$\forall$ splits xyz")
4. there exists $i \geq 0$ with $xy^i z \notin A$ ("$\exists i$").

Reducing the number of states of DFA

A DFA recognizes a single (unique) language. But there are more than one (in fact, arbitrarily many) DFAs which recognizes the same language.

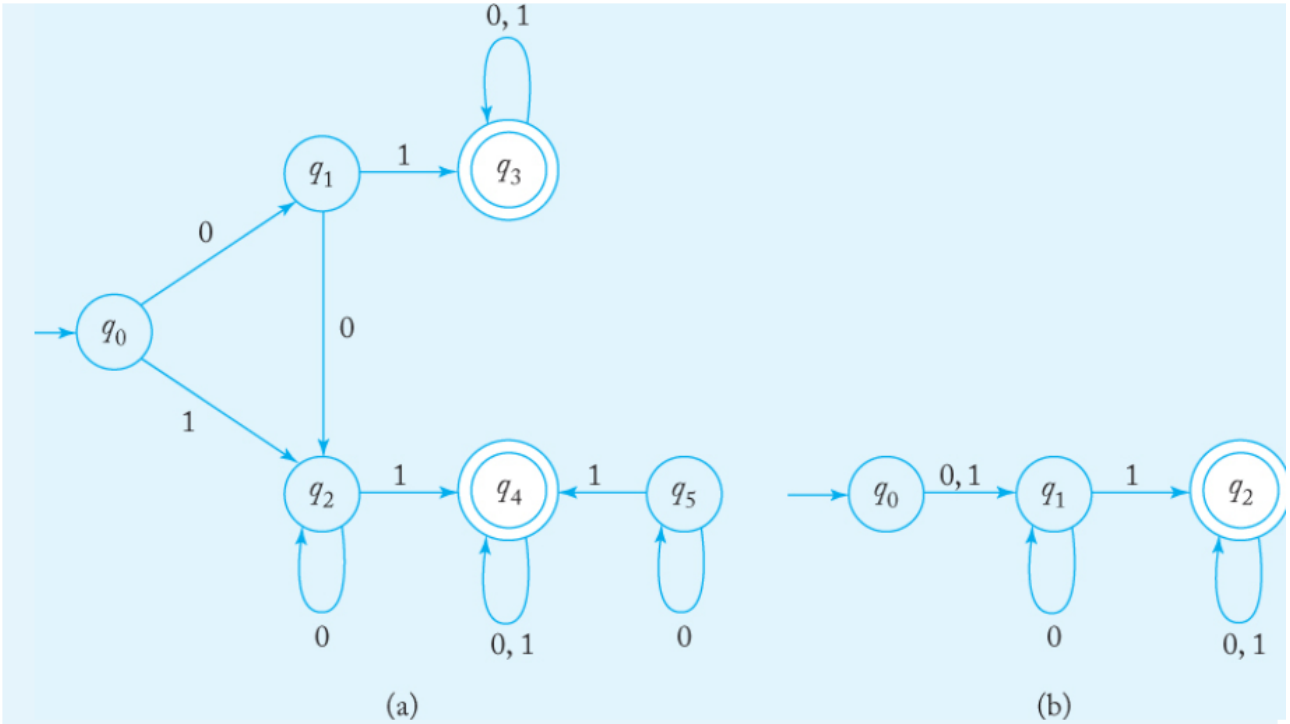


FIGURE 2.17

Figure 2.17, Peter Linz 2014.

Reducing the number of states of DFA

Indistinguishable states

Given DFA M, two states $p, q \in Q$ are indistinguishable if for every string $w \in \Sigma^*$,

$$\hat{\delta}(p, w) \in F \text{ if and only if } \hat{\delta}(q, w) \in F.$$

Remark: indistinguishability is an equivalence relation on Q.

Reducing the number of states of DFA

Indistinguishable states

Given DFA M, two states $p, q \in Q$ are indistinguishable if for every string $w \in \Sigma^*$,

$$\hat{\delta}(p, w) \in F \text{ if and only if } \hat{\delta}(q, w) \in F.$$

Remark: indistinguishability is an equivalence relation on Q.

Distinguishing string

We say that a string $w \in \Sigma^*$ distinguishes two states p, q if

$$\hat{\delta}(p, w) \in F \text{ and } \hat{\delta}(q, w) \notin F \text{ or vice versa.}$$

Reducing the number of states of DFA

Procedure for reducing # states of given DFA.

1. Remove all inaccessible (i.e. not accessible from q_0) states.
2. Any pair $(p, q) \in F \times Q \setminus F$ is marked as distinguishable.
3. Mark a pair p, q as distinguishable if there exists $a \in \Sigma$ such that the pair $\delta(p, a), \delta(q, a)$ is already marked as distinguishable.
4. Repeat above until there is no more pair to be marked distinguishable.
5. Group all states which are not marked as indistinguishable; the groups ($\sim$) form a partition of Q.
6. $M/\sim$ is well-defined; this is our reduced automaton.

Reducing the number of states of DFA

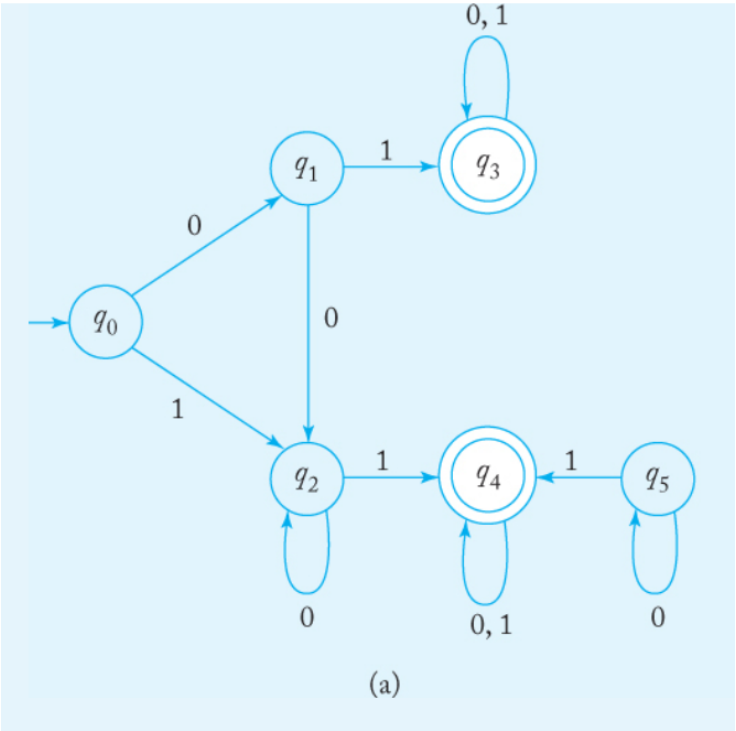


Figure 2.17 (a), Peter Linz 2014.

Reducing the number of states of DFA

- Why does this procedure works? (i.e. produces an equivalent automaton)
- Given a DFA M, the procedure leads to a unique outcome?
- Is this a DFA with the minimum possible number of states?
- Does the procedure leads to the same (minimum) DFA regardless of the starting DFAs?

Why does this procedure works?

We observe

- Any pair marked as distinguishable are indeed distinguishable.
  $\rightsquigarrow$ By induction, we argue that any marked pair has a distinguishing string.

- Any pair unmarked at the end of procedure are indistinguishable.
  $\rightsquigarrow$ Suppose not, and unmarked pair p, q is distinguished by a string w of length n. Consider the sequence of states in the computation histories of (p, w) and (q, w)...

Why does this procedure works?

Now the "groups" in Q are indeed the equivalence classes of $\sim$.

- Let $Q_1, \ldots, Q_\ell$ be the equivalence classes.
- Key fact: For $p, p' \in Q_i$ (i.e. $p \sim p'$), $\delta(p, a) \sim \delta(p', a)$ for every $a \in \Sigma$.
- So the "quotient $M/\sim$ of M is well-defined; this is our new DFA.
- Uniqueness of the procedure's outcome from a given DFA follows.
- Check yourself that $L(M) = L(M/\sim)$.

Another example

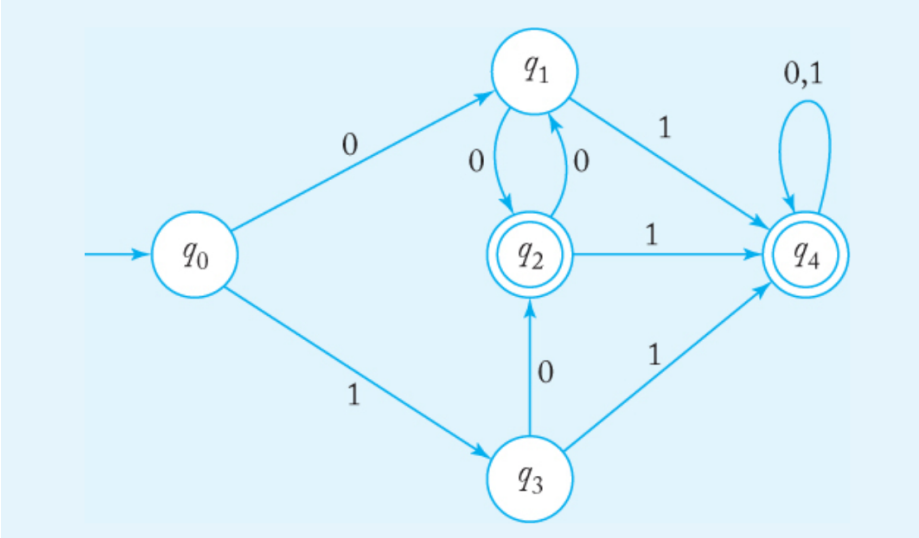


Figure 2.18, Peter Linz 2014.

Is this a DFA with the minimum # states?

Does the procedure leads to the same (minimum) DFA regardless of the starting DFAs?

- Here, we are asking if there is a unique minimum DFA (up to renaming the states).
- Answer via so-called Myhill-Nerode Theorem.
- This can also be used as an alternative approach for establishing non-regularity of a language.

Lec 06. Minimum DFA, Myhill-Nerode and MSO logic

Eunjung Kim

Reducing the number of states of DFA

- Why does this procedure works? (i.e. produces an equivalent automaton)
- Given a DFA M, the procedure leads to a unique outcome?
- Is this a DFA with the minimum possible number of states?
- Does the procedure leads to the same (minimum) DFA regardless of the starting DFAs?

Why does this procedure works?

We observe

- Any pair marked as distinguishable are indeed distinguishable.
  $\rightsquigarrow$ By induction, we argue that any marked pair has a distinguishing string.

- Any pair unmarked at the end of procedure are indistinguishable.
  $\rightsquigarrow$ Suppose not, and unmarked pair p, q of states is distinguished by a string w of length n. Consider the sequence of states in the computation histories of (p, w) and (q, w)...

Why does this procedure works?

Now the "groups" in Q are indeed the equivalence classes of $\sim$.

- Let $Q_1, \ldots, Q_\ell$ be the equivalence classes.
- Key fact: For $p, p' \in Q_i$ (i.e. $p \sim p'$), $\delta(p, a) \sim \delta(p', a)$ for every $a \in \Sigma$.
- So the "quotient $M/\sim$ of M is well-defined; this is our new DFA.

$$\delta'([p], a) := [\delta(p, a)]$$

  well-defined; $\delta'([p], a) = [\delta(p, a)] = [\delta(q, a)] = \delta'([q], a)$
- Uniqueness of the procedure's outcome from a given DFA follows.
- Check yourself that $L(M) = L(M/\sim)$.

Is this a DFA with the minimum # states?

New states of $M/\sim$ are distinguishable

- Choose two inequivalent states of M, i.e. $q_1 \not\sim q_2$, and let w be a string distinguishing q, q'.
- For any $q_1' \sim q_1$, w also distinguishes q_1' and q_2. (Why?)

$\rightsquigarrow$ every pair of new states in $M/\sim$ are distinguishable.

Is this a DFA with the minimum # states?

Let $p_0, p_1, \ldots, p_\ell$ be the states of $M' = (Q', \Sigma, \delta', p_0, F')$ (our new DFA obtained from M).

Suppose there is another DFA D with $q < \ell$ states.

