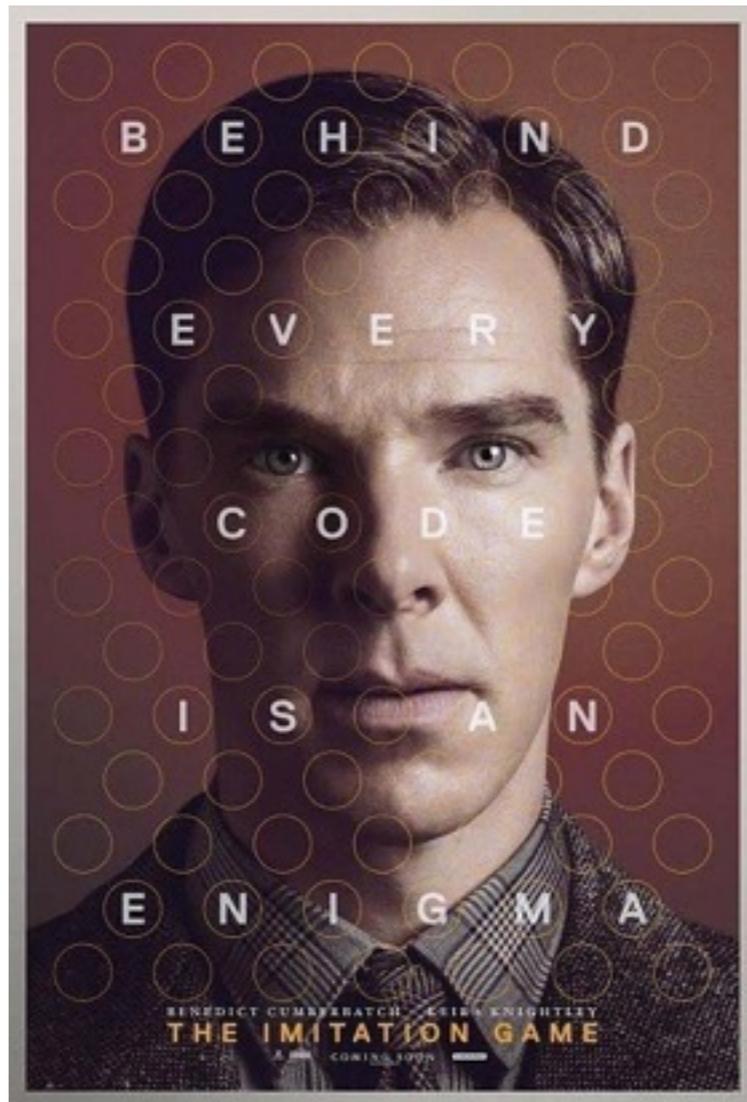


Automata & languages

A primer on the Theory of Computation



Laurent Vanbever

www.vanbever.eu

ETH Zürich (D-ITET)

October, 12 2017

Part 4 out of 5

Last week, we showed the
equivalence of DFA, NFA and REX

is equivalent to



We also looked at nonregular languages

Pumping lemma

If A is a regular language, then there exist a number p s.t.

Any string in A whose length is at least p can be divided into three pieces xyz s.t.

- $xy^iz \in A$, for each $i \geq 0$ and
- $|y| > 0$ and
- $|xy| \leq p$

To prove that a language A is not regular:

- 1 Assume that A is regular
- 2 Since A is regular, it must have a pumping length p
- 3 Find one string s in A whose length is at least p
- 4 For any split $s=xyz$,
Show that you can not satisfy all three conditions
- 5 Conclude that s cannot be pumped

To prove that a language A is not regular:

- 1 Assume that A is regular
- 2 Since A is regular, it must have a pumping length p
- 3 Find one string s in A whose length is at least p
- 4 For any split $s=xyz$,
Show that you can not satisfy all three conditions
- 5 Conclude that **s cannot be pumped $\longrightarrow A$ is not regular**

This week is all about

Context-Free Languages

a superset of Regular Languages

CFG's: Proving Correctness

- The recursive nature of CFG's means that they are especially amenable to correctness proofs.

- For example let's consider again our grammar

$$G = (S \rightarrow \varepsilon \mid ab \mid ba \mid aSb \mid bSa \mid SS)$$

- We claim that $L(G) = L = \{x \in \{a,b\}^* \mid n_a(x) = n_b(x)\}$,
where $n_a(x)$ is the number of a 's in x , and $n_b(x)$ is the number of b 's.

