Università
degli Studi
di Padova

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

INFORMATION ENGINEERING DEPARTMENT

COMPUTER ENGINEERING - AI & ROBOTICS

AUTOMATA, LANGUAGES AND COMPUTATION 2022/2023

Francesco Crisci 2076739

Lezione 1

We want to setup a mathematical study of computation, we'll ask the following questions:

- What can be computed?
- What can be efficiently computed? (tractability)

The study of computation requires automata theory (abstract models of machine) and formal language theory (abstract representation of data). When we talk about computer we have to talk about data and how to abstract them. Automata provides abstraction of computer, where languages provides abstraction of data.

Famous model of computation

The most famous models of computations are the following:

- Turing machine, which introduced the study of computability;
- Fina automata, FA, which in the 50s introduced models of neuronal calculus (it has othing to do with neural networks);
- Formal grammar, introduced by Noam Chomsky as a linguistic model. This guy was the Isaac Newton of linguistic, the first one to say it was a mathematical problem.

Fine automata are mathematical objects, they are used nowadays for digital circuit design, lexical analyazers...
The simplest representation of a FA is a graph, with nodes, arcs and labels:
What is a state? In some computation, how far did you go? YOu want to know where you are, like

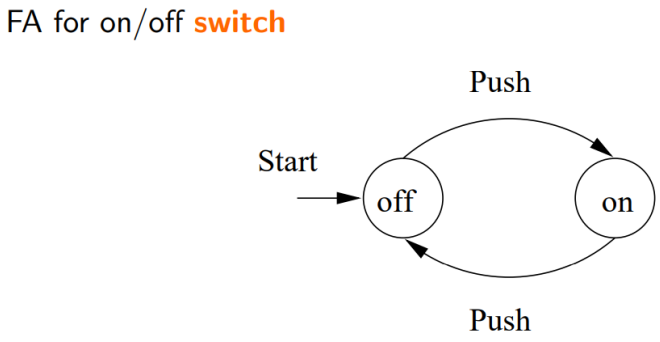


Figure 1: Finite automata for an On/Off switch

reconizing which letter are you on.

Each node represent the specific state of affair, the transition is the next step of the computation, like what you are looking right now for example.
There is an important distinction between two family of Structural Representation:

- Recognition models: it takes an input sequence(string) and either accepts or rejects it. It is an operational model.
- Generative models: it generates all of the desired sequences. It has no input. It's a declarative model.

We are interested in arithmetic expressions. You need to describe when an arithmetic expression is well written or no. You would use a generative model.
You can divide an expression in two more simple expression. If you have two expressions well formed and you put a + in the midddle, you'll get a new well formed expression. You usually read expressions right to left: $E \leftarrow E + E$.
You can also have regular expressions which can generate a string, for example.
In the slides example, both are from generative models.

Proofs

Typical form of the statement to be proved. Given H and C as properties: IF H, THEN C, also written as $H \implies C$, where H is the hypothesis and C the conclusion. This means that:

- H is a sufficient condition for C
- C is a necessary condition fo H

$H \implies C$ is equivalent to $H \subseteq C$: if H is true, C can't be false. If H implies C, it doesn't mean that C implies H.

Deduction

Sequence of statements that starts from one or more hypothesis and leads to a conclusion. Each step in the sequence uses some logial rule, applying it to hypothesis or to one of the previous obtained statements.
Modus ponens: logical rule to move from one statement to the next. If we know that "if H then C" is true, and if we know that H is true, then we can conclude that C is true.
Example: if x is the sum of the squared of four positive integers, then $2^x \geq x^2$
Theorems having the form: C_1 if and only if C_2 require proofs for both directions. We can use also techniques like Reduction to definitions (convert all terms in the assumptions using the corresponding definitions) and Proof by contraddction (instead of prooving If H then C, you can proove that H and not C is falsehood).
We can also use a counterexample: to prove that a theorem is false, it is enough to show a case in which the statement is false.

Quantifiers

- For each x ($\forall x$): applies to all values of the variable
- Exists x ($\exists x$): applies to at least one value of the variable.

The ordering of the quantifiers affects the meanig of the statement.

Set equality

If E and F are sets, to prove that E = F we have to prove that $E \subseteq F$ and $F \subseteq E$.

Contrapositive

The statement if H then C is equivalent to the statement if C is false, then H is false.

Induction

It's the main technique when working in recursivelly defined objects. You have a base case and a property to prove. Afterwards, Through the inductive step you can prove the whole statement.

- In the base case we show that S(i) for some specific integer i, usully 0 or 1, is true.
- In the inductive step, for n greater or equal than i, prove statement "if S(n) then S(n+1)"."

Structural Induction

We use it for objects that are not numbers, like arithmetic expressions:

- Base: any variable or number is an arithmetic expression.
- Induction: If E and F are arithmetic expressions, then also E + F, ExF, and (E) are arithmetic expressions. This objects are more complicated than just a single number. I cannot define a total order of the expressions.

You can think expressions are ordered in this way, like a tree (FIG 1.2). You'll have to prove all the

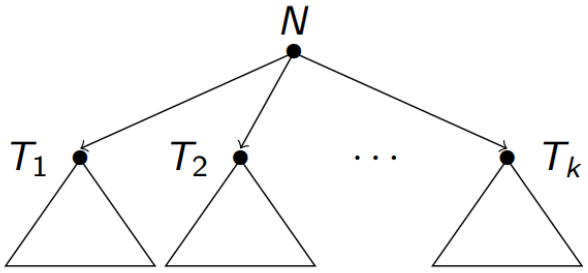


Figure 2: Structural Induction's tree

nodes, like leaves and such. You hav eto prove every sigle rule.

Mutual induction

Sometimes is not possible to prove a statement $S_1(n)$ by induction, because the statement depends in statements $S_2(n), ..., S_k(n)$ of different types. We need to prove jointly a family of statements $S_1(n), ..., S_k(n)$ by mutual induction for n.

Lezione 2

Finite Objects

Alphabet

It'a a finite and nonempty set of atomic symbols, for examples: $\{0,1\}, \{a,b,c,d,e,f,.....,z\}$ or the set of all printable ASCII characters.

String

Is a finite sequence of symbols from some alphabet. Since infinite objects cannot be put as a n input to the computer, it's important to define finite objects.
There is a special string, the empty one, which is usually rappresented by ϵ, which has length zero an it contains zero symbols. The length of the string is the number of occurences for the symbols of the string. You can have multiple strings with the same length but there is always gonna be just one with length zero.