- Choose ℓ strings $s_1, \ldots, s_\ell \in \Sigma^*$ such that $\hat{\delta}'(p_0, s_i) = p_i$ for each $i \in [\ell]$.
- Such strings exist because every state of M' is accessible from p_0.
- Run D on these ℓ strings; there exist two strings s_i, s_j s.t. D ends up in the same state upon s_i and s_j.
- Note that there is a string distinguishing p_i and p_j for any pair $0 \leq i < j \leq \ell$ by the previous observation.
- What are the states you reach when you run D on $s_i \circ w$ and $s_j \circ w$?

Does the procedure leads to the same (minimum) DFA regardless of the starting DFAs?

- Here, we are asking if there is a unique minimum DFA (up to renaming the states).
- Answer via so-called Myhill-Nerode Theorem.
- Myhill-Nerode Theorem can also be used as an alternative approach for establishing non-regularity of a language.

Myhill-Nerode Theorem

Fix an alphabet Σ and let L be a language over Σ.

Indistiguishability of two strings by L

We say that two strings $x, y \in \Sigma^*$ is indistinguishable by L if for all $z \in \Sigma^*$,

$$x \cdot z \in L \text{ if and only if } y \cdot z \in L,$$

written as $x \equiv_L y$.

Distiguishability of two strings by L

We say that $z \in \Sigma^*$ is a distinguishing extension of two strings $x, y \in \Sigma^*$ for L if

$$x \circ z \in L \text{ and } y \circ z \notin L, \text{ or vice versa.}$$

Note that $x \not\equiv_L y$ if and only if there is a distinguishing extension of them.

Myhill-Nerode Theorem

Myhill-Nerode Theorem

L is regular if and only if the number of equivalence classes of $\equiv_L$ is finite.

($\leftarrow$) Build a DFA $D = (Q, \Sigma, \delta, q_0, F)$ from the equivalence classes of $\equiv_L$.
Use the fact that $x \equiv_L y$ implies $x \circ a \equiv_L y \circ a$ for every $a \in \Sigma$ (why?).

- $Q =$ the set of the equivalence classes of $\equiv_L$ (often written as $\Sigma^* / \equiv_L$).
- $q_0 =$???. $[\varepsilon]$
- $\delta([x], a) =$??? for each $a \in \Sigma$. $[xa]$
- $F \subseteq Q$: $[x] \in F$ for every $x \in L$.

Myhill-Nerode Theorem

Myhill-Nerode Theorem

L is regular if and only if the number of equivalence classes of $\equiv_L$ is finite.

Moreover, the number of equivalence classes equals the number of states in a minimal (minimum) DFA.

($\rightarrow$, also the second part) Consider any DFA M with $L(M) = L$. Note that if $\hat{\delta}(q_0, x) \sim \hat{\delta}(q_0, y)$ for two strings $x, y \in \Sigma^*$, then $x \equiv_L y$.

Myhill-Nerode Theorem for non-regularity

Myhill-Nerode Theorem, in contraposition

L is non-regular if and only if there is an infinite set $S \subseteq \Sigma^*$ consisting of pairwise distinguishable strings.

Myhill-Nerode Theorem for non-regularity

Myhill-Nerode Theorem, in contraposition

L is non-regular if and only if there is an infinite set $S \subseteq \Sigma^*$ consisting of pairwise distinguishable strings.

- Mind that we seek for distinguishable strings, which are not necessarily in L.
- For pairwise distinguishable strings $S = \{s_1, \ldots, s_m, \cdots\}$, a distinguishing extension for (s_i, s_j) might be in general different from a distinguishing extension for (s_j, s_k).

Myhill-Nerode Theorem for non-regularity, example

- $L_1 = \{0^n 1^n \mid n \geq 1\}$
- $L_2 = \{w \in \{0,1\}^* \mid w \text{ is a palindrome}\}$

Strategy: find an infinite subset of Σ^* which consists of pairwise distinguishable (inequivalent) strings.

MSO LOGIC ON STRINGS

We saw several, all equivalent, characterization of regular language.

- DFA / NFA (algorithm)
- Regular expression (composability via basic operations)
- Recognizability by monoid (algebraic property)
- Myhill-Nerode Theorem
- Generated by left/right linear grammar (not covered, yet)
- Definability by Monadic Second Order logic

MSO LOGIC ON STRINGS, BY EXAMPLE

We want to express the language

$$L = \{w \in \{0,1\}^* \mid w \text{ does not contain 11 as a substring}\}$$

with an MSO-sentence.

MSO-SENTENCE

$$\varphi = \forall x \forall y \, (x < y) \rightarrow \big(\exists z \, (x < z < y) \vee P_0(x) \vee P_0(y)\big)$$

Here, $P_0(x)$ is read as "the x-th symbol in the string is 0".

Likewise, $P_1(y)$ is read as "the y-th symbol in in the string is 1".

10010 satisfies φ whereas 1101 not, which we denote as $10010 \models \varphi$ and $1101 \not\models \varphi$.

MSO LOGIC ON STRINGS, BY EXAMPLE

We want to express that

a set S of positions in the given string forms an "interval".

MSO-FORMULA

$$\varphi_{int}(S) = \forall x\, \forall y\, (x \in S \wedge y \in S \wedge x \leq y) \rightarrow \big(\forall z\, (x \leq z \leq y) \rightarrow z \in S\big)$$

Note that the validity of $\varphi_{int}(S)$ depends not only on the given string, but also the variable S.

MSO LOGIC ON STRINGS

We first express a string $s \in \Sigma^*$ as a logical structure (often called "relational structure").

String w as a logical structure

Universe $= [n]$, where n is the length of the string.

- That is, each "position" (from 1 to n) in the string is an element in the universe. If $w = \epsilon$, the universe is $\emptyset$.

A binary relation $<$ and $|\Sigma|$ unary relations P_a for all $a \in \Sigma$ on the universe.

- $x < y$: "the x-th position precedes the y-th position in the string."
- $P_0(x)$ is true if "the x-th symbol is 0."

$\tau = \{<\} \cup \{P_a \mid a \in \Sigma\}$ is called the vocabulary on Σ-strings.

MSO LOGIC ON STRINGS

MSO-FORMULA ON Σ-STRINGS

An MSO-formula on strings is a well-formed string that can be constructed using from atomic formulas for (infinite supply of) individual variables $x, y, z \ldots$, and set variables $X, Y, Z \cdots$ i.e.

- $x < y$; note that $< \in \tau$,
- $P_a(x)$ for each $a \in \Sigma$,
- $x = y$, and $x \in X$.

by applying

- the logical connectives $\wedge, \vee, \neg, \rightarrow$; $\varphi_1 \wedge \varphi_2$, $\neg\varphi$, etc,
- the universal and existential quantifier $\forall, \exists$; in the form $\exists x\varphi$, $\exists X\varphi$, etc.

An MSO-formula in which all variables are quantified (by $\forall$ or $\exists$) is called an MSO-sentence.

MSO LOGIC ON STRINGS

A property = the set of all Σ-strings which has the property.

A PROPERTY ON STRINGS AS AN MSO-SENTENCE

We say that a property $L \subseteq \Sigma^*$ on strings (a.k.a. a language) is expressible, or equivalently definable, in MSO if there is an MSO-sentence φ on Σ-strings such that

$$w \in L \text{ if and only if } w \models \varphi$$

for every string $w \in \Sigma^*$.

MSO LOGIC ON STRINGS, BY EXAMPLE

Let us express the property L on $\{0,1\}$-strings having even number of 1's, i.e.

$$L = \{w \in \{0,1\}^* \mid \text{there are even number of 1's in } w\}.$$

Use the fact that $w \in L$ if and only if

- either $w = \epsilon$,
- or the positions of 1's in w can be "uniquely colored" in RED or BLUE so that two colors alternate.

MSO LOGIC ON STRINGS, BY EXAMPLE

MSO-FORMULA DEFINING L

- $\varphi_\epsilon = \neg\exists x\ (x = x)$
- $\varphi_{color}(R, B) = \forall x\ (P_1(x) \to (x \in R \vee x \in B)) \wedge (P_0(x) \to \neg(x \in R \vee x \in B))$
- $\varphi_{unique}(R, B) = \forall x\ (x \in R \to \neg x \in B) \wedge (x \in B \to \neg x \in R)$
- $\varphi_{alternate}(R, B) =$??????

$\forall x \forall y ((x<y) \wedge (x \in R \wedge y \in R \to \exists z (x<z<y) \wedge z \in B))$

Finally, we get a sentence φ_L defining L as

$$\varphi_L = \varphi_\epsilon \vee \exists R\ \exists B\ \varphi_{color}(R, B) \wedge \varphi_{unique}(R, B) \wedge \varphi_{alternate}(R, B)$$

Büchi's Theorem 1960

Recognizability equals definability on strings

A language is regular if and only if it is definable in Mso.

Lec 06. Minimum DFA, Myhill-Nerode and MSO logic

Eunjung Kim

Reducing the number of states of DFA

- Why does this procedure works? (i.e. produces an equivalent automaton)
- Given a DFA M, the procedure leads to a unique outcome?
- Is this a DFA with the minimum possible number of states?
- Does the procedure leads to the same (minimum) DFA regardless of the starting DFAs?

Why does this procedure works?

We observe

- Any pair marked as distinguishable are indeed distinguishable.
  $\rightsquigarrow$ By induction, we argue that any marked pair has a distinguishing string.

- Any pair unmarked at the end of procedure are indistinguishable.
  $\rightsquigarrow$ Suppose not, and unmarked pair p, q of states is distinguished by a string w of length n. Consider the sequence of states in the computation histories of (p, w) and (q, w)...

Why does this procedure works?

Now the "groups" in Q are indeed the equivalence classes of $\sim$.

- Let $Q_1, \ldots, Q_\ell$ be the equivalence classes.
- Key fact: For $p, p' \in Q_i$ (i.e. $p \sim p'$), $\delta(p, a) \sim \delta(p', a)$ for every $a \in \Sigma$.
- So the "quotient $M/\sim$ of M is well-defined; this is our new DFA.

$$\delta'([p], a) := [\delta(p, a)]$$

  well-defined; $\delta'([p], a) = [\delta(p, a)] = [\delta(p', a)] = \delta'([p'], a)$
- Uniqueness of the procedure's outcome from a given DFA follows.
- Check yourself that $L(M) = L(M/\sim)$.

Is this a DFA with the minimum # states?

New states of $M/\sim$ are distinguishable

- Choose two inequivalent states of M, i.e. $q_1 \not\sim q_2$, and let w be a string distinguishing q, q'.
- For any $q_1' \sim q_1$, w also distinguishes q_1' and q_2. (Why?)

$\rightsquigarrow$ every pair of new states in $M/\sim$ are distinguishable.

Is this a DFA with the minimum # states?

Let $p_0, p_1, \ldots, p_\ell$ be the states of $M' = (Q', \Sigma, \delta', p_0, F')$ (our new DFA obtained from M).

Suppose there is another DFA D with $q < \ell$ states.

- Choose ℓ strings $s_1, \ldots, s_\ell \in \Sigma^*$ such that $\hat{\delta}'(p_0, s_i) = p_i$ for each $i \in [\ell]$.
- Such strings exist because every state of M' is accessible from p_0.
- Run D on these ℓ strings; there exist two strings s_i, s_j s.t. D ends up in the same state upon s_i and s_j.
- Note that there is a string distinguishing p_i and p_j for any pair $0 \leq i < j \leq \ell$ by the previous observation.
- What are the states you reach when you run D on $s_i \circ w$ and $s_j \circ w$?

Does the procedure leads to the same (minimum) DFA regardless of the starting DFAs?

- Here, we are asking if there is a unique minimum DFA (up to renaming the states).
- Answer via so-called Myhill-Nerode Theorem.
- Myhill-Nerode Theorem can also be used as an alternative approach for establishing non-regularity of a language.

Myhill-Nerode Theorem

Fix an alphabet Σ and let L be a language over Σ.

Indistiguishability of two strings by L

We say that two strings $x, y \in \Sigma^*$ is indistinguishable by L if for all $z \in \Sigma^*$,

$$x \cdot z \in L \text{ if and only if } y \cdot z \in L,$$

written as $x \equiv_L y$.