- *Proof:* To prove that $L = L(G)$ is to show both inclusions:
 - i.* $L \subseteq L(G)$: Every string in L can be generated by G .
 - ii.* $L \supseteq L(G)$: G only generate strings of L .

CFG's: Proving Correctness

- The recursive nature of CFG's means that they are especially amenable to correctness proofs.

- For example let's consider again our grammar

$$G = (S \rightarrow \varepsilon \mid ab \mid ba \mid aSb \mid bSa \mid SS)$$

- We claim that $L(G) = L = \{x \in \{a,b\}^* \mid n_a(x) = n_b(x)\}$, where $n_a(x)$ is the number of a 's in x , and $n_b(x)$ is the number of b 's.

- *Proof:* To prove that $L = L(G)$ is to show both inclusions:

i. $L \subseteq L(G)$: Every string in L can be generated by G .

ii. $L \supseteq L(G)$: G only generate strings of L .

Part *ii.* is easy (see why?), so we'll concentrate on part *i.*

Proving $L \subseteq L(G)$

- $L \subseteq L(G)$: Show that every string x with the same number of a 's as b 's is generated by G . Prove by induction on the length $n = |x|$.
- **Base case:** The empty string is derived by $S \rightarrow \varepsilon$
- **Inductive hypothesis:**
Assume that G generates all strings of equal number of a 's and b 's of (even) length up to n .

Consider any string of length $n+2$. There are essentially 4 possibilities:

1. awb
2. bwa
3. awa
4. bwb

Proving $L \subseteq L(G)$

- Inductive hypothesis:

Consider any string of length $n+2$. There are essentially 4 possibilities:

1. awb
2. bwa
3. awa
4. bwb

Given $S \Rightarrow^* w$, awb and bwa are generated from w using the rules $S \rightarrow aSb$ and $S \rightarrow bSa$ (induction hypothesis)

Proving $L \subseteq L(G)$

- Inductive hypothesis:

Now, consider a string like awa . For it to be in L requires that w isn't in L as w needs to have 2 more b 's than a 's.

- Split awa as follows: ${}_0a_1 \dots {}_{-1}a_0$
where the subscripts after a prefix v of awa denotes $n_a(v) - n_b(v)$
- Think of this as counting starting from 0.
Each a adds 1. Each b subtracts 1. At the end, we should be at 0.

Somewhere along the string (in w), the counter crosses 0 (more b 's)

Proving $L \subseteq L(G)$

- Inductive hypothesis:

Somewhere along the string (in w), the counter crosses 0:

$$\begin{array}{c} \xleftrightarrow{u} \\ {}_0 a_1 \dots {}_{-1} x_0 y \dots {}_{-1} a_0 \text{ with } x, y \in \{a, b\} \\ \xleftrightarrow{v} \end{array}$$

- u and v have an equal number of a 's and b 's and are shorter than n .
- Given $S \Rightarrow^* u$ and $S \Rightarrow^* v$, the rule $S \rightarrow SS$ generates $awa = uv$ (induction hypothesis)
- The same argument applies for strings like bwb

Designing Context-Free Grammars

- As for regular languages this is a **creative process**.
- However, if the grammar is the union of simpler grammars, you can design the simpler grammars (with starting symbols S_1, S_2), and then add a new starting symbol/production $S \rightarrow S_1 \mid S_2$.

Designing Context-Free Grammars

- As for regular languages this is a **creative process**.
- However, if the grammar is the union of simpler grammars, you can design the simpler grammars (with starting symbols S_1, S_2), and then add a new starting symbol/production $S \rightarrow S_1 \mid S_2$.
- If the CFG happens to be regular, you can first design the FA, introduce a variable/production for each state of the FA, and then add a rule $x \rightarrow ay$ to the CFG if $\delta(x,a) = y$ is in the FA. If a state x is accepting in FA then add $x \rightarrow \varepsilon$ to CFG. The start symbol of the CFG is the start state in the FA.
- There are quite a few other tricks. Try yourself...