Power of an alphabet

Given k as a natural number, we can define Σ^k to represent all set of the alphabet that have length k. There is ambiguity between $\Sigma and \Sigma^1$ cause Σ is an alphabet and Σ^1 are all the strings with length 1. let's see some examples:

- $\Sigma^2 = \{00, 01, 10, 11\}$ if $\Sigma = \{0,1\}$
- $\Sigma^0 = \{\epsilon\}$

The set of all strings form Σ is denoted Σ^*, so that:
$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup ....$
$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup ....$
$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

Concatenation

If x and y are strings, then xy is the string obtained by putting a copy of y immediately after a copy of x, for example:
x = 01101 and y = 110 $\rightarrow$ x $\cdot$ y = 01101110
ϵ is the neutral element of the concatenation. It can occurs any number of times within a string:
$x \cdot y = x \cdot \epsilon \cdot y$
$x \cdot y = x \cdot \epsilon \cdot \epsilon \cdot \epsilon \cdot y$

Conventions

Lower case symbols from the beginning of the alphabet are used to denote symbols of the alphabet {a,b,c,d,e,f...}
Lower cases symbols from the end of the laphabet are used to denote strings.
for n ≥ 0, a^n = aaa.....a (a repeated n times) with n $\in \mathbb{N}$
$a^0 = \epsilon, a^1 = a$

Language

Let's now put string into a set, so that we can obtain a language, which can be infinite (this is the major difference with string/alphabet).
$L \subseteq \Sigma^*$ is a language.
A language that has no strings is represented by $\emptyset$. It's important to not confuse the empty language with the empty string, cause a language that contatins the empty string is not an empty language: $\epsilon \neq \emptyset$.
We can have two types of notations for languages:

- Extensive representation: L = {ϵ, 01, 0011,000111,.....} or L = {1,2,3}
- Intensive representation: it uses a set former: L = { w |$statementspecifing$}.

We can see as an example the language which represent a string containing the same number of 0's and 1's.
L = { w | w = $0^n1^n, n \geq 0$ }, which can be astracted even more in the form L = $\{0^n1^n, n \geq 0\}$, which is valid for every n due to the fact the quantifier is implicit. Those two notations are equivalent to L = {ϵ, 01, 0011, 000111, ... }
It's important to point out that the and operator is implicit, where or is not:
$\{0^n1^n | n \geq 0\} \neq \{0^n1^n\}, n \geq 0$ since the first one is infinite and the second one is a language with just one string. The first language contains strings with same number of zeros and ones.
The first chapter ends here

Finite Automata (FA)

It's the simplest model of computation we'll look at. They read input strictly form left to right. If the string ends, the job is complete. The important notion state are infromation about what happened so far in the string you've read. The memory contains the information you are using, so you'll have a finite number of states.
Let's define a deterministic finite automata, or DFA, which is made of 5-tuples:

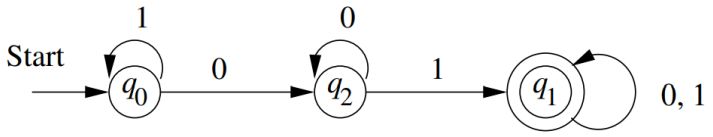
$$A = (Q, \Sigma, \delta, q_0, F)$$

Q is a finite set of states, somethijg like the gear you are at while driving.
Σ is the input symbol alphabet. δ is a transition function such that Q x $\Sigma \rightarrow$ Q.
$q_0 \in Q$ is the initial state.
$F \subseteq Q$ is a set of final states, or better the answers tot the computation, like yes or no.
Only at the end of the string you have to check if you have reached a finale state or not.
Q x Σ, state and symbol, will give you a new state Q.
The term deterministic derives from the fact that σ is a function: given an input there is gonna be an

answer, a unique response. There are no left-or.rigth decisions.
Example: Let's see a DFA that accepts the language of all strings of 0 and 1 containing the substring 01:

$$\mathrm{L} = \{x01y | x, y \in \{0,1\}^*\}$$

$Q = \{q_0, q_1, q_2\}, \Sigma = \{0,1\}, F = \{q_1\}$
Let's write the δ function, specified by means of a transition table, or better, let's look at the transition diagram:
I want to find the sequence 01 in the string.



	0	1
$\rightarrow q_0$	q_2	q_0
$\star q_1$	q_1	q_1
q_2	q_2	q_1

Figure 3: Transition table of the automata on top

In the table of transition, the final states are marked with a * and the inistial state with a $\rightarrow$.

Lezione 3

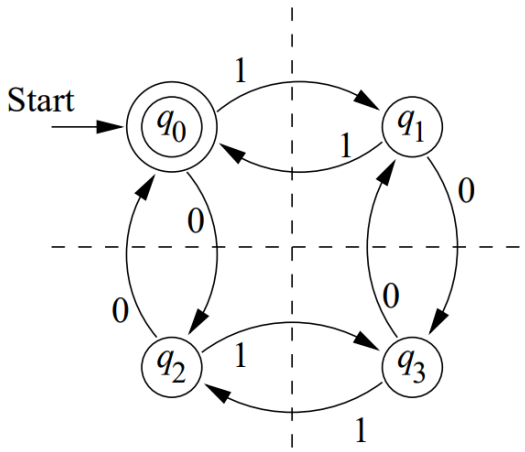
Acceptance

A DFA accepts a string w = $a_1a_2.....a_n$ if ther is a path in the transition diagram that:

- starts in the initial state
- ends in some final state
- has a sequence of transitions with labels $a_1a_2....a_n$

The definition uo here is not a mathematical defiition.
The δ transiction function can be extended to function $\hat{\delta}$ defined over state and string pairs.
Base : $\hat{\delta}(q, \epsilon) = q$, it's a string of length zero so there are no steps to take.
Induction: $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$ which represents one or more steps. a is a symbol of the alphabet, where x is all the symbols before the last one.
The number of occurences of a specific symbol in a string w, from the alphabet Σ is written $\#_a(w)$. It specifies a DFA A accepting all and only the strings in the followig language:

$$L = \{w | w \in \{0,1\}^*, \#_0(w)even, \#_1(w)even\}$$



This automata cannot be represented with 3 states or 2 cause we have 4 different scenarios.
Is w = 1010 accepted by A?