Distiguishability of two strings by L

We say that $z \in \Sigma^*$ is a distinguishing extension of two strings $x, y \in \Sigma^*$ for L if

$$x \circ z \in L \text{ and } y \circ z \notin L, \text{ or vice versa.}$$

Note that $x \not\equiv_L y$ if and only if there is a distinguishing extension of them.

Myhill-Nerode Theorem

Myhill-Nerode Theorem

L is regular if and only if the number of equivalence classes of $\equiv_L$ is finite.

($\leftarrow$) Build a DFA $D = (Q, \Sigma, \delta, q_0, F)$ from the equivalence classes of $\equiv_L$.
Use the fact that $x \equiv_L y$ implies $x \circ a \equiv_L y \circ a$ for every $a \in \Sigma$ (why?).

- $Q =$ the set of the equivalence classes of $\equiv_L$ (often written as $\Sigma^* / \equiv_L$).
- $q_0 =$???.
- $\delta([x], a) =$??? for each $a \in \Sigma$.
- $F \subseteq Q$: $[x] \in F$ for every $x \in L$.

Myhill-Nerode Theorem

Myhill-Nerode Theorem

L is regular if and only if the number of equivalence classes of $\equiv_L$ is finite.

Moreover, the number of equivalence classes equals the number of states in a minimal (minimum) DFA.

($\rightarrow$, also the second part) Consider any DFA M with $L(M) = L$. Note that if $\hat{\delta}(q_0, x) \sim \hat{\delta}(q_0, y)$ for two strings $x, y \in \Sigma^*$, then $x \equiv_L y$.

Myhill-Nerode Theorem for non-regularity

Myhill-Nerode Theorem, in contraposition

L is non-regular if and only if there is an infinite set $S \subseteq \Sigma^*$ consisting of pairwise distinguishable strings.

Myhill-Nerode Theorem for non-regularity

Myhill-Nerode Theorem, in contraposition

L is non-regular if and only if there is an infinite set $S \subseteq \Sigma^*$ consisting of pairwise distinguishable strings.

- Mind that we seek for distinguishable strings, which are not necessarily in L.
- For pairwise distinguishable strings $S = \{s_1, \ldots, s_m, \cdots\}$, a distinguishing extension for (s_i, s_j) might be in general different from a distinguishing extension for (s_j, s_k).

Myhill-Nerode Theorem for non-regularity, example

- $L_1 = \{0^n 1^n \mid n \geq 1\}$
- $L_2 = \{w \in \{0,1\}^* \mid w \text{ is a palindrome}\}$

Strategy: find an infinite subset of Σ^* which consists of pairwise distinguishable (inequivalent) strings.

MSO LOGIC ON STRINGS

We saw several, all equivalent, characterization of regular language.

- DFA / NFA (algorithm)
- Regular expression (composability via basic operations)
- Recognizability by monoid (algebraic property)
- Myhill-Nerode Theorem
- Generated by left/right linear grammar (not covered, yet)
- Definability by Monadic Second Order logic

MSO LOGIC ON STRINGS, BY EXAMPLE

We want to express the language

$$L = \{w \in \{0,1\}^* \mid w \text{ does not contain 11 as a substring}\}$$

with an MSO-sentence.

MSO-SENTENCE

$$\varphi = \forall x \forall y \left((x < y) \rightarrow \left(\exists z \, (x < z < y) \vee P_0(x) \vee P_0(y)\right)\right)$$

Here, $P_0(x)$ is read as "the x-th symbol in the string is 0".

Likewise, $P_1(y)$ is read as "the y-th symbol in in the string is 1".

10010 satisfies φ whereas 1101 not, which we denote as $10010 \models \varphi$ and $1101 \not\models \varphi$.

MSO LOGIC ON STRINGS, BY EXAMPLE

We want to express that

a set S of positions in the given string forms an "interval".

MSO-FORMULA

$$\varphi_{int}(S) = \forall x\, \forall y\, \big((x \in S \wedge y \in S \wedge x \leq y) \rightarrow \big(\forall z\, (x \leq z \leq y) \rightarrow z \in S\big)\big)$$

Note that the validity of $\varphi_{int}(S)$ depends not only on the given string, but also the variable S.

MSO LOGIC ON STRINGS

We first express a string $s \in \Sigma^*$ as a logical structure (often called "relational structure").

String w as a logical structure

Universe $= [n]$, where n is the length of the string.

- That is, each "position" (from 1 to n) in the string is an element in the universe. If $w = \epsilon$, the universe is $\emptyset$.

A binary relation $<$ and $|\Sigma|$ unary relations P_a for all $a \in \Sigma$ on the universe.

- $x < y$: "the x-th position precedes the y-th position in the string."
- $P_0(x)$ is true if "the x-th symbol is 0."

$\tau = \{<\} \cup \{P_a \mid a \in \Sigma\}$ is called the vocabulary on Σ-strings.

Lec 06. More MSO & Properties of Regular Languages

Eunjung Kim

MSO LOGIC ON STRINGS

We first express a string $s \in \Sigma^*$ as a logical structure (often called "relational structure").

String w as a logical structure

Universe $= [n]$, where n is the length of the string.

- That is, each "position" (from 1 to n) in the string is an element in the universe. If $w = \epsilon$, the universe is $\emptyset$.

A binary relation $<$ and $|\Sigma|$ unary relations P_a for all $a \in \Sigma$ on the universe.

- $x < y$: "the x-th position precedes the y-th position in the string."
- $P_0(x)$ is true if "the x-th symbol is 0."

$\tau = \{<\} \cup \{P_a \mid a \in \Sigma\}$ is called the vocabulary on Σ-strings.

MSO LOGIC ON STRINGS

MSO-FORMULA ON Σ-STRINGS

An MSO-formula on strings is a well-formed string that can be constructed using from atomic formulas for (infinite supply of) individual variables $x, y, z \ldots$, and set variables $X, Y, Z \cdots$ i.e.

- $x < y$; note that $< \in \tau$,
- $P_a(x)$ for each $a \in \Sigma$,
- $x = y$, and $x \in X$.

by applying

- the logical connectives $\wedge, \vee, \neg, \rightarrow$; $\varphi_1 \wedge \varphi_2$, $\neg\varphi$, etc,
- the universal and existential quantifier $\forall, \exists$; in the form $\exists x\varphi$, $\exists X\varphi$, etc.

An MSO-formula in which all variables are quantified (by $\forall$ or $\exists$) is called an MSO-sentence.

MSO LOGIC ON STRINGS

A property = the set of all Σ-strings which has the property.

A PROPERTY ON STRINGS AS AN MSO-SENTENCE

We say that a property $L \subseteq \Sigma^*$ on strings (a.k.a. a language) is expressible, or equivalently definable, in MSO if there is an MSO-sentence φ on Σ-strings such that

$$w \in L \text{ if and only if } w \models \varphi$$

for every string $w \in \Sigma^*$.

$$\varphi = \forall x \forall y \left((x < y) \rightarrow \left(\exists z \, (x < z < y) \vee P_0(x) \vee P_0(y) \right) \right)$$

MSO LOGIC ON STRINGS, BY EXAMPLE

Let us express the property L on $\{0,1\}$-strings having even number of 1's, i.e.

$$L = \{w \in \{0,1\}^* \mid \text{there are even number of 1's in } w\}.$$

Use the fact that $w \in L$ if and only if

- either $w = \epsilon$,
- or the positions of 1's in w can be "uniquely colored" in RED or BLUE so that the colors alternate, and the first 1 is RED and the last 1 is in BLUE.

MSO LOGIC ON STRINGS, BY EXAMPLE

MSO-FORMULA DEFINING L

- $\varphi_\epsilon = \neg\exists x\ (x = x)$
- $\varphi_{color}(R, B) = \forall x\ (P_1(x) \rightarrow (x \in R \vee x \in B)) \wedge (P_0(x) \rightarrow \neg(x \in R \vee x \in B))$
- $\varphi_{unique}(R, B) = \forall x\ (x \in R \rightarrow \neg x \notin B) \wedge (x \in B \rightarrow \neg x \notin R)$
- $\varphi_{alternate}(R, B) =$??????
- $\varphi_{firstlast}(R, B) =$??????

Finally, we get a sentence φ_L defining L as

$$\varphi_L = \varphi_\epsilon \vee \exists R\ \exists B\ \varphi_{color}(R, B) \wedge \varphi_{unique}(R, B) \wedge \varphi_{alternate} \wedge \varphi_{firstlast}$$

Büchi's Theorem 1960

Recognizability equals definability on strings

A language is regular if and only if it is definable in Mso.

Proof sketch of ($\Rightarrow$).

- Show that for each atomic regular expressions ($\emptyset$, ϵ, a for each $a \in \Sigma$), the corresponding language can be defined in Mso.
- Show that the languages of $R_1 \cup R_2$, $R_1 \circ R_2$ and R_1^* can be defined in Mso, assuming that $L(R_1), L(R_2)$ can be defined in Mso.

Büchi's Theorem 1960

How to define the language of an atomic regular expression in MSO.

- $\emptyset$
- ϵ
- a for each $a \in \Sigma$.

Büchi's Theorem 1960

How to define the language of $\cup, \circ, *$ in MSO assuming that $L(R_1), L(R_2)$ are defined in MSO.

- $R_1 \cup R_2$
- $R_1 \circ R_2$
- R_1^*

MSO LOGIC ON STRINGS, BY EXAMPLE

Define in MSO

$$L = L_1 \circ L_2$$

where $L_1 = L((00)^+)$ and $L_2 = L((11)^+)$.

Questions to examine

1. Given an NFA M, decide if $L(M) = \emptyset$ or not.
2. Given two regular languages L_1 and L_2, decide if $L_1 = L_2$.
3. Is $\mathit{Prefix}(L)$ is regular when L is regular?
4. How about $\mathit{Suffix}(L)$?
5. Quotient of L by a symbol $a \in \Sigma$, denoted by L/a, is regular when L is?
6. How about $a \setminus L$?
7. Fix a DFA M and a state $s \in Q$. The set of all strings w such that the (accepting) computation history of w visits the state s, is it regular?
8. Fix a DFA M. The set of all strings w such that the (accepting) computation history of w visits all the state of M, is it regular?

Deciding if $L = \emptyset$

Given a regular language L, we want to decide if $L = \emptyset$ or not.

L is given by NFA N

$L(N) \neq \emptyset$ if and only if there is a directed path from the initial state q_0 to OOOOOOOOOO in the transition diagram of N.

Recall: $w \in \Sigma^*$ satisfies $\delta^*(q_0, w) = q$ if and only if there is a (q_0, q)-walk in the transition diagram labelled by w (ϵ-label allowed).

Deciding if $L = \emptyset$

Given a regular expression R, we want to decide if $L(R) = \emptyset$ or not.
You can convert R into an NFA and apply the previous criteria, or do the following.

L is given by a regular expression R

If there is no occurrence of $\emptyset$ in R, $L(R) \neq \emptyset$.

Otherwise, check if $L(R) = \emptyset$ inductively:

1. $L(R_1 \cup R_2) = \emptyset$ if and only if $L(R_1) = \emptyset$ and $L(R_2) = \emptyset$.
2. $L(R_1 \cdot R_2) = \emptyset$ if and only if $L(R_1) = \emptyset$ or $L(R_2) = \emptyset$.
3. $L(R^\star) \neq \emptyset$ (even when $R = \emptyset$).

When L is regular, so is $Prefix(L)$?

Given two strings $x, w \in \Sigma^*$, x is a prefix of w if $w = xy$ for some $y \in \Sigma^*$.
For a language $L \subseteq \Sigma^*$, let $Prefix(L) = \{x \in \Sigma^* : x \text{ is a prefix of } w \in L\}$.

If L is regular, $Prefix(L)$ is regular

Let M be an DFA with $L = L(M)$. Notice that

$w \in L$ can be written as $w = xy$ if and only if $\hat{\delta}(q_0, x) = q$ for some state $q \in Q$ such that .....