Push-Down Automata (PDA)

- Finite automata where the machine interpretation of regular languages.
- **Push-Down Automaton** are the machine interpretation for grammars.
- The problem of finite automata was that they couldn't handle languages that needed some sort of unbounded memory... something that could be implemented easily by a single (unbounded) integer **register**!
- Example: To recognize the language $L = \{0^n 1^n \mid n \geq 0\}$, all you need is to count how many 0's you have seen so far...
- Push-Down Automata allow even more than a register: a full **stack**!

Recursive Algorithms and Stacks

- A stack allows the following basic operations
 - **Push**, pushing a new element on the top of the stack.
 - **Pop**, removing the top element from the stack (if there is one).
 - **Peek**, checking the top element without removing it.
- General Principle in Programming:
Any recursive algorithm can be turned into a non-recursive one using a stack and a while-loop which exits only when stack is empty.

Recursive Algorithms and Stacks

- A stack allows the following basic operations
 - **Push**, pushing a new element on the top of the stack.
 - **Pop**, removing the top element from the stack (if there is one).
 - **Peek**, checking the top element without removing it.
- General Principle in Programming:
Any recursive algorithm can be turned into a non-recursive one using a stack and a while-loop which exits only when stack is empty.
- It seems that with a stack at our fingertips we can even recognize **palindromes**! Yoo-hoo!
 - Palindromes are generated by the grammar $S \rightarrow \varepsilon \mid aSa \mid bSb$.
 - Let's simplify for the moment and look at $S \rightarrow \# \mid aSa \mid bSb$.

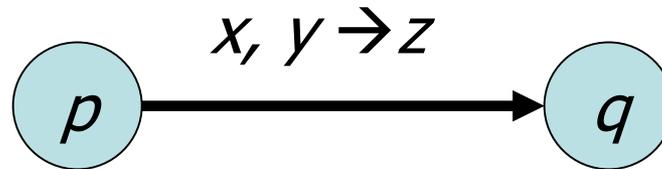
From CFG's to Stack Machines

- The CFG $S \rightarrow \# \mid aSa \mid bSb$ describes palindromes containing exactly 1 #.
- Question: Using a stack, how can we recognize such strings?

PDA's à la Sipser

- To aid analysis, theoretical stack machines restrict the allowable operations. Each text-book author has his/her own version.
- Sipser's machines are especially simple:
 - Push/Pop rolled into a single operation: **replace top stack symbol**.
 - In particular, replacing top by ϵ is a pop.
- No intrinsic way to test for empty stack.
 - Instead often push a special symbol (“\$”) as the very first operation!
- Epsilon's used to increase functionality
 - result in default **nondeterministic** machines.

Sipser's PDA Version



If at state p and next input is x and top stack is y , then go to state q and replace y by z on stack.

- $x = \varepsilon$: ignore input, don't read
- $y = \varepsilon$: ignore top of stack and push z
- $z = \varepsilon$: pop y

In addition, push “\$” initially to detect the empty stack.

PDA: Formal Definition

- Definition: A **pushdown automaton** (PDA) is a 6-tuple

$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$:

- Q, Σ , and q_0 , and F are defined as for an FA.
- Γ is the **stack alphabet**.
- δ is as follows:

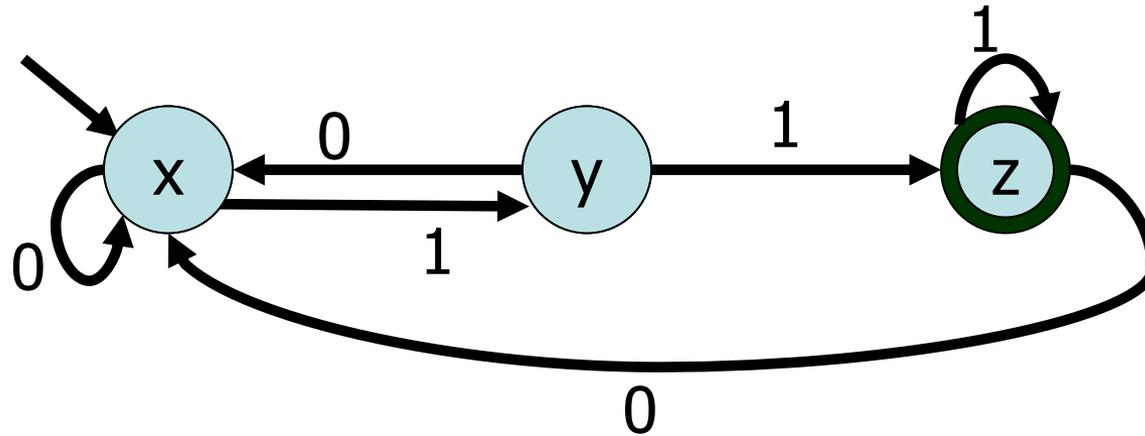
Given a state p , an input symbol x and a stack symbol y , $\delta(p, x, y)$ returns all (q, z) where q is a target state and z a stack replacement for y .