- $\hat{\delta}(q_0, \epsilon) = q_0$
- $\hat{\delta}(q_0, 0) = \delta(\hat{\delta}(q_0, \epsilon), 0) = \delta(q_0, 0) = q_2$
- $\hat{\delta}(q_0, 01) = \delta(\hat{\delta}(q_0, 0), 1) = \delta(q_2, 1) = q_3$

- $\hat{\delta}(q_0, 010) = \delta(\hat{\delta}(q_0, 01), 0) = \delta(q_3, 0) = q_1$
- $\hat{\delta}(q_0, 0101) = \delta(\hat{\delta}(q_0, 010), 1) = \delta(q_1, 1) = q_0 \in F$

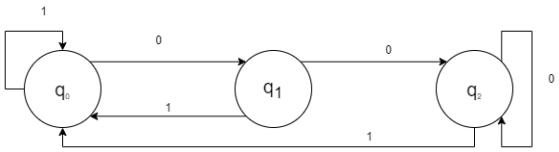
The machine has no way to know if we are at the end, ther eis an outside process from the δ function.

Regular languages

The languages recognized by DFA A is:

$$L(A) = \{w|\hat{\delta}(q_0, w) \in F\}$$

they are called regular languages. Usually p,q,r,s, $q_0, q_1, ...$ are used to represent states.
Excercise:
$L = \{w|\hat{\delta}(q_0, x00) \in F\}$, sinc ewe have no ϵ, so q_0 cannot be final state.



Nondeterministic Finite Automata (NFA)

They accept only regular languages, easier implementation than DFAs. Useful to search patterns in a text. Can be simultaneously be different states. Accepts if at least one final state is reached at the end of the scan of the input string. It can guess which the next state will lead to acceptance.
Existential: we check if there is a final state in the end.

$$A = (Q, \Sigma, \delta, q_0, F)$$

Q is a finite set of states, somethijg like the gear you are at while driving.
Σ is the input symbol alphabet. δ is a transition function such that Q x $\Sigma \to 2^Q$, where 2^Q is a set of all subset of Q
$q_0 \in Q$ is the initial state.
$F \subseteq Q$ is a set of final states, or better the answers tot the computation, like yes or no.

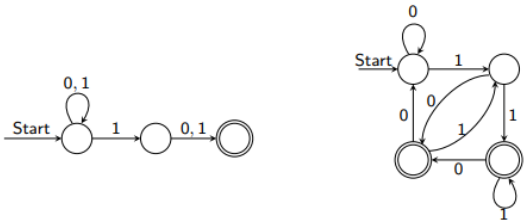
Lezione 4

$\hat{\sigma}_n$ function of NFAs

Base: $\hat{\delta}(q, \epsilon) = \{q\}$
Induction: $\hat{\delta}(q, xa) = \cup_{p \in \hat{\delta}(q,x)} \delta(p, a)$
As you can see, we have to go through all states connected to the one we are examining, it's different than from a DFAs. When you read x, you are in a set of states and the σ function wants just one state. Teh computation is really similar to the DFA's one, with the difference that you might find yourself in to diferent states a t the same time.
The accepted language for an NFA A is the following:

$$L(A) = \{w | \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

we have a set of strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ contains at least a final state.
An NFA is easier to project than a DFA since they can simplify the structure of the problem. For example, in the following image, we can see a DFA and a DFA accepting strings in $\{0, 1\}^*$ with penultimate symbol 1. The 2 models have the same omputational power.



You can always convert a NFA into a DFA such that L(A) = L(D).
Build a state oin D for every state set representing a "configuration" in a computation of N. The collection of all configurations is still a finite set.
Given an NFA N = $(Q_N \Sigma, \delta_N, q_0, F_N)$, the subset construction produces a DFA D = $(Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that L(D) = L(N).

Subset construction

$Q(D) = \{S | S \subseteq Q_N\}$, $F(D) = \{S \subseteq Q_N | S \cap F_N \neq \emptyset\}$, for every $S \subseteq Q_N$ and $a \in |Sigma$:

$$\delta_D(S, a) = \cup_{p \in S} \delta_N(p, a)$$

where $|Q_D| = 2 * |Q_N|$
To avoid an exponential growth we can use the lazy evaluation to get a DFA. We do this cause a lot of states are garbage and they cannot be achieved by any string.

Lezione 5

Partial DFA

It has at most one outgoing transition for each state in Q and for eact symbol in Σ. If we simply add one non-accepting state having the status of a trap state to a partial DFA we obtain a DFA.

NFA with ϵ transition

They can change state without consuming one of their output. They acceot all and only regular languages. You don't advance in your input. Let's now define it:

$$N = (Q, \Sigma, \delta, q_0, F)$$

Where all the elements outside the delta function are the same as the NFAs. The δ is a transiction function $\text{Qx}(\Sigma \cup \{\epsilon\}) \rightarrow 2^Q, with\ 2^Q$ denoting the class of subsets of Q.

ϵ-CLOSURE

Represented by the notation ECLOSE(q). It consists in adding all the states reachable from q itself through a sequence of one or more symbols ϵ.
BASE: $q \in ECLOSE(q)$
INDUCTION: $\hat{\delta}(q, xa)$ is computed as:

- $\{p_1, ..., p_k\} = \hat{\delta}(q, x)$
- $\{r_1, ..., r_m\} = \cup_{i=1}^{k} \delta(p_i, a)$
- $\hat{\delta}(q, xa) = ECLOSE(\{r_1, ..., r_m\})$

The language accepted by a ϵNFA E is $L(E) = \{w | \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$

Lezione 6

Mutual induction

Given several theorems $T_1, ..., T_k$ we wanna prove. Imagine we cannot apply inductive step on T_1, we need to apply it to other theorems. We use mutual induction.
BASE: for every T_i $i = i, ..., k \quad |x| = 0$
INDUCTIVE STEP: $T_1 \quad \leftarrow weassume\ T_1, ..., t_k$ with length (n-1) and $|x| = n \geq 0$. We do the same till T_k.

ϵ-NFA to DFA convrtion

We can convert a ϵ-DFA E into a DFA D, such that L(D) = L(E).
Construction details:

- $Q_D = \{S | S \subseteq Q_E, S = ECLOSE(S)\}$
- $q_0 = ECLOSE(q_0)$
- $F_D = \{S | S \in Q_D, S \cap F_E \neq \emptyset\}$

THEOREM: A language L is recognised by ϵ-NFA E if and only if L is recognized by DFA D.