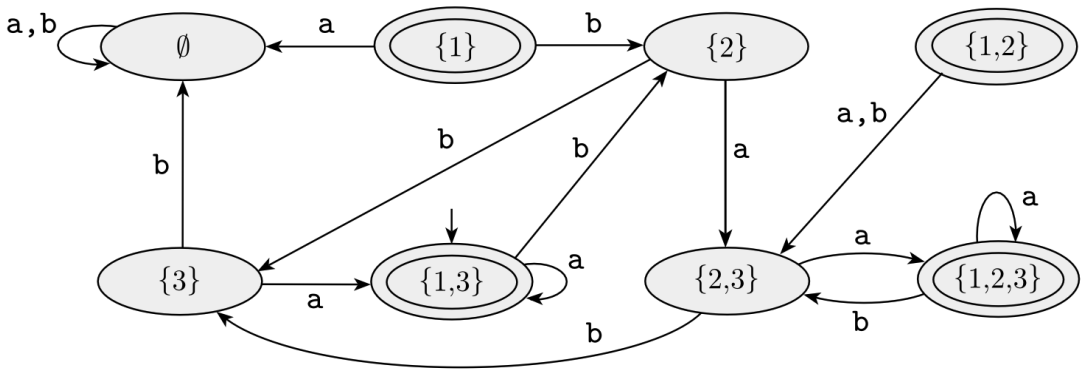


Figure 1.43, Sipser 2012

When L is regular, so is $Prefix(L)$?

Let M be an DFA with $L = L(M)$. Then

$$Prefix(L) = \bigcup_{q \in Q \text{ such that....}} L_q.$$

where $L_q = \{x \in \Sigma^* : \hat{\delta}(q_0, x) = q\}$.

- If L_q is regular, then $Prefix(L)$ is regular (why?)
- Is L_q regular?
- $properPrefix(L)$ be the set of all proper prefixes of some $w \in L$; x is a proper prefix of w if $w = xy$ for some $y \in \Sigma^+$.
- Is $properPrefix(L)$ regular?

When L is regular, so is $Suffix(L)$?

Given two strings $x, w \in \Sigma^*$, x is a suffix of w if $w = yx$ for some $y \in \Sigma^*$.
For a language $L \subseteq \Sigma^*$, let $Suffix(L) = \{x \in \Sigma^* : x \text{ is a suffix of } w \in L\}$.

If L is regular, $Suffix(L)$ is regular

Let $reverse(L)$ be the set of all strings each of which is a reversal w^R of some string $w \in$L.

- If L is regular, $reverse(L)$ is regular as well; homework.
- $Suffix(L)$ can be obtained by applying ???? and ???? operations on L.

Quotient L/a for $a \in \Sigma$

Given a language L over Σ and a symbol $a \in \Sigma$, the quotient of L by a denoted as L/a is the language

$$\{x \in \Sigma^* : xa \in L\}.$$

Is L/a regular?

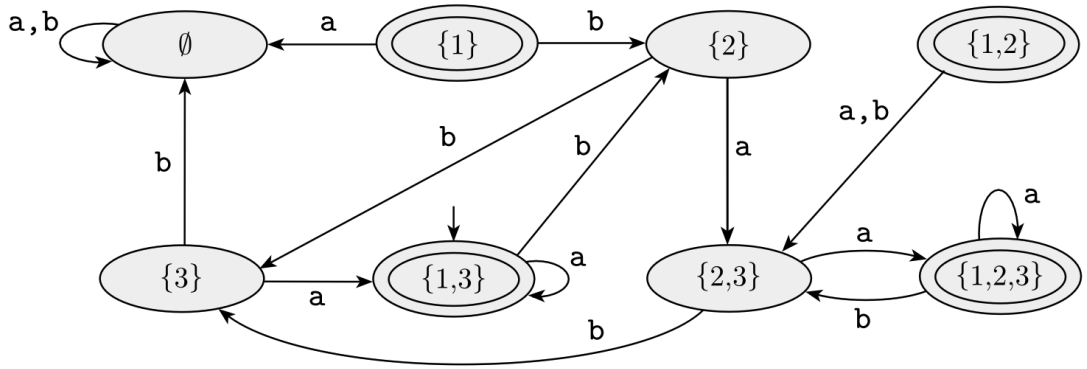


Figure 1.43, Sipser 2012

- For a state $q \in Q$, if $x \in L_q$ satisfies $xa \in L$ for some x, then for all $y \in L_q$ we have $ya \in L$.
- That is, $L_q \subseteq L/a$ or $L_q \cap L/a = \emptyset$.
- How to tell if $L_q \subseteq L/a$?

The language $a \setminus L$ for $a \in \Sigma$

Given a language L over Σ and a symbol $a \in \Sigma$, the language $a \setminus L$ is defined as

$$\{x \in \Sigma^* : ax \in L\}.$$

Is $a \setminus L$ regular?

Idea: Express $a \setminus L$ using the operations we examined so far to immediately conclude.

More exotic language P_s

- Fix a DFA M and a state $s \in Q$.
- Let P_s be the set of all string $w \in L$ such that the accepting computation history of w visits the state s.
- Is P_s regular?

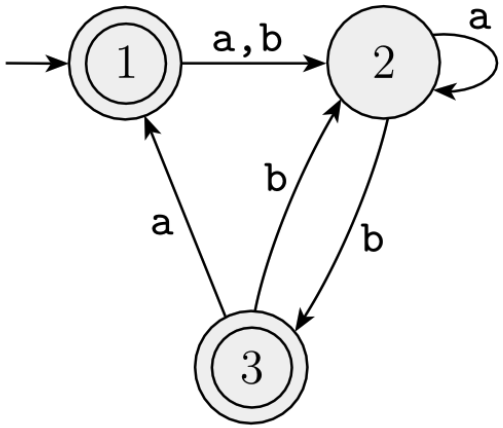


Figure 1.21, Sipser 2012

More exotic language P_s

First approach.

- For any string w, $w \in P_s$ if and only if it can be written as $w = xy$ with $\hat{\delta}(q_0, x) = s$ and $\hat{\delta}(s, y) \in F$.
- That is $P_s = L_s \cdot A_s$, where L_q and A_q are defined for all $q \in Q$ as

$$L_q = \{x \in \Sigma^* : \hat{\delta}(q_0, x) = q\}.$$

$$A_q = \{x \in \Sigma^* : \hat{\delta}(q, x) \in F\}.$$

- Is any one of L_q and A_q regular?

More exotic language P_s

Second approach: use Myhill-Nerode Theorem.

Myhill-Nerode Theorem

L is regular if and only if the number of equivalence classes of $\equiv_L$ is finite.

Idea: use the DFA M recognizing L to identify the equivalence relation $\equiv_{P_s}$, (or a refinement of it) of finite index.

- For $Z \subseteq Q$ and $q \in W$, let $L_{Z,q}$ be the set of all strings w such that the computation history of w on M visits precisely the states in Z and end in q.
- $\Sigma^* = \dot{\bigcup}_{Z \subseteq Q, q \in Z} L_{w,q}$.
- We want to argue that any strings $x, y \in L_{Z,q}$ are indistinguishable by P_s. But for proving this claim, we need to try *all strings z which might potentially distinguish x and y... or do we?*

More exotic language P_s

Second approach: use Myhill-Nerode Theorem and test for a finite number of extensions z (and argue that it suffices).

Myhill-Nerode Theorem, in action

P_s is regular if for any $Z \subseteq Q$ and $q \in Z$,

- any $x, y \in L_{Z,q}$ are indistinguishable by P_s, or equivalently
- for any $x, y \in L_{Z,q}$ and for any $z \in \Sigma^*$, $xz \in P_s$ if and only if $yz \in P_s$.

What are the key property of z which will make $xz \in P_s$ (or not) for $x \in L_{Z,q}$?

More exotic language P_s

Second approach: use Myhill-Nerode Theorem and test for a finite number of extensions z (and argue that it suffices).

Myhill-Nerode Theorem, in action

P_s is regular if for any $Z \subseteq Q$ and $q \in Z$,

- any $x, y \in L_{Z,q}$ are indistinguishable by P_s, or equivalently
- for any $x, y \in L_{Z,q}$ and for any $z \in \Sigma^*$, $xz \in P_s$ if and only if $yz \in P_s$.

What are the key property of z which will make $xz \in P_s$ (or not) for $x \in L_{Z,q}$?

1. whether $\delta^*(q, z) \in F$ or not: this dictates whether $xz \in L$.
2. whether the states visited by the computation history of $\delta^*(q, z)$ include s or not: this affects whether the computation history of xz from q_0 visits s or not.

A bit more exotic language

Fix a DFA M. The set of all strings w such that the (accepting) computation history of w visits all the state of M, is it regular?

Even More exotic language

Why do we care about the second approach using Myhill-Nerode theorem when the first approach seems much simpler?

Even more exotic language. Fix two states s_1, s_2 of a DFA M. Let P_{s_1,s_2} be the set of strings $w \in L$ whose computation history visits both s_1, s_2 and visiting s_2 only after visiting s_1.

Is P_{s_1,s_2} regular?

Lec 08. More on regular language & Context-free grammar

Eunjung Kim

When L is regular, so is $Prefix(L)$?

Given two strings $x, w \in \Sigma^*$, x is a prefix of w if $w = xy$ for some $y \in \Sigma^*$.
For a language $L \subseteq \Sigma^*$, let $Prefix(L) = \{x \in \Sigma^* : x \text{ is a prefix of } w \in L\}$.

If L is regular, $Prefix(L)$ is regular

Let M be an DFA with $L = L(M)$. Notice that

$w \in L$ can be written as $w = xy$ if and only if $\hat{\delta}(q_0, x) = q$ for some state $q \in Q$ such that .....

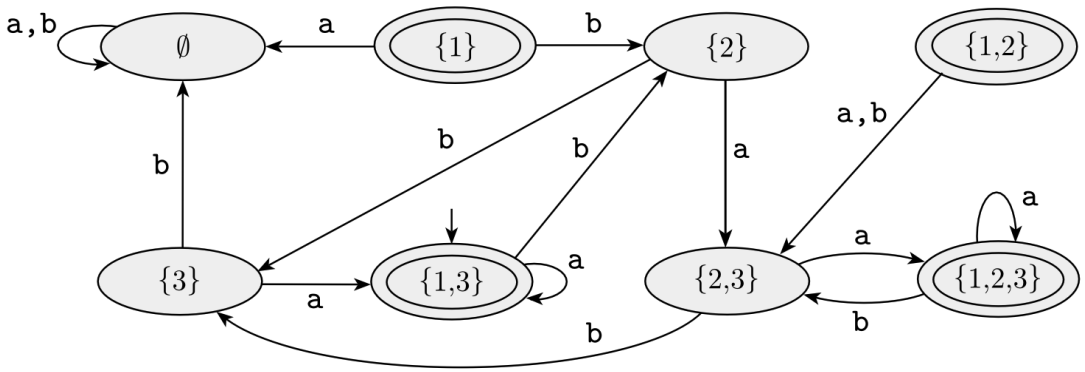


Figure 1.43, Sipser 2012

When L is regular, so is $Prefix(L)$?

Let M be an DFA with $L = L(M)$. Then

$$Prefix(L) = \bigcup_{q \in Q \text{ such that....}} L_q.$$

where $L_q = \{x \in \Sigma^* : \hat{\delta}(q_0, x) = q\}$.

- If L_q is regular, then $Prefix(L)$ is regular (why?)
- Is L_q regular?
- $properPrefix(L)$ be the set of all proper prefixes of some $w \in L$; x is a proper prefix of w if $w = xy$ for some $y \in \Sigma^+$.
- Is $properPrefix(L)$ regular?

When L is regular, so is $Suffix(L)$?

Given two strings $x, w \in \Sigma^*$, x is a suffix of w if $w = yx$ for some $y \in \Sigma^*$.
For a language $L \subseteq \Sigma^*$, let $Suffix(L) = \{x \in \Sigma^* : x \text{ is a suffix of } w \in L\}$.

If L is regular, $Suffix(L)$ is regular

Let *reverse*(L) be the set of all strings each of which is a reversal w^R of some string $w \in$L.

- If L is regular, *reverse*(L) is regular as well; homework.
- *Suffix*(L) can be obtained by applying ???? and ???? operations on L.

Quotient L/a for $a \in \Sigma$

Given a language L over Σ and a symbol $a \in \Sigma$, the quotient of L by a denoted as L/a is the language

$$\{x \in \Sigma^* : xa \in L\}.$$

Is L/a regular?