$$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$$

Model Robustness

- The class of regular languages was quite **robust**
 - Allows multiple ways for defining languages (automaton vs. regexp)
 - Slight perturbations of model do not change result (non-determinism)
- The class of context free languages is also robust:
you can use either PDA's or CFG's to describe the languages in the class.
- However, it is less robust than regular languages when it comes to slight perturbations of the model:
 - Smaller classes
 - Right-linear grammars
 - Deterministic PDA's
 - Larger classes
 - Context Sensitive Grammars

Right Linear Grammars vs. Regular Languages



- The DFA above can be simulated by the grammar
 - $x \rightarrow 0x \mid 1y$
 - $y \rightarrow 0x \mid 1z$
 - $z \rightarrow 0x \mid 1z \mid \varepsilon$
- Definition: A **right-linear grammar** is a CFG such that every production is of the form $A \rightarrow uB$, or $A \rightarrow u$ where u is a terminal string, and A, B are variables.

Right Linear Grammars vs. Regular Languages

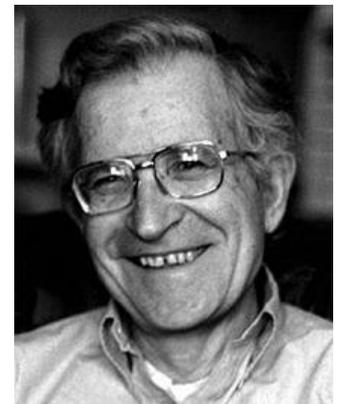
- Theorem: If $M = (Q, \Sigma, \delta, q_0, F)$ is an NFA then there is a right-linear grammar $G(M)$ which generates the same language as M .
- *Proof:*
 - Variables are the states: $V = Q$
 - Start symbol is start state: $S = q_0$
 - Same alphabet of terminals Σ
 - A transition $q_1 \xrightarrow{a} q_2$ becomes the production $q_1 \rightarrow aq_2$
 - For each transition, $q_1 \xrightarrow{a} q_2$ where q_2 is an accept state, add $q_1 \rightarrow a$ to the grammar
- Homework: Show that the reverse holds. Right-linear grammar can be converted to a FSA. This implies that $RL \approx$ Right-linear CFL.

Right Linear Grammars vs. Regular Languages

- Theorem: If $M = (Q, \Sigma, \delta, q_0, F)$ is an NFA then there is a right-linear grammar $G(M)$ which generates the same language as M .
- *Proof:*
 - Variables are the states: $V = Q$
 - Start symbol is start state: $S = q_0$
 - Same alphabet of terminals Σ
 - A transition $q_1 \xrightarrow{a} q_2$ becomes the production $q_1 \rightarrow aq_2$
 - For each transition, $q_1 \rightarrow aq_2$ where q_2 is an accept state, add $q_1 \rightarrow a$ to the grammar
- Homework: Show that the reverse holds. Right-linear grammar can be converted to a FSA. This implies that $RL \approx$ Right-linear CFL.
- Question: Can every CFG be converted into a right-linear grammar?

Chomsky Normal Form

- Chomsky came up with an especially simple type of context free grammars which is able to capture all context free languages, the **Chomsky normal form** (CNF).
- Chomsky's grammatical form is particularly useful when one wants to prove certain facts about context free languages. This is because assuming a much more restrictive kind of grammar can often make it easier to prove that the generated language has whatever property you are interested in.
- Noam Chomsky, linguist at MIT, creator of the Chomsky hierarchy, a classification of formal languages. Chomsky is also widely known for his left-wing political views and his criticism of the foreign policy of U.S. government.