Lezione 7

A finite automata is a "Bluepint" for constructing a machine recognizing a regular language. It can be easily implemented but it's often difficult interpreting it's meaning.

Regular Languages

Is a declarative way of describing a regular language. For example $01^* + 10^*$ denotes all binary strings that start with 0 followed by zero or more 1's or start with 1 followed by zero or more 0's.
+ is the disjunction operator, it means one expression or the other.
The following expression: $L \cup M$ uses the union operator, and gives us as a result the following set: $\{w|w \in L \ or \ w \in M\}$.
The following expression uses the concatenation operator: $L.M = \{w|w = xy, x \in L, y \in M\}$. This operator is not commutative as the union and is usually implicit. The concatenation between the empty set and any other set gives us still the empty set: $\emptyset.L = L.\emptyset = \emptyset$
The powers: $L^0 = \{\epsilon\}$, $L^k = L.L^{k-1}, for \ k \geq 1$
Kleen Closure: $L^* = \cup_{i=0}^{\infty} L^i$
Let's now construct $\emptyset^*$: $\emptyset^0 = \{\epsilon\}$, $\emptyset^i = \emptyset$, for every $i \geq 1$, therefore $\emptyset^0 = \{\epsilon\}$

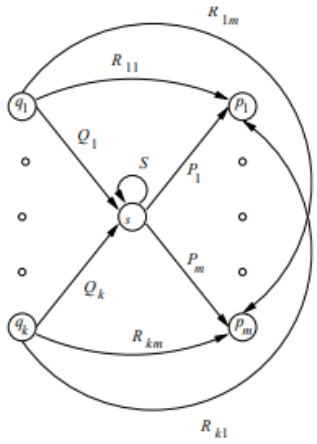
Regular Expressions

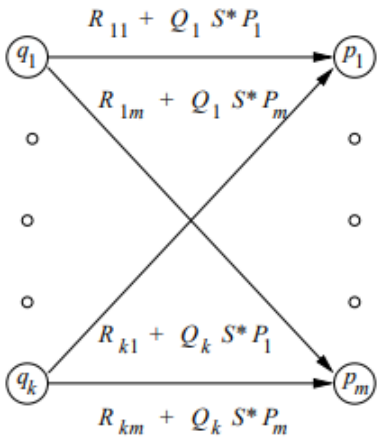
A regular expression E over alphabet Σ and the generated language L(E) are recursively defined as follows:
BASE: ϵ is a regular expression and $L(\epsilon) = \{\epsilon\}$, $\emptyset$ is a regular expression, and $L(\emptyset) = \emptyset$, if $\underline{a} \in \Sigma$, then $\underline{a}$ is a regular expression, and $L(\underline{a}) = \{a\}$
INDUCTION: if E and F are regular expressions, then E + F is a regular expression and L(E + F) $= L(E) \cup L(F)$. If E and F are regular expressions, then EF is a regular expression, and L(EF) = L(E)L(F). If E is a regular expression, then E^* is a regular expression, $L(E^*) = (L(E))^*$. If E is a regular expression, (E) is a regular expression, and L((E)) = L(E).
The operator, like in algebra, have a priority when combined with other operators and brackets: Keen closure ($*$), concatenation(.) and the union ($+$).

Theorem

If L = L(A) for some DFA A, then there exists a regular expression R such that L(R)=L
PROOF: We construct R form A using the elimination technique. It's based on the subsequent elimination of the states of the DFA, without altering the generated language. We replace the symbols with he equivalent regular expression ($a \rightarrow \underline{a}$), if there is o transition pair p,q, we create a new transition $p \rightarrow q$ with label $\emptyset$.
States $q_1, ..., q_k$ are the antecedents of s and the states $p_1, ..., p_m$ are the successors of s, assuming

$s \neq q_i, p; j$; those two sets are not necessarily disjoint (an antecedent can be a successor).
We can now eliminate state s and obtain the following: If antecedent or successor set is empty, we can





eliminate s without adding arcs (R empty, Q or P empty).
Now, to create a regular expression: for each final state q, we remove from the initial automanton all states excpet q_0 and q, resulting in an automaton A_q with almost two states. We convert each automaton A_q to a regular expression E_q and combine with the union operator. A_q can be in one of the following forms:
The case 1 correspond to the regular expression $E_q = (R+SU^*T)^*SU^*$, while the second case correspond to the regular expression $E_q = R^*$.
The final regular expression is then $\oplus_{q \in F} E_q$

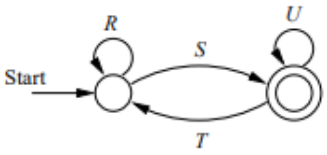


Figure 4: Case 1

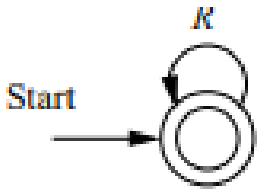
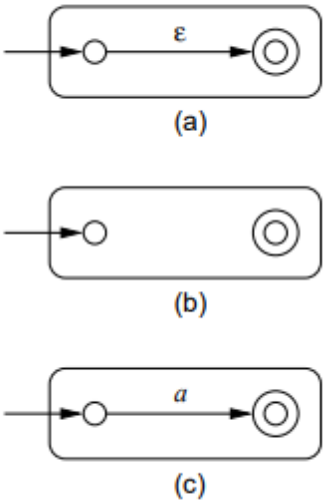


Figure 5: Case 2

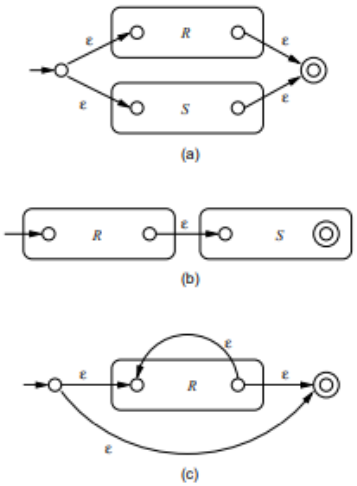
Lezione 8

Theorem

For every regular expression R we can construct a ϵ-NFA E such that L(E) = L(R).
PROOF: We construct E with only one final state, no entering arc in the initial state and no exiting arcs in the final state. Now we are gonna use structural induction to proof the theorem. As a base case we create automatas for the following elements: ϵ, $\emptyset$ *and* $\underline{a}$: INDUCTION: We now create Automata



for R+S, RS and R^*. We have now proved the previous theorem.