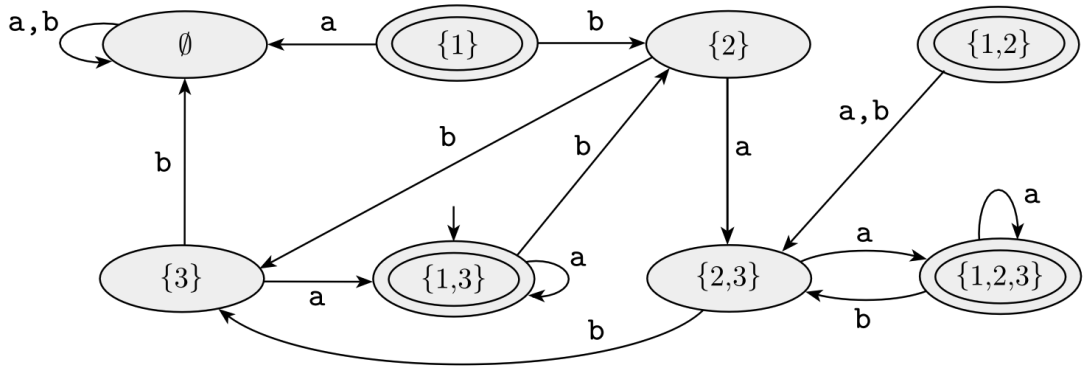


Figure 1.43, Sipser 2012

- For a state $q \in Q$, if $x \in L_q$ satisfies $xa \in L$ for some x, then for all $y \in L_q$ we have $ya \in L$.
- That is, $L_q \subseteq L/a$ or $L_q \cap L/a = \emptyset$.
- How to tell if $L_q \subseteq L/a$?

The language $a \setminus L$ for $a \in \Sigma$

Given a language L over Σ and a symbol $a \in \Sigma$, the language $a \setminus L$ is defined as

$$\{x \in \Sigma^* : ax \in L\}.$$

Is $a \setminus L$ regular?

Idea: Express $a \setminus L$ using the operations we examined so far to immediately conclude.

More exotic language P_s

- Fix a DFA M and a state $s \in Q$.
- Let P_s be the set of all string $w \in L$ such that the accepting computation history of w visits the state s.
- Is P_s regular?

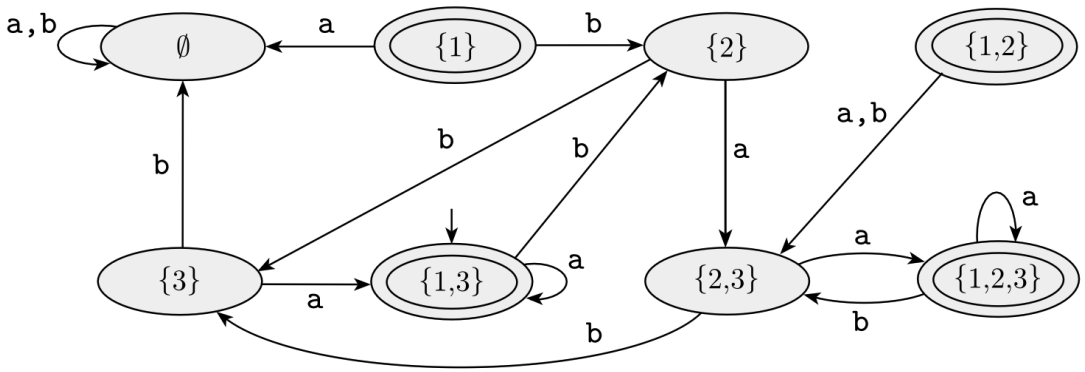


Figure 1.43, Sipser 2012

More exotic language P_s

First approach.

- For any string w, $w \in P_s$ if and only if it can be written as $w = xy$ with $\hat{\delta}(q_0, x) = s$ and $\hat{\delta}(s, y) \in F$.
- That is $P_s = L_s \cdot A_s$, where L_q and A_q are defined for all $q \in Q$ as

$$L_q = \{x \in \Sigma^* : \hat{\delta}(q_0, x) = q\}.$$

$$A_q = \{x \in \Sigma^* : \hat{\delta}(q, x) \in F\}.$$

- Is any one of L_q and A_q regular?

More exotic language P_s

Second approach: use Myhill-Nerode Theorem.

Myhill-Nerode Theorem

L is regular if and only if the number of equivalence classes of $\equiv_L$ is finite.

Idea: use the DFA M recognizing L to identify the equivalence relation $\equiv_{P_s}$, (or a refinement of it) of finite index.

- For $T \subseteq Q$ and $q \in T$, let $L_{T,q}$ be the set of all strings w such that the computation history of w on M visits precisely the states in T and end in q.
- $\Sigma^* = \dot{\bigcup}_{T \subseteq Q, q \in T} L_{T,q}$ (disjoint union).
- We want to argue that any strings $x, y \in L_{T,q}$ are indistinguishable by P_s.
- This shows that P_s has finitely many equivalent classes, thus regular.

More exotic language P_s

Second approach: use Myhill-Nerode Theorem

Myhill-Nerode Theorem, in action

P_s is regular if for any $T \subseteq Q$ and $q \in T$,

- any $x, y \in L_{T,q}$ are indistinguishable by P_s, or equivalently
- for any $x, y \in L_{T,q}$ and for any $z \in \Sigma^*$, $xz \in P_s$ if and only if $yz \in P_s$.

What are the key property of z which will make $xz \in P_s$ (or not) for $x \in L_{T,q}$?

More exotic language P_s

Second approach: use Myhill-Nerode Theorem

Myhill-Nerode Theorem, in action

P_s is regular if for any $T \subseteq Q$ and $q \in T$,

- any $x, y \in L_{T,q}$ are indistinguishable by P_s, or equivalently
- for any $x, y \in L_{T,q}$ and for any $z \in \Sigma^*$, $xz \in P_s$ if and only if $yz \in P_s$.

What are the key property of z which will make $xz \in P_s$ (or not) for $x \in L_{T,q}$?

1. whether $\hat{\delta}(q, z) \in F$ or not: this determines whether $xz \in L$ or not.
2. whether $s \in T$ or not.
3. whether s is visited during the computation history of $\hat{\delta}(q, z)$ or not.

A bit more exotic language

Fix a DFA M. The set of all strings w such that the (accepting) computation history of w visits all the state of M, is it regular?

Even More exotic language

- Why do we care about the second approach using Myhill-Nerode theorem when the first approach seems much simpler?
- Even more exotic language. Fix two states s_1, s_2 of a DFA M. Let P_{s_1,s_2} be the set of strings $w \in L$ whose computation history visits each of s_1, s_2 exactly once.
- Is P_{s_1,s_2} regular?

What we learned so far

- Finite (state) automata: a machine with limited memory.
- Nondeterministic FA has the extra feature of making multiple transitions in parallel and ϵ-transition. Conversions between DFA and NFA possible (no added power).
- Regular expression: describes the 'shape' of a regular language directly.
- Conversion between regular expression and NFA using Generalized NFA.
- The class of regular languages is closed under various operations.
- Pumping lemma as a tool to prove that a language is nonregular.
- Myhill-Nerode Theorem as a powerful characterization of regular languages.
- Büchi's Theorem as another characterization of regular language using logic.
- One can prove various properties of NFA/DFA and regular language combining the tools we learned.

Context-Free Language

Expressing palindromes

- A string w is a palindrome if and only if $w = w^R$.
- The set of palindromes (e.g. over $\{0, 1\}$) is not regular, so one cannot use a regular expression or NFA to describe the language.
- Recursive definition:
  1. Base case: ϵ, 0 and 1 are palindromes.
  2. Induction: if w is a palindrome, then $0w0$ and $1w1$ are palindromes.
- Any word generated in this way is a palindrome.
- Conversely, any palindrome can be generated in this way: a word of the form $x \cdot w \cdot y$ with $x, y \in \Sigma$ is a palindrome (if and) only if $x = y$ and w is a palindrome.

Expressing palindromes

- Recursive definition:
  1. Base case: ϵ, 0 and 1 are palindromes.
  2. Induction: if w is a palindrome, then $0w0$ and $1w1$ are palindromes.
- Definition by rules.
  1. $S \to \epsilon$.
  2. $S \to 0$.
  3. $S \to 1$.
  4. $S \to 0S0$.
  5. $S \to 1S1$.
- Any word generated using the above rules is a palindrome.
- Conversely, any palindrome can be generated using the above rules.

Context-free grammars (CFG)

Definition of Context-free grammar

There are four component of CFG $G = (V, \Sigma, R, S)$.

1. A finite set of nonterminals, often called the variables and denoted by V.
2. A finite set of terminals (equivalently, alphabet) Σ.
3. A finite set of rules R (often called substitution rules/ production rules) in the form

$$X \to \gamma,$$

   where
   - X is a variable; $X \in V$.
   - γ is a string of terminal and nonterminal symbols; $\gamma \in (\Sigma \cup V)^*$.
4. A unique nonterminal symbol, often denoted as S, called the start symbol.

A quadruple $G = (V, \Sigma, R, S)$ is a context-free grammar (CFG) if the four components are as above.

Expressing palindromes

- Consider the grammar $G_{pal} = (\{S, \}, \{0, 1\}, R, S)$, where R is the five production rules below.
  1. $S \to \epsilon$.
  2. $S \to 0$.
  3. $S \to 1$.
  4. $S \to 0S0$.
  5. $S \to 1S1$.
  6. (or equivalently, we write all the bodies of rules sharing the same head) $S \to \epsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1$.
- Observe: a word over $\{0, 1\}$ is a palindrome if and only if it can be derived from S, that is, by a recursively substituting a variable using one of the substitution rules.
- In other words, the language of palindromes over $\{0, 1\}$ is precisely the language of the grammar G_{pal}, denoted as $L(G_{pal})$.

Expressing MSO-formulae on strings using "rules"

Recall that we defined Mso-formula on Σ-strings as a well-formed strings such that ....

- Definition by rules.
  1. $\varphi \to x = x \,|x \in X \mid x < x \mid P_a(x)$ (for each $a \in \Sigma$).
  2. $\varphi \to (\varphi)$.
  3. $\varphi \to \varphi \wedge \varphi \,|\varphi \vee \varphi$.
  4. $\varphi \to \exists x\, \varphi \mid \forall x\, \varphi \mid \exists X\, \varphi \mid \forall X\, \varphi$
  5. $x \to x_1 \mid x_2 \mid x_3 \mid \cdots$.
  6. $X \to X_1 \mid X_2 \mid X_3 \mid \cdots$.
- Any string generated using the above rules is an Mso-formula.
- Conversely, any Mso-formula can be generated using the above rules.

Derivation

Consider a CFG $G = (V, \Sigma, R, S)$, $u, v, w \in (\Sigma \cup V)^*$ (a string of terminals and nonterminals) and a variable (nonterminal) $A \in V$.

Yield, derive, derivation

- We say that uAv yields uwv, written $uAv \Rightarrow_G uwv$, if G has the rule $A \to w$; put another way, uwv is obtained by substituting a variable in the string uAv by the body of a rule whose head is the said variable.
- We say that u derives v, written $u \Rightarrow_G^* v$. if $u = v$ or there is a sequence $u_1, \ldots, u_k$ for some $k \geq 1$ such that

$$u \Rightarrow_G u_1 \Rightarrow_G \cdots \Rightarrow_G u_k \Rightarrow_G v$$

and the sequence is called a derivation.

We omit the subscript G in $\to_G$ and $\Rightarrow_G$ if the CFG under consideration is clear in the context.

Derivation by example

We want to describe, as CFG,

- all arithmetic expressions with $+$ and $\times$
- over the variables of the form $(a \cup b)(a \cup b \cup 0 \cup 1)^*$.

Consider the following CFG $G_{ari} = (\{E, I\}, \{a, b, 0, 1, +, \times, (,)\}, R, E)$, where R consists of the following rules

1. $E \to I$
2. $E \to E + E$
3. $E \to E \times E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

Context-free language

For a CFG $G = (V, \Sigma, R, S)$, the language of G, denoted by $L(G)$ is the set of all strings consisting of terminals (only) that have derivations from the start symbol, i.e.

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}.$$

A language is said to be context-free if it is the language of a context-free grammar. A context-free languages is often abbreviated as CFL.