Chomsky Normal Form

- Definition: A CFG is said to be in **Chomsky Normal Form** if every rule in the grammar has one of the following forms:
 - $S \rightarrow \varepsilon$ (ε for epsilon's sake only)
 - $A \rightarrow BC$ (dyadic variable productions)
 - $A \rightarrow a$ (unit terminal productions)

where S is the start variable, A, B, C are variables and a is a terminal.

- Thus epsilons may only appear on the right hand side of the start symbol and other rights are either 2 variables or a single terminal.

CFG \rightarrow CNF

- Converting a general grammar into Chomsky Normal Form works in four steps:
 1. Ensure that the **start** variable doesn't appear on the **right** hand side of any rule.
 2. Remove all **epsilon** productions, except from start variable.
 3. Remove unit variable productions of the form $A \rightarrow B$ where A and B are variables.
 4. Add variables and dyadic variable rules to replace any **longer** non-dyadic or non-variable productions

CFG \rightarrow CNF: Example

$$S \rightarrow \varepsilon | a | b | aSa | bSb$$

1. No start variable on right hand side

$$S' \rightarrow S$$

$$S \rightarrow \varepsilon | a | b | aSa | bSb$$

2. Only start state is allowed to have ε

$$S' \rightarrow S | \varepsilon$$

$$S \rightarrow \varepsilon | a | b | aSa | bSb | aa | bb$$

3. Remove unit variable productions of the form $A \rightarrow B$.

$$S' \rightarrow S | \varepsilon | a | b | aSa | bSb | aa | bb$$

$$S \rightarrow a | b | aSa | bSb | aa | bb$$

CFG \rightarrow CNF: Example continued

$$S' \rightarrow \mathcal{S}|\varepsilon|a|b|aSa|bSb|aa|bb$$

$$S \rightarrow a|b|aSa|bSb|aa|bb$$

4. Add variables and dyadic variable rules to replace any longer productions.

$$S' \rightarrow \varepsilon|a|b|aSa|bSb|aa|bb|AB|CD|AA|CC$$

$$S \rightarrow a|b|aSa|bSb|aa|bb|AB|CD|AA|CC$$

$$A \rightarrow a$$

$$B \rightarrow SA$$

$$C \rightarrow b$$

$$D \rightarrow SC$$

CFG \rightarrow PDA

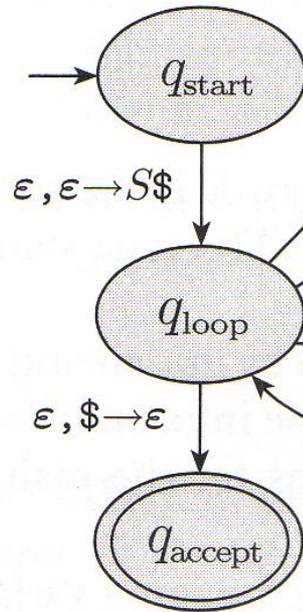
- CFG's can be converted into PDA's.
- In "NFA \rightarrow REX" it was useful to consider GNFA's as a middle stage. Similarly, it's useful to consider Generalized PDA's here.
- A **Generalized PDA** (GPDA) is like a PDA, except it allows the top stack symbol to be replaced by a whole string, not just a single character or the empty string. It is easy to convert a GPDA's back to PDA's by changing each compound push into a sequence of simple pushes.

CFG \rightarrow GPDA Recipe

1. Push the marker symbol $\$$ and the start symbol S on the stack.
2. Repeat the following steps forever
 - a. If the top of the stack is the variable symbol A , nondeterministically select a rule of A , and substitute A by the string on the right-hand-side of the rule.
 - b. If the top of the stack is a terminal symbol a , then read the next symbol from the input and compare it to a . If they match, continue. If they do not match reject this branch of the execution.
 - c. If the top of the stack is the symbol $\$$, enter the accept state.
(Note that if the input was not yet empty, the PDA will still reject this branch of the execution.)

CFG \rightarrow GPDA \rightarrow PDA: Example

- $S \rightarrow aTb \mid b$
- $T \rightarrow Ta \mid \epsilon$



CFG \rightarrow PDA: Now you try!