Algebraic laws

Regular expressions shares similarities with arithmetic expressions, we can consider the union as the sum and concatenation as the product, and share similar properties such as commutativity and so on. The oly operator which do not shares arithmetic properties is the kleen's operator $*$. In the following explanation, L,M and N are regular expressions and NOT LANGUAGES.
The union is commutative and associative: L + M = M + L and (L + M) + N = L + (M + N).
The concatenation is associative but <u>*NOT*</u> commutative: (LM)N = L(MN), $LM \neq ML$.
The empty set is the neutral element for the union operator: $\emptyset + L = L + \emptyset = L$.
ϵ is left and right identity for the concatenation operator $\epsilon L = L\epsilon = L$
$\emptyset$ is the left and right annihilator for concatenation $\emptyset L = L\emptyset = \emptyset$
Concatenation is left or right distributive over union: L(M + N) = LM + LN and (M+N)L=ML+NL
Union is idempotent: L + L = L

Properties of Kleen's closure

$(L^*)^* = L^*$, $\emptyset^* = \epsilon$, $\epsilon^* = \epsilon$, $L^+ = LL^* = L*L$, $L^* = L^+ + \epsilon$ and $L? = \epsilon + L$

Infix

The infix of a string is represented by the following notation and definition: given $x \in \Sigma^*$, we define inf(x) = $\{w|x = vxz, v, w, z \in \Sigma^*\}$. ϵ and the string x itself belong to the inf(x).
We can now define the infix of a language as follows: Given $L \subseteq \Sigma^*$ inf(L) = $w|\exists X \in Ls.t.\ w \in inf(x)$, or equivalently inf(L) = $\cup_{x \in L} inf(x)$. We can now present the following theorem: <u>If $L \in REG$ then $inf(L) \in REG$</u>

Lezione 9

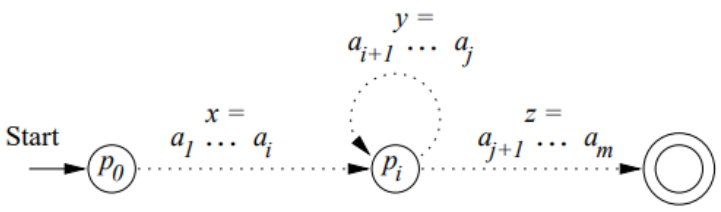
Suppose $L_{01} = \{0^n 1^n | n \geq 1\}$ were a regular language. Then it must be recognized by some DFA A; let k be the number of states of A. Assume A reads 0^k, then A must go through the following transitions: We have k + 1 references but just k states. Some 2 references points to the same state (Pigeonhole).

ϵ	p_0
0	p_1
00	p_2
...	...
0^k	p_k

By the Pigeonhole principle, there must exist a pair i,j with $i < j \leq k$ such that $p_i = p_j$. Let us call q this state. If you are using twice the same state, it means this state is collapsing information.
Now you can fool A: if $\sigma(q, 1\hat{i}) \notin F$, then the machine will foolishly reject $0^i 1^i$. if $\sigma(q, 1\hat{i}) \in F$, then the machine will foolishly accept $0^j 1^i$. Therefore A doesn't exists and L_{01} is not a regular language.

Pumping Lemma(Always in the exam Bro)

THEOREM: Let L be any regular language. Then $\exists n \in \mathbb{N}$ depending on L, $\forall w \in L$, with $|w| \geq n$, we can factorize w= xyz with: $y \neq \epsilon$, $|xy| \leq n$ and $\forall k \geq 0$, $xy^k z \in L$
PROOF:
Let us write xyz, where $x = a_1....a_i$, $y = a_{i+1}....a_j$, $z = a_{j+1}...a_m$, and let us represent it as it follows: Evidently, $xy^k z \in L$ $\forall k \geq 0$.



We use the pumping lemma to show that a language is not regular, basically we prove that is not satisfied, so it means that the language is not a regular language.

Lezione 10-11

Theorem: For any regular languages L e M, $L \cup M$ is regular.
Theorem: if L is regular language over Σ, then so is $I = \Sigma^* \setminus L$
Theorem: if L and M are regular languages, then so is $L \cap M$. We use intersection construction(which is in the proof of this theorem).
Therem: if L and M are regular languages, so is $L \setminus M$.
Theorem: if L is a regular, then also L^R is.
The intersection of a non-regular language L, and a infinite regular language L_2 is never a regular language.
Assume $L \in REG$. We cannot say anything about $L' \subset$ L and L $\subset L''$. More precisely: L' could or couldn't be REG, same for L''.

Lezione 12

Homomorphism

Let Σ and Δ be two alphabets. A homomorphism over Σ is a function $h : \Sigma \to \Delta^*$.
Informally, a homomorphism is a function which replaces each symbol with a string. For example: let $\Sigma = \{0, 1\}$ and define h(0) = ab and h(1)=ϵ; h is a homomorphism over Σ. We extend h to Σ^*: if $w = a_1a_2....a_n$ *then* $h(w) = h(a_1)....h(a_n)$. Equivalently, we can use a recursive definition:

$$h(w) = \begin{cases} \epsilon \; if \; w \; = \; \epsilon \\ h(x)h(a) \; if \; w \; = \; xa, \; x \in \Sigma^* \end{cases} \tag{1}$$