Some remarks on CFG

- In general, the rule of a grammar has the form $u \to v$ with both u and v are strings of terminals and nonterminals.
- A grammar is context-free if the head u is a nonterminal (variable) in all the rules; we do not need to consider the context.
- Different restrictions on the grammar define the hierarchy of formal languages.

Class	Languages	Automaton	Rules	Word Problem	Example
type-0	recursively enumerable	Turing machine	no restriction	undecidable	Post's corresp. problem
type-1	context sensitive	linear-bounded TM	$\alpha \to \gamma$ $\lvert\alpha\rvert \leq \lvert\gamma\rvert$	PSPACE-complete	$a^n b^n c^n$
type-2	context free	pushdown automaton	$A \to \gamma$	cubic	$a^n b^n$
type-3	regular	NFA / DFA	$A \to a$ or $A \to aB$	linear time	$a^* b^*$

Figure 1: Chomsky Hierarchy

Figure 1, Lecture note on 15-411: Compiler Design, CMU, 2023

Lec 09. Context-free language and Parse trees

Eunjung Kim

Derivation

Consider a CFG $G = (V, \Sigma, R, S)$, $u, v, w \in (\Sigma \cup V)^*$ (a string of terminals and nonterminals) and a variable (nonterminal) $A \in V$.

Yield, derive, derivation

- We say that uAv yields uwv, written $uAv \Rightarrow_G uwv$, if G has the rule $A \to w$; put another way, uwv is obtained by substituting a variable in the string uAv by the body of a rule whose head is the said variable.
- We say that u derives v, written $u \Rightarrow_G^* v$. if $u = v$ or there is a sequence $u_1, \ldots, u_k$ for some $k \geq 1$ such that

$$u \Rightarrow_G u_1 \Rightarrow_G \cdots \Rightarrow_G u_k \Rightarrow_G v$$

and the sequence is called a derivation.

We omit the subscript G in $\to_G$ and $\Rightarrow_G$ if the CFG under consideration is clear in the context.

Derivation by example

We want to describe, as CFG,

- all arithmetic expressions with $+$ and $\times$
- over the variables of the form $(a \cup b)(a \cup b \cup 0 \cup 1)^*$.

Consider the following CFG $G_{ari} = (\{E, I\}, \{a, b, 0, 1, +, \times, (,)\}, R, E)$, where R consists of the following rules

1. $E \to I$
2. $E \to E + E$
3. $E \to E \times E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

Context-free language

For a CFG $G = (V, \Sigma, R, S)$, the language of G, denoted by $L(G)$ is the set of all strings consisting of terminals (only) that have derivations from the start symbol, i.e.

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}.$$

A language is said to be context-free if it is the language of a context-free grammar. A context-free languages is often abbreviated as CFL.

Some remarks on CFG

- In general, the rule of a grammar has the form $u \to v$ with both u and v are strings of terminals and nonterminals.
- A grammar is context-free if the head u is a nonterminal (variable) in all the rules; we do not need to consider the context.
- Different restrictions on the grammar define the hierarchy of formal languages.

Class	Languages	Automaton	Rules	Word Problem	Example
type-0	recursively enumerable	Turing machine	no restriction	undecidable	Post's corresp. problem
type-1	context sensitive	linear-bounded TM	$\alpha \to \gamma$ $\lvert\alpha\rvert \leq \lvert\gamma\rvert$	PSPACE-complete	$a^n b^n c^n$
type-2	context free	pushdown automaton	$A \to \gamma$	cubic	$a^n b^n$
type-3	regular	NFA / DFA	$A \to a$ or $A \to aB$	linear time	$a^* b^*$

Figure 1: Chomsky Hierarchy

Figure 1, Lecture note on 15-411: Compiler Design, CMU, 2023

Grammar and meaning of the language

Let us show that $L(G_{pal})$ is precisely the set of palindromes consisting of 0's and 1's.

- ($\rightarrow$) We want to show that if $S \Rightarrow^* w$, then w is a palindrome. Induction on the length of derivation.
  1. Base: if length at most 1, then $w = \epsilon, 0$ or 1, which is trivially a palindrome.
  2. Induction: if the derivation has length $n+1$, then it is of the form

$$S \Rightarrow 0S0 \Rightarrow^* 0x0 = w$$

     (or 0 replaced by 1) where $S \Rightarrow^* x$ is a derivation of length n. By I.H. x is a palindrome, and thus $0x0$ is.

Grammar and meaning of the language

Let us show that $L(G_{pal})$ is precisely the set of palindromes consisting of 0's and 1's.

- ($\rightarrow$) We want to show that if $S \Rightarrow^* w$, then w is a palindrome. Induction on the length of derivation.
  1. Base: if length at most 1, then $w = \epsilon, 0$ or 1, which is trivially a palindrome.
  2. Induction: if the derivation has length $n+1$, then it is of the form

     $$S \Rightarrow 0S0 \Rightarrow^* 0x0 = w$$

     (or 0 replaced by 1) where $S \Rightarrow^* x$ is a derivation of length n. By I.H. x is a palindrome, and thus $0x0$ is.
- ($\leftarrow$) We want to show that if w is a palindrome, then $S \Rightarrow^* w$. Induction on $|w|$.
  1. Base: $|w| \leq 1$, then $S \Rightarrow w$ by a single application of one of the rules $S \rightarrow \epsilon|0|1$.
  2. Induction: if w is a palindrome of length ≥ 2, it is of the form $0x0$ or $1x1$ for some palindrome x. By I.H., $S \Rightarrow^* x$. Therefore,

     $$S \Rightarrow 0S0 \Rightarrow^* 0x0 = w$$

Grammar and derivation

Consider CFG G with the following rules.

- $S \to XSX \mid R$
- $R \to aTb \mid bTa$
- $T \to XTX \mid X \mid \epsilon$
- $X \to a \mid b$

1. Variables? Terminals? Start Variable?
2. $T \Rightarrow^* T$?
3. $T \Rightarrow^* XXX$?
4. $XXX \Rightarrow^* aba$?
5. $R \Rightarrow^* \epsilon$?
6. $S \Rightarrow^* abaababbaaba$?

Grammar and meaning of the language

Consider CFG G with the following rules. Describe the language $L(G)$ in plain English.

1. $S \rightarrow aSb \mid aA \mid bB$
2. $A \rightarrow aA \mid \epsilon$
3. $B \rightarrow bB \mid \epsilon$

Designing a context-free grammar

1. $L = \{0^n 1^n : n \geq 1\}$.
2. $L = \{0^n 1^n : n \geq 1\} \cup \{0^n 1^n : n \geq 1\}$.
3. $L = \{a^i b^j : i > j\}$.
4. $L = \{a^i b^j c^k : i \neq j \text{ or } j \neq k\}$.
5. $L = \{\text{the set of all well-formed parentheses}\}$.
6. $L = \{\text{all strings with the same number of 0's and 1's}\}$.

How to design CFG

The design of CFG requires some ingenuity. Some useful tips here.

- Many CFLs are the union of simpler CFLs.
- It is convenient to think of a variable as something that represents a set of strings; those which can be derived from that variable.
- A CFG for a regular language is easy to construct.
- Sometimes you use some nice combinatorial property of the language.

$$L = \{0^n 1^n : n \geq 1\}.$$

$$L = \{0^n 1^n : n \geq 1\} \cup \{1^n 0^n : n \geq 1\}.$$

$$L = \{a^i b^j : i > j\}.$$

$$L = \{a^i b^j c^k : i \neq j \textbf{ OR } j \neq k\}.$$

$L = \{$**THE SET OF ALL WELL-FORMED PARENTHESES**$\}$.

$L = \{$THE SAME # OF 0'S AND 1'S$\}$.

Leftmost/rightmost derivation

Definition

A derivation is a leftmost derivation if a production rule is applied to the leftmost variable in each step. A rightmost derivation is defined similarly.

Example: a leftmost derivation of the string "$a \times (a + b00)$" in the CFG G_{ari}

1. $E \to I \mid E + E \mid E \times E \mid (E)$
2. $I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Graphic representation of the derivation

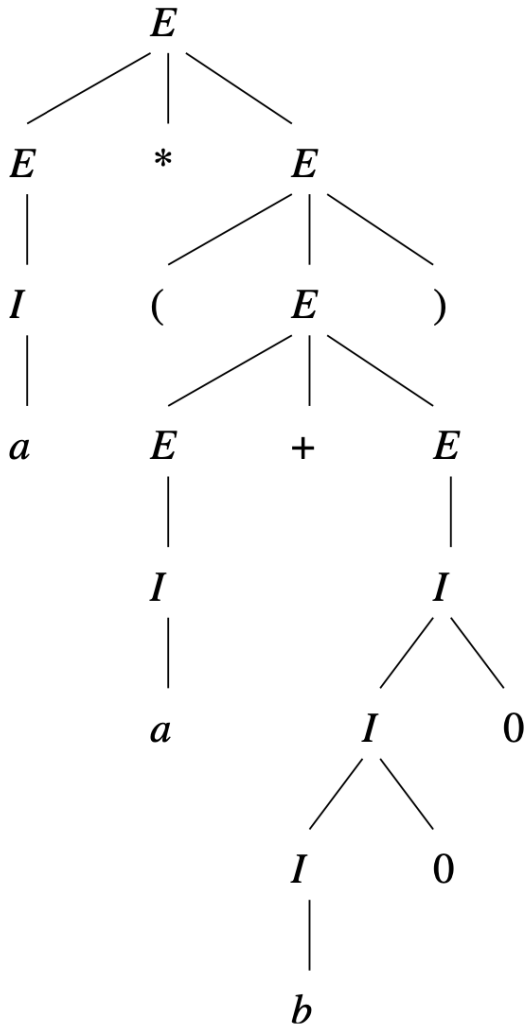


Figure 5.6, Hopcroft et. al. 2006

Parse tree

Definition

Let $G = (V, \Sigma, R, S)$ be a context-free grammar. A parse tree for the grammar G is a (rooted) tree satisfying the following.

1. Each internal node is labelled by a variable in V.
2. Each leaf is labelled by a member of $V \cup \Sigma \cup \{\epsilon\}$. If a leaf is labelled by ϵ, it is the only child of its parent.
3. If an internal node is labelled by A, and its children are labelled by

$$X_1, \ldots, X_k$$

when read from the left to right, then there is a rule $A \to X_1 \cdots X_k$ in R.

Yield of a parse tree

Definition

Let $G = (V, \Sigma, R, S)$ be a context-free grammar. The yield of a parse tree is a string in $(V \cup \Sigma \cup \{\epsilon\})^*$ obtained by concatenating the labels on the leaves of the parse tree from left to right.

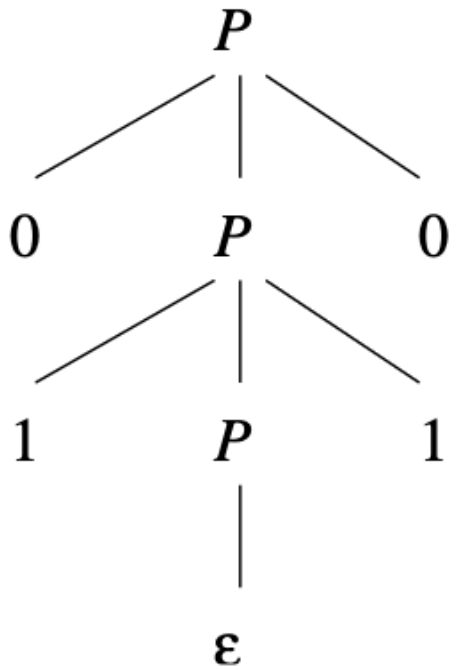


Figure 5.5, Hopcroft et. al. 2006

Equivalence of parse tree and derivation

Theorem

Let $G = (V, \Sigma, R, S)$ be a context-free grammar. The following are equivalent.

- $S \Rightarrow^* w$ *(i.e.* $w \in L(G)$*).*
- *There is a parse tree with root S and yield w.*
- $S \Rightarrow^*_{lm} w$.
- $S \Rightarrow^*_{rm} w$.

Lec 10. Ambiguity and Pushdown Automata

Eunjung Kim

Ambiguity in grammars and languages

In the grammar G_{ari}

1. $E \rightarrow I \quad | \quad E + E \quad | \quad E \times E \quad | \quad (E)$
2. $I \rightarrow a \quad | \quad b \quad | \quad Ia \quad | \quad Ib \quad | \quad I0 \quad | \quad I1$

How many parse trees that yield the string "$E + E \times E$"?