- Convert the grammar $S \rightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$

PDA \rightarrow CFG

- To convert PDA's to CFG's we'll need to simulate the stack inside the productions.
- Unfortunately, in contrast to our previous transitions, this is not quite as constructive. We will therefore only state the theorem.
- Theorem: For each push-down automation there is a context-free grammar which accepts the same language.
- Corollary: **PDA \approx CFG.**

Context Sensitive Grammars

- An even more general form of grammars exists.
In general, a non-context free grammar is one in which whole mixed variable/terminal substrings are replaced at a time.
For example with $\Sigma = \{a,b,c\}$ consider:

$$\begin{array}{ll} S \rightarrow \varepsilon \mid ASBC & aB \rightarrow ab \\ A \rightarrow a & bB \rightarrow bb \\ CB \rightarrow BC & bC \rightarrow bc \\ & cC \rightarrow cc \end{array}$$

What language is generated by this non-context-free grammar?

- When length of LHS always \leq length of RHS (plus some other minor restrictions), these general grammars are called **context sensitive**.

Are all languages context-free?

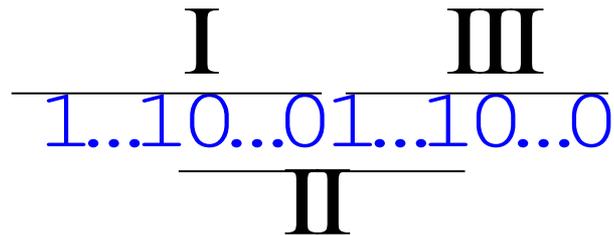
- Design a CFG (or PDA) for the following languages:
- $L = \{ w \in \{0,1,2\}^* \mid \text{there are } k \text{ 0's, } k \text{ 1's, and } k \text{ 2's for } k \geq 0 \}$
- $L = \{ w \in \{0,1,2\}^* \mid \text{with } |0| = |1| \text{ or } |0| = |2| \text{ or } |1| = |2| \}$
- $L = \{ 0^k 1^k 2^k \mid k \geq 0 \}$

Tandem Pumping

- Analogous to regular languages there is a pumping lemma for context free languages. The idea is that you can pump a context free language at **two** places (but not more).
- Theorem: Given a context free language L , there is a number p (**tandem-pumping number**) such that any string in L of length $\geq p$ is tandem-pumpable within a substring of length p . In particular, for all $w \in L$ with $|w| \geq p$ we can write:
 - $w = uvxyz$
 - $|vy| \geq 1$ (pumpable areas are non-empty)
 - $|vxy| \leq p$ (pumping inside length- p portion)
 - $uv^ixy^iz \in L$ for all $i \geq 0$ (tandem-pump v and y)
- If there is no such p the language is not context-free.

Proving Non-Context Freeness: Example

- $L = \{1^n 0^n 1^n 0^n \mid n \text{ is non-negative}\}$
- Let's try $w = 1^p 0^p 1^p 0^p$. Clearly $w \in L$ and $|w| \geq p$.
- With $|vxy| \leq p$, there are only three places where the “sliding window” vxy could be:



- In all three cases, pumping up such a case would only change the number of 0s and 1s in that part and not in the other two parts; this violates the language definition.

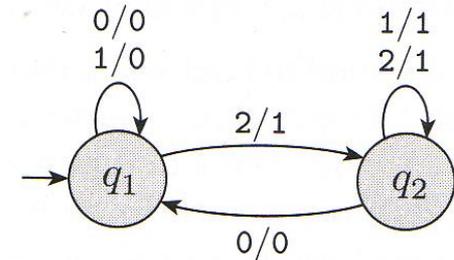
Proving Non-Context Freeness: You try!

- $L = \{ x=y+z \mid x, y, \text{ and } z \text{ are binary bit-strings satisfying the equation} \}$
- The hard part is to come up with a word which cannot be pumped, such as...

Transducers

- Definition: A finite state **transducer** (FST) is a type of finite automaton whose **output is a string** and not just accept or reject.
- Each transition of an FST is labeled with two symbols, one designating the input symbol for that transition (as for automata), and the other designating the output symbol.
 - We allow ε as output symbol if no symbol should be added to the string.

- The figure on the right shows an example of a FST operating on the input alphabet $\{0,1,2\}$ and the output alphabet $\{0,1\}$



- Exercise: Can you design a transducer that produces the inverted bit-string of the input string (e.g. $01001 \rightarrow 10110$)?