Example: using h from the previous example on the string 01001, we obtain ababab(we removed the 1 and replaced the 0's with ab). Homomorphism can be used to increase or decrease the length of a string. For a language L$\subseteq \Sigma^*$: h(L) = $\{h(w)|w \in L\}$.
Theorem: let L$\subseteq \Sigma^*$ be a regular language and let h be a homomorphism over $|Sigma$. Then h(L) is a regular language.
Proof: Let E be a regular expression generating L. We define h(E) as the regular expression obtained by substituting in E each symbol **a** with a_1 a_2 **....** a_k , $k \geq 0$. We now prove the statement L(h(E))=h(L(E)) using structural induction on E.
BASE: E = ϵ or else E = $\emptyset$. Then h(E) = E, and L(h(E))=L(E)=h(L(E)). E = **a** with $a \in \Sigma$. Let h(a) = $a_1a_2....a_k$, $k \geq 0$. Then L(**a**) = $\{a\}$ and thus h(L(**a**))=$\{a_1a_2...a_k\}$. The regular expression h(**a**) is $a_1a_2...a_k$. Then L(h(**a**)) = $\{a_1a_2...a_k\}$ = h(L(**a**)).
INDUCTION: Let E = F + G. We can write L(h(E)) =
= L(h(F+G))
= L(h(F)+h(G)) h defined over regex
= L((h(F)) $\cup$ L(h(G)) + definition
= h(L(F)$\cup$L(G)) hypothesis for F,G
= h(L(F+G)) + definition
= h(L(E))
We can do in the same way the proof for E = F.G. The hard part of the proving is the E = F^*. We can write:
L(h(E)) =
= L(h(F^*))
= L([h(F)]*) h defined over regex
= $\cup_{k\geq 0}$[L(h(F))]^k * definition
= $\cup_{k\geq 0}$[h(L(F))]^k inductive hypothesis for F
= $\cup_{k\geq 0}$h([L(F)]^k) h definition over languages
= h($\cup_{k\geq 0}$[L(F)]^k) h definition over languages

$= \mathrm{h}(\mathrm{L}(F^*))$ * definition
$= \mathrm{h}(\mathrm{L}(\mathrm{E}))$.

Conversion complexity

We can convert DFA, NFA, ϵ-NFA and regular expressions. What is the computational complexity of these conversions?

- From ϵ-NFA to DFA it takes $\mathcal{O}(2^n n^3)$ steps, so exponential time. With lazy evaluation it takes $\mathcal{O}(n^3 s)$.
- from NFA to DFA is exponential.
- From DFA to NFA it takes linear time.
- From FA to RE it takes exponential time.
- From RE to ϵ-NFA it takes linear time.

Decision problems

They give only yes or no as answers. The following are the decision problems we'll take care of: $L = \emptyset$? $w \in L$? and L=M?.
Empty language: L(A)$\neq \emptyset$ for FA A is and only if at least 1 final state is reachable from the initial state of A.

Lezione 13

Language membership: yes if the string is in the language, no otherwise. For an automaton with s states and $|w| = n$, we need $\mathcal{O}(n \cdot s^3)$ steps. If the automaton is an ϵ-NFA, it requires $\mathcal{O}(n \cdot s^3)$, or if we compute the ECLOSE(p) $\mathcal{O}(n \cdot s^2)$. Converting an NFA, or an ϵ-NFA, into a DFA it takes $\mathcal{O}(n)$ to see if a strings is in the language or not (carefull, the conversion can take exponential time).
Equivalent states: We want to know if two states have the same behavior. We need the states p and q to have equal response to input strings, with respect to acceptance.
We say that two states p and q are distinguishable if and only if

$$\exists w \ : \hat{\delta}(p, w) \in F \ and \hat{\delta}(q, w) \notin F \text{ or otherwise}$$

We can have two possible algorithms to see if two state are distinguishable or not:

- BASE: if $p \in F \ and \ q \notin F, then \ p \not\equiv q$
  INDUCTION: if $\exists a \in \Sigma : \ \delta(p, a) \not\equiv \delta(q, a), then \ p \not\equiv q$.
- correctness theorem: if p and q are not distinguishable by the algorithm, then $p \equiv q$.

Regular expression equivalence: to prove $L \stackrel{?}{=} M$ we do the following test:
Convert L and M in DFA, we construct the union (Don't give a shit about 2 initial states bruh), apply state equivalence algorithm. If two states are distinguishable then $L \neq M$, otherwise L = M.
DFA Minimization: we convert DFA to a version of it with the minimum number of states necessary to recognize the same language as before. We basically merge the equivalent states.
To minimize a DFA A (Q,Σ, δ, q_0,F) construct a DFA B = $(Q/_{\equiv}, \Sigma, \gamma, q_0/_{\equiv}, F/_{\equiv})$
Where the elements of $Q/_{\equiv}$ are the equivalence classes of $\equiv$; the elements of $F/_{\equiv}$ are the equivalence classes of $\equiv$ composed by states from F; $q_0/_{\equiv}$ is the set of states that are equivalent to q_0; $\sigma(p/_{\equiv}) = \delta(p, a)/_{\equiv}$
In order B to be well defined, we have to show that if $p \equiv q \ then \ \delta(p, a) \equiv \delta(q, a)$. In case $\delta(p, a) \not\equiv \delta(q, a)$ then the equivalence algorithm will conclude that $p \not\equiv q$. Thus B is well defined.
Transitivity theorem: if $P \equiv q \ and \ q \equiv r, then \ p \equiv r$. Proof: let's suppose the contrary $p \not\equiv r$. We'll have that $\exists w \ s.t. \hat{\delta}(p, w) \in F \ and \ \hat{\delta}(r, w) \notin F$ or the other way around. We now can have two cases: $\hat{\delta}(q, w)$ is accepting, then $q \not\equiv r$. Or, $\hat{\delta}(q, w)$ is not accepting, then $p \not\equiv q$. Therefore, it must be that $p \equiv r$.

Lezione 14

Context-Free Grammars CFG

Informal example: Let $L_{pal} = \{w | w \in \Sigma^*, w = w^R\}$, also called the language of all palindrome strings. It's obviously a not regular language since the pumping lemma is not satisfied. We can actually define it inductively:
BASE: ϵ,0,1 are palindrome
INDUCTION: if w is a palindrome, than also 0w0 and 1w1 are.
CFS are a formalism for recursively defining languages, such as L_{pal}, using rewriting rules: where P is

1. $P \to \epsilon$
2. $P \to 0$
3. $P \to 1$
4. $P \to 0P0$
5. $P \to 1P1$

a variable representing strings of a language. In this grammar P is also the initial symbol.
Formal Definition: A context free grammar(CFG) is a tupe and a generative device defined as follows:

G=(V,T,P,S)

where V is the finite set of variables, or nonterminals; T is a finite set of terminal symbols, representing also the language alphabet; P is a finite set of productions having the form $A \to \alpha$, where A is a variable and α is a string; S is a variable called initial symbol. A CFG for a palindrome string is $G_{pal} = (\{P\}, \{0,1\}, A, P)$ with A = $\{P \to \epsilon, P \to 0, P \to 1, P \to 0P0, P \to 1P1\}$. In case we have several productions involving the same variable we can use the **compact notation**: $A \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$.