How many parse trees that yield the string "$a + b$"?

Ambiguity in grammars and languages

Ambiguous grammar

- A grammar is ambiguous if there is a string $w \in \Sigma^*$ such that there are (at least two) parse trees, in each of which the root is labelled by the start variable S and w is the yield.
- Equivalently, a grammar is ambiguous if a string has two leftmost derivations.
- A grammar is unambiguous if every string has at most one parse tree in the grammar.

Inherent ambiguity

Inherently Ambiguous CFL

- A context-free language L is inherently ambiguous if any grammar G which generates L (i.e. $L(G) = L$) is ambiguous.
- A CFL L is unambiguous if there is an unambiguous grammar G which generates L.

Eliminating ambiguity

- There is no algorithm which decides whether a given CFG is ambiguous or not ("undecidable problem").
- There are inherently ambiguous languages: e.g. $\{a^n b^n c^m d^m : n, m \geq 1\} \cup \{a^n b^m c^m d^n : n, m \geq 1\}$
- Showing if a language is inherently ambiguous or unambiguous is not easy (in terms of proof...)
- BUT, many CFL's we care are unambiguous, and there are techniques to modify the grammar to eliminate ambiguity.

Eliminating ambiguity: example 1

The grammar G_{ari} is ambiguous: e.g. $a + a \times a$ and $a + a + a$

1. $E \to I \mid E + E \mid E \times E \mid (E)$
2. $I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Goal: we want the grammar to

- respect the priority of operators, and
- generate a sequence of identical operations in a unique way, e.g. grouped from left to right.

Eliminating ambiguity: example 1

The grammar G_{ari} is ambiguous: e.g. $a + a \times a$ and $a + a + a$

1. $E \to I \quad | \quad E + E \quad | \quad E \times E \quad | \quad (E)$
2. $I \to a \quad | \quad b \quad | \quad Ia \quad | \quad Ib \quad | \quad I0 \quad | \quad I1$

Arithmetic expression with $+, \times$ are written like

$$(a0 + b1) \times a0 + b0 \times b \times b_0 + b11 \times (a1 + b0) = term + term + term$$

where each term is factored into

$$(a0 + b1) - a0,\ b_0\text{-}b\text{-}b_0 \text{ and } b11 - (a11 + b0).$$

Eliminating ambiguity: example 1

We introduce intermediary variables which represent the following.

- "Identifier": the existing variable. *I* already represents them.
- "Factor": the operands of $\times$. Identifiers or an expression surrounded by () in the expressions of G_{ari}.
- "Term": those separated by $+$ in an arithmetic expression. Either an identifier or have multiple factors.
- "Expression": any expression generated by G_{ari}. Either a term or "+" of ≥ 2 terms.

Eliminating ambiguity: example 1

We introduce intermediary variables which represent the following.

- "Identifier": the existing variable. I already represents them.
- "Factor": the operands of $\times$. Identifiers or an expression surrounded by () in the expressions of G_{ari}.
- "Term": those separated by $+$ in an arithmetic expression. Either an identifier or have multiple factors.
- "Expression": any expression generated by G_{ari}. Either a term or "+" of ≥ 2 terms.

Let us construct a CFG with the additional variables as above (Start E).

- $I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
- $F \to I \mid (E)$
- $T \to F \mid T \times F$
- $E \to T \mid E + T$

Eliminating ambiguity: example 1

New CFG generating $L(G_{ari})$ (Start E).

- $I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
- $F \rightarrow I \mid (E)$
- $T \rightarrow F \mid T \times F$
- $E \rightarrow T \mid E + T$

Parse tree for $a + a \times a$.

Eliminating ambiguity: example 1

New CFG generating $L(G_{ari})$ (Start E).

- $I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
- $F \rightarrow I \mid (E)$
- $T \rightarrow F \mid T \times F$
- $E \rightarrow T \mid E + T$

Parse tree for $a + a \times a$.

The new CFG is an unambiguous grammar generating the same language.

Eliminating ambiguity: example 2

CFG generating a well-formed parenthesis.

$$S \rightarrow SS \quad | \quad (S) \quad | \quad \epsilon$$

How many parse trees for ()()()?

Eliminating ambiguity: example 2

CFG generating a well-formed parenthesis.

$$S \rightarrow SS \quad | \quad (S) \quad | \quad \epsilon$$

How many parse trees for ()()()?

Ambiguity of the grammar arises from that a concatenation of well-formed parenthesis can be expressed by multiple parse trees.

We use the same principle for eliminating ambiguity as for the arithmetic expression.

Pushdown automata

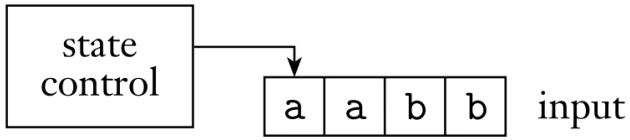


Figure 2.11, Sipser 2012

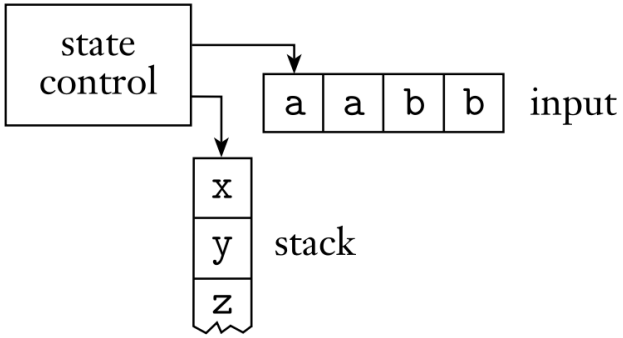


Figure 2.12, Sipser 2012

Pushdown automata

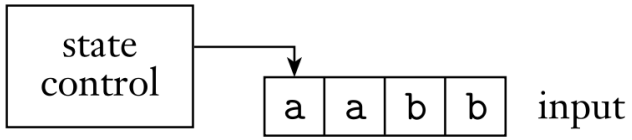


Figure 2.11, Sipser 2012

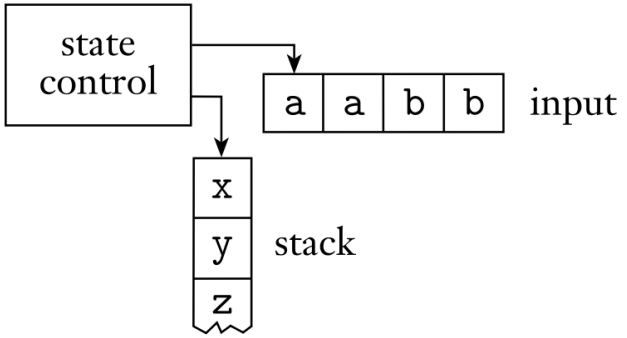


Figure 2.12, Sipser 2012

PDA: informal description

A pushdown automata has three components: an input tape, a state (control), and a stack. At each step, PDA does the following

- current state: q.
- read from input: the first symbol a of the input tape; possibly ϵ.
- pops off stack: the first symbol x at (the top of) the stack; possibly ϵ.
- transition: depending on (q, a, x), PDA
  1. updates the current state q to a new state q'
  2. pushes a symbol y to the stack; possibly ϵ.

PDA FOR $L = \{w \cdot w^R : w \in \{0,1\}^*\}$

(INFORMAL) PDA RECOGNIZING A PALINDROME

- It has three states: q_{start}, q_{push}, $q_{pop\&match}$, q_{accept}.
- Starting at q_{start}, it pushes \$ to stack and updates to q_{push}.
- Stays in q_{push} while: it reads the symbol b from input and pushes b to stack.
- "Guess" the middle of the word; implemented as an update from q_{push} to $q_{pop\&match}$
- Stays in $q_{pop\&match}$ while: it reads the symbol b from input, pops the symbol b; they match.
- Moves $q_{pop\&match}$ to q_{accept} if: there is nothing to read in the input when \$ is popped.
- In all other cases (e.g. "mismatch" between the input content and stack symbol): "dies"

Formal definition of PDA

A pushdown automata is

a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where all entries except for q_0 is a finite set:

- Q is the set of states.
- Σ is the input alphabet.
- Γ is the stack alphabet.
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition function.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is the set of accept states.

Language recognized by a PDA

PDA accepting a string

A pushdown automata $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts an input string $w \in \Sigma^*$ if

I. w can be written as $w_1 w_2 \cdots w_n$, where $w_i \in \Sigma_\epsilon$,

II. there is a sequence $r_0, r_1, \ldots, r_n$ of states, and

III. a sequence of strings $s_0, s_1, \ldots, s_n$ of strings over Γ^*,

such that the following holds:

1. $r_0 = q_0$ and $s_0 = \epsilon$,
2. $s_i = xt$, $s_{i+1} = yt$ and $(r_{i+1}, y) \in \delta(r_i, w_{i+1}, x)$ hold for some $x, y \in \Gamma_\epsilon$ and $t \in \Gamma^*$
3. $r_n \in F$.

Instantaneous description (a.k.a configuration)

Configuration of PDA

For a pushdown automata $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$, a configuration of P (also called an instantaneous description (ID)) is a triple $(q, w, \gamma) \in Q \times \Sigma^* \times \Gamma^*$. Essentially

- q is the current state.
- w is the remaining input string.
- γ is the string in the stack,read from top to bottom.

If $(q', y) \in \delta(q, a, x)$, then we write

$$(q, aw, x\beta) \vdash (q', w, y\beta).$$

This means that one can reach the configuration $(q', w, y\beta)$ from $(q, aw, x\beta)$ in a single step (transition). The notation $\vdash^*$ is used when one is reach from the other in $n \geq 0$ steps.

Language recognized by a PDA

Language recognized by PDA

For a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$, the language recognized by PDA P is defined as

- the set of all strings in Σ^* accepted by P, or equivalently
- the set of all strings $w \in \Sigma^*$ such that $(q_0, w, \epsilon) \vdash^* (q, \epsilon, \gamma)$ for some $q \in F$ and $\gamma \in \Gamma^*$.

We write as $L(P)$ the language recognized by PDA P.

PDA FOR $L = \{0^n 1^n : n \geq 0\}$

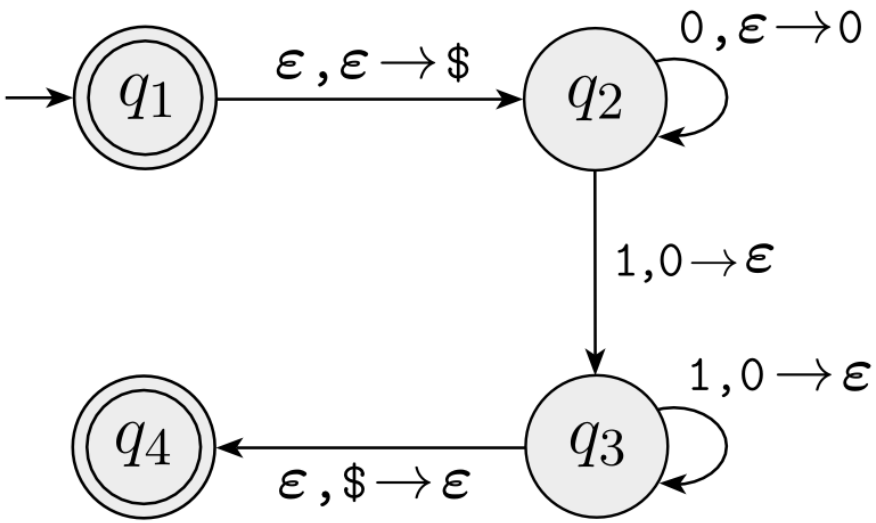


Figure 2.15, Sipser 2012

PDA FOR $L = \{0^n 1^n : n \geq 0\}$

Formal description of PDA recognizing L: it is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_1, F)$ as follows.

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, \$\}$
- the transition function δ is given as the transition table below.
- Start state is q_1
- $F = \{q_1, q_4\}$

Input:	0			1			ϵ		
Stack:	0	\$	ϵ	0	\$	ϵ	0	\$	ϵ
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \epsilon)\}$					
q_3				$\{(q_3, \epsilon)\}$				$\{(q_4, \epsilon)\}$	
q_4									

From example 2.14, Sipser 2012

PDA FOR $L = \{w \cdot w^R : w \in \{0,1\}^*\}$

PDA FOR $L = \{w \in 0^n 1^m : n \geq m\}$

PDA FOR $L = \{w \in \{a, b\}^* : |w|_a = |w|_b\}$

PDA FOR

$$L = \{w \in a^i b^j c^k : i = j \textbf{ OR } i = k\}$$

Lec 11. Pushdown Automata and CFG

Eunjung Kim

PDA FOR $L = \{w \cdot w^R : w \in \{0,1\}^*\}$

PDA FOR $L = \{w \in 0^n 1^m : n \geq m\}$

PDA FOR $L = \{w \in \{a, b\}^* : |w|_a = |w|_b\}$

PDA FOR

$L = \{w \in a^i b^j c^k : i = j \textbf{ OR } i = k\}$

PDA, SEEMINGLY MORE POWERFUL

A slightly general form of PDA which pops a string in Γ^* and pushes Γ^* can be converted into a usual one.

PDA WITH SPECIFIC CONDITIONS

A given PDA can be transformed to satisfy any combination of the following conditions.

1. It has a single accept state q_{accept}.

2. It empties its stack before accepting.

3. Each transition move either pushes a symbol onto the stack (push move) or pops a symbol off the stack (pop move), but does not do both at the same time.

Equivalence of CFG and PDA

Theorem

A language is context-free if and only if some pushdown automaton recognizes it.

- ($\Rightarrow$): converting a CFG to an equivalent PDA.
- ($\Leftarrow$): converting a PDA to an equivalent CFG.

$\Rightarrow$: Converting CFG to PDA

- From a context-free grammar $G = (V, \Sigma, R, S)$, we aim to construct a PDA P such that $L(P) = L(G)$.

$\Rightarrow$: Converting CFG to PDA

- From a context-free grammar $G = (V, \Sigma, R, S)$, we aim to construct a PDA P such that $L(P) = L(G)$.

- Key idea: we design PDA P which simulates a leftmost derivation of w.

$\Rightarrow$: Converting CFG to PDA

- From a context-free grammar $G = (V, \Sigma, R, S)$, we aim to construct a PDA P such that $L(P) = L(G)$.

- Key idea: we design PDA P which simulates a leftmost derivation of w.
  1. "matching" the input symbol and the stack symbol if the stack symbol is an element of Σ.
  2. "replacing" the stack symbol A by z if A is a variable of G and there is a rule $A \to z$.
  3. while maintaining, in the stack, the suffix of a string $w \in (\Sigma \cup V)^*$ s.t. $S \Rightarrow_{lm}^* w$ starting with the leftmost variable in w.

$\Rightarrow$: Converting CFG to PDA

$L = \{0^n 1^n : n \geq 0\}$ is the language of the grammar $S \to 0S1 \quad | \quad \epsilon$.

$\Rightarrow$: Converting CFG to PDA

Construct a PDA P as follows.

1. There are three states q_{start}, q, q_{accept}.
2. The stack alphabet is $V \cup \Sigma \cup \{\$\}$.
3. Initially, P places the marker \$ onto the (empty) stack, then the start symbol S of CFG G.
4. It loops at the state q and executes the following unless the stack symbol is \$
   - If the stack symbol is $A \in V$, then P nondeterministically chooses a rule of the form $A \to \gamma$ and pushes γ onto stack so that the first symbol of γ is at the top.
   - If the stack symbol is $a \in \Sigma$, then P reads the symbol $a \in \Sigma$ in the input, pop a, and stays in the current state. If a cannot be read ("does not match"), no move is defined and the current computation branch dies out.
5. If the stack symbol is \$, then it goes to q_{accept}. The input string is accepted if the string has been read fully. If not, the current branch dies out.

$\Rightarrow$: Converting CFG to PDA

Construct a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ from $G = (V, \Sigma, R, S)$:

1. $Q = \{q_0, q, q_{accept}\}$. $\Gamma = V \cup \Sigma \cup \{\$\}$.
2. $\delta(q_0, \epsilon, \epsilon) = \{(q, S\$)\}$.
3. For each stack symbol in $V \cup \Sigma \cup \{\$\}$
   - for every $A \in V$: $\delta(q, \epsilon, A) = \{(q, \gamma) :$ for all rules $A \to \gamma$ in $G\}$
   - for every $a \in \Sigma$: $\delta(q, a, a) = \{(q, \epsilon)\}$
   - $\delta(q, \epsilon, \$) = \{(q_{accept}, \epsilon)\}$.

$\Rightarrow$: Converting CFG to PDA

Construct a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ from $G = (V, \Sigma, R, S)$:

1. $Q = \{q_0, q, q_{accept}\}$. $\Gamma = V \cup \Sigma \cup \{\$\}$.
2. $\delta(q_0, \epsilon, \epsilon) = \{(q, S\$)\}$.
3. For each stack symbol in $V \cup \Sigma \cup \{\$\}$
   - for every $A \in V$: $\delta(q, \epsilon, A) = \{(q, \gamma) :$ for all rules $A \to \gamma$ in $G\}$
   - for every $a \in \Sigma$: $\delta(q, a, a) = \{(q, \epsilon)\}$
   - $\delta(q, \epsilon, \$) = \{(q_{accept}, \epsilon)\}$.

How to implement a transition such as $\{(q, \gamma) \in \delta(q, \epsilon, A)$ when γ is a string, not necessarily a symbol in Γ_ϵ?

$\Leftarrow$: Converting PDA to CFG

Step A. Streamlining the PDA.

1. It has a single accept state q_{accept}.
2. It empties its stack before accepting.
3. Each transition move either pushes a symbol onto the stack (push move) or pops a symbol off the stack (pop move), but does not do both at the same time.

$\Leftarrow$: Converting PDA to CFG

Step B. Variables A_{pq} for all $p, q \in Q$.

1. Meaning of A_{pq}: we intend to design CFG G so that

$$L(A_{pq}) := \{w : A \Rightarrow_G^* w\}$$

coincides with

$$\{w : (p, w, \epsilon) \vdash_P^* (q, \epsilon, \epsilon)\}$$

2. Take A_{st} as the start variable of CFG G, where $s = q_0$ and $t = q_{accept}$.
3. Then $L(G)(= L(A_{st}))$ coincides with

$$\{w : (q_0, w, \epsilon) \vdash_P^* (q_{accept}, \epsilon, \epsilon)\},$$

which is precisely $L(P)$.

$\Leftarrow$: Converting PDA to CFG

Step C. Designing a production rule for the variable A_{pq}.

1. For a string w in

$$\{w : (p, w, \epsilon) \vdash_P^* (q, \epsilon, \epsilon)\},$$

two situation can occur when P runs on w.

   A. the stack gets empty while running
   B. the symbol pushed at the beginning is never popped till the last moment.

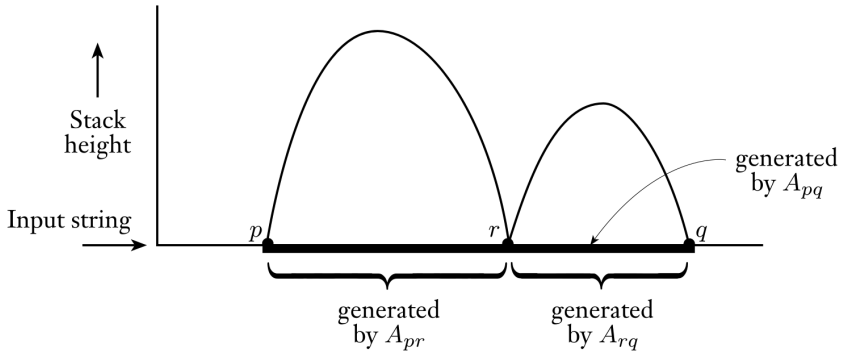


Figure 2.28, Sipser 2012

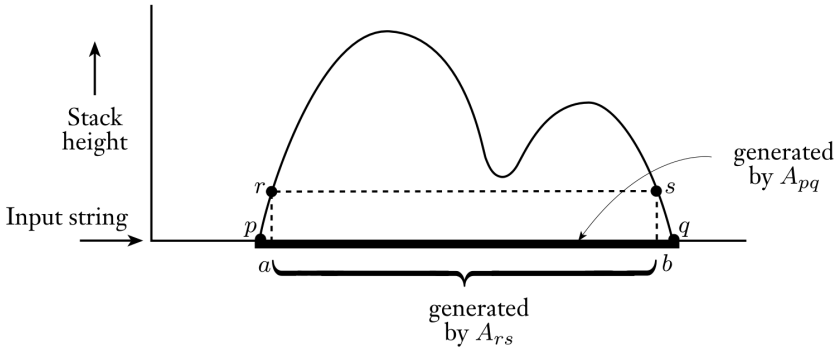


Figure 2.29, Sipser 2012

$\Leftarrow$: Converting PDA to CFG

Step C. Designing a production rule for the variable A_{pq}.

1. For a string w in
$$\{w : (p, w, \epsilon) \vdash_P^* (q, \epsilon, \epsilon)\},$$
two situation can occur when P runs on w.
   - A. the stack gets empty while running:
     i.e. $w \in L(A_{pr}) \cdot L(A_{rq})$.
   - B. the symbol pushed at the beginning is never popped till the last moment.
     i.e. $w \in aL(rs)b$ for all $a, b \in \Sigma$ whenever $\delta(p, a, \epsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ϵ) for some $u \in \Gamma$.
2. The trivial case $(p, \epsilon, \epsilon) \vdash_P^* (p, \epsilon, \epsilon)$.

Each case is simulated by the next rules.

- case A: $A_{pq} \to A_{pr}A_{rq}$ for all $p, q, r \in Q$
- case B: $A_{pq} \to aA_{rs}b$ for all $p, q, r, s \in Q$ and $a, b \in \Sigma_\epsilon$, and $u \in \Gamma$ such that $\delta(p, a, \epsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, ϵ).
- case C: $A_{pp} \to \epsilon$.

The new CFG G contains all the above rules.

$\Leftarrow$: Converting PDA to CFG

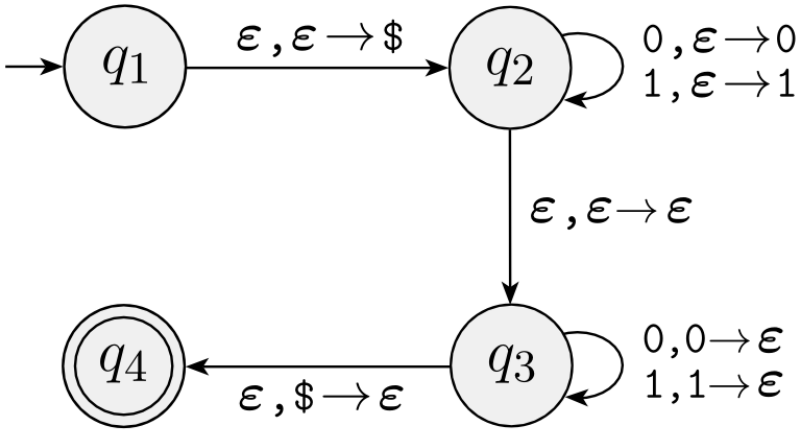


Figure 2.19, Sipser 2012

Why the conversions produce equivalent PDA / CFG?

(A quick words, which you can turn into a correctness proof.)

$\Rightarrow$: from CFG to PDA

- As an invariant, at each step of PDA's run on w,
  (the prefix of w that P already read) $\circ$ (the string in the stack, save \$) ($\star$)
  forms a leftmost derivation from S, implying $L(P) \subseteq L(G)$.
- For any $w \in (\Sigma \cup V)^*$ with $S \Rightarrow_{lm}^* w$, there is a run of P ending in a configuration with ($\star$). Especially, there is a run which ends up with an empty stack (save \$) after having read all symbols in the input, implying $L(G) \subseteq L(P)$.
- Use induction to argue both.

Why the conversions produce equivalent PDA / CFG?

(A quick words, which you can turn into a correctness proof.)

$\Leftarrow$: from PDA to CFG

- For both $L(P) \subseteq L(G)$ and $L(G) \subseteq L(P)$, Tedious induction...