Derivation

In order to generate strings using a CFG, we define a binary relation $\underset{G}{\Rightarrow}$ over $(V \cup T)^*$, called rewrites. Let G =(V,T,P,S) be a CFG, $A \in V, \{\alpha, \beta\} \subset (V \cup T)^*$. if $A \to \gamma \in P$, then $\alpha A \beta \underset{G}{\Rightarrow} \alpha\gamma\beta$ and we say that $\alpha A \beta$ derives in one step from $\alpha\gamma\beta$. In case G is understood by the context, we use a simplified notation: $\alpha A \beta \Rightarrow \alpha\gamma\beta$.
We can define $\overset{*}{\Rightarrow}$ as the reflexive closure of $\Rightarrow$:
BASE: Let $\alpha \in (V \cup T)^*. Then\ \alpha \overset{*}{\Rightarrow} \alpha$.
INDUCTION: if $\alpha \overset{*}{\Rightarrow} \beta and\ \beta \Rightarrow \gamma, then\ \alpha \overset{*}{\Rightarrow} \gamma$.
Relation $\overset{*}{\Rightarrow}$ is called derivation. We often write derivations by indicating all of the intermediate steps.
We can have two way of ding derivations by avoiding the choice of the variable we want to rewrite:

- Leftmost derivation $\underset{\text{lm}}{\Longrightarrow}$ and can use the reflexive and transitive closure, so we can have $\overset{*}{\underset{\text{lm}}{\Longrightarrow}}$. It always rewrites the leftmost variable with some production.
- Rightmost derivation: it always rewrites the rightmost variable with some production. $\underset{\text{rm}}{\Longrightarrow}$ and $\overset{*}{\underset{\text{rm}}{\Longrightarrow}}$ as well.

Language generated by a CFG: Let G = (V,T,P,S) be some CFG. The generated language of G is

$$\text{L(G)}=\{w \in T^* | S \overset{*}{\underset{\text{G}}{\Longrightarrow}} \} w$$

that is the set of all strings in T^* that can be derived from the start symbol. L(G) is a context free language, or CFL.

Lezione 15

Let $G_{pal} = (\{P\}, \{0,1\}, A, P)$, where A = $\{P \to \epsilon|0|1|0P0|1P1\}$.
Theorem:$L(G_{pal}) = \{w|w \in \{0,1\}^*, w = w^R\}$.
Proof: ($\supseteq$ part). Assume w = w^R. Using induction on $|w|$, we show $w \in L(G_{pal})$.
BASE: $|w| = 0$ *or* $|w| = 1$. Then w is ϵ, 0 or 1. Since $P \to \epsilon, P \to 0, P \to 1$ are productions of the grammar, we conclude that $\underset{\text{G}}{\overset{*}{\Rightarrow}} w$.
INDUCTION: Assume now $|w| \geq 2$. Since $w = w^R$, we must have w = 0x0 or else w = 1x1, with $x = x^R$. If w = 0x0, from the inductive hypothesis we have $P \overset{*}{\Rightarrow} x$. Then $P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$. Therefore $w \in L(G_{pal})$. The case w = 1x1 can be dealt in the same way.
Proof: ($\subseteq$ part). Assume now $w \in L(G_{pal})$. We show $w = w^R$. Since $w \in L(G_{pal})$, we have $P \overset{*}{\Rightarrow} w$. We use induction on the number of steps of the derivation.
BASE: The derivation $P \overset{*}{\Rightarrow} w$ has 1 step. Then w must be ϵ, 0 or 1. All the three generated strings are palindrome.
INDUCTION: Let $n \geq 2$ the number of steps in the derivation. At the first step only two cases are possible: $P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$ or else $P \Rightarrow 1P1 \overset{*}{\Rightarrow} 1x1 = w$, and in both cases the second part of the derivation consists on n - 1 steps. By the inductive hypothesis, x is a palindrome string. The also w is a palindrome string.

Sentential form

Let G = (V,T,P,S) be a CFG and let $\alpha \in (V \cup T)^*$. If $S \overset{*}{\Rightarrow} \alpha$ we say that α is a sentential form. If $S \underset{\text{lm}}{\overset{*}{\Longrightarrow}} \alpha$ we say that α is a left sentential form. Same for the right sentential form.

Derivation factorization

Assume $A \Rightarrow X_1X_2 \cdots X_k \overset{*}{\Rightarrow} w$. We can factorize w as $w_1w_2 \cdot w_k$ such that $X_i \overset{*}{\Rightarrow} w_i,\ i \leq i \leq k$. Substring w_i can be identified from derivation $A \overset{*}{\Rightarrow} w$ by considering only those derivation steps that rewrite X_i.

Parse trees

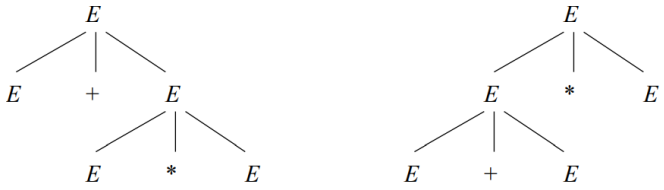
Graphical representation alternative to derivation. They represent the syntactic structure of a sentence according to the grammar. In compilers, they are the structure of choice when translating in executable code.

Lezione 16

The yeald of a tree is the string obtained by reading the leaves from left to right. We say that a parse tree is complete when the root is the initial symbol and the leaves compose the string. A parse tree can be associated with several derivations (it's not a 1-to-1 relation) and a derivation can be associated with several parse trees. All types of derivations, normal, leftmost and rightmost must be true at the same time. We can always construct on of them starting from one of the others.

Ambiguity

Given a derivation $E+E*E$ we can obtain several parse trees that will give the same output. It depends by the fact that we can give priority to 2 different operators. Let g be a CFG. G is ambiguous if there



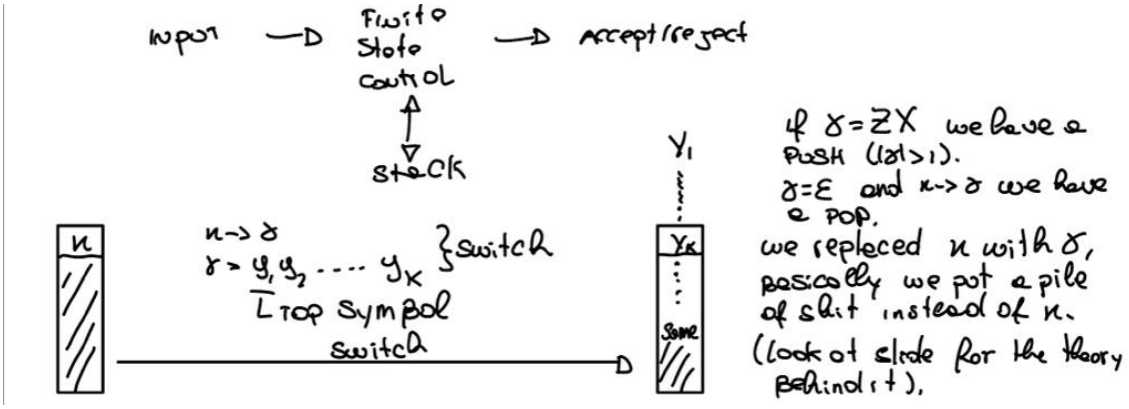
exists a string in L(G) with more than one parse tree. If all strings in L(G) have only one parse tree, G is said to be unambiguous.
A parse tree is associated with a unique left(right)most derivation and a left(right)most derivation is associated with a unique parse tree.
A CFL is inherently ambiguous when every CFG s.t L(G) = L is ambiguous.
A regular language is always a CFL. From a regular expression or from a FA, we can always construct a CFG generating the same language.
From REGEX to CFG LEt E be any regular expression. We use a variable for E(start symbol) and a variable for each subexpression on E. By using structural induction we build the productions of our CFG. Here are some rules:

- if E = **a**, we add the production $E \to a$
- if $E = \epsilon$ we add the production $E \to \epsilon$
- if $E = \emptyset$, then the production set is empty
- if E = F+G we add the production $E \to F|G$
- if E = FG we add the production $E \to FG$
- if $E = F^*$ then we add the production $E \to FE|\epsilon$

We can convert a FA into a CFG. We use a variable Q for each state q of the FA. The initial symbol is Q_0. For each transition from state p to state q under symbol a, add production $P \to aQ$. If q is a final state, add production $Q \to \epsilon$.

PUSH-DOWM AUTOMATA

They are based on stacks or piles. It is basically a ϵ-NFA with a stack representing the auxiliary memory. The stack can record an arbitrary number of symbols and can release symbolse with a strict policy: LIFO. On a formalism level, PUSH-DOWN AUTOMATA $\equiv$ CFGs.
A transition consumes a sigle symbol from the input, or else is a ϵ-transition, it updates the current state or replaces the top-most symbol of the stack with a string of symbols, including ϵ.



Lezione 18

Let us consider the language $L_{wwr} = \{ww^R | w \in \{0,1\}^*\}$ generated by the CFG productions $P \to 0P0$ $P \to 1P1$ $P \to \epsilon$. A push-down automaton has 3 states, and operates as follows:
Guess that you are reading w. Stay in state q_0 and push the input symbol into the stack. Guess that you are reading w. guess that you are at the boundary between w and w^R. Go to q_1 using an ϵ-transition. You are now reading the first symbol of w^R. Compare it to the topof the stack. If they match, pop the stack and remain in state q_1. If they don't match, the automaton halts, it doesn't have a next move. if the stack is empty, go to state q_2 and accept. The mathematical definition of a PDA is the following:

$$\mathrm{P} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

Q, Σ, q_0 and F are the same as an ϵ-NFA. Γ is the alphabet for the stack, $Z_0 \in \Gamma$ it's the initial stack symbol. δ is similar as before: $Qx\Sigma \cup \{\epsilon\}x\Gamma \to 2^{Qx\Gamma^*}$ is a transition function, always **finite** subset of $2^{Qx\Gamma^*}$. It has 3 inputs and not 2 as before.
The PDA for the language L_{wwR} is the following:

$$\mathrm{P} = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

and can be represented also with the following table: or can also be represented graphically(Fig. 6)

	$0, Z_0$	$1, Z_0$	0,0	0,1	1,0	1,1	ϵ, Z_0	$\epsilon, 0$	$\epsilon, 1$
$\to q_0$	$q_0, 0Z_0$	$q_0, 1Z_0$	$q_0, 00$	$q_0, 01$	$q_0, 10$	$q_0, 11$	q_1, Z_0	$q_1, 0$	$q_1, 1$
q_1			q_1, ϵ			q_1, ϵ	q_2, Z_0		
$\star q_2$									

Instantaneous Description

a computation of a PDA is a sequence of "configurations" of the automaton obtained from the other by consuming an input symbol or else by reading ϵ. In order to formalize the configuration of a PDA we introduce the mathematical notion of instantaneous description. To formalize the computation of a PDA we then introduce a binary relation over instantaneous description called moves. Formally, an ID is a triple (q,w, γ) where q is the current state, w is the part of the input still to be read and γ is the stack content, the topmost symbol at the left.

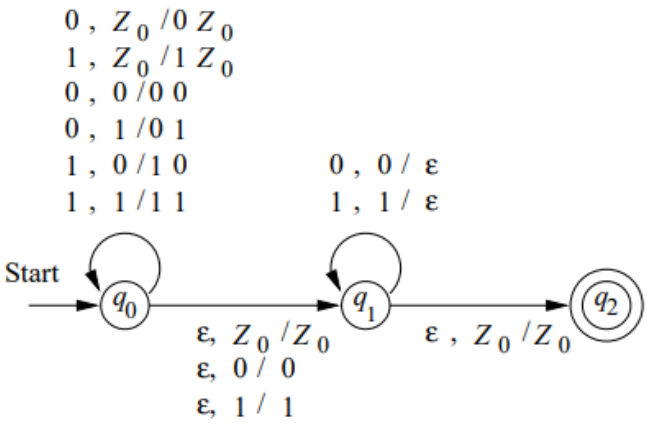


Figure 6: Graphical representation of a PDA

Computation

Let P $= (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be a PDA. We deine a binary relation over the set of IDs called moves, written $\vdash_P$ or simply $\vdash$. $\forall w \in \Sigma^*, \beta \in \Gamma^*$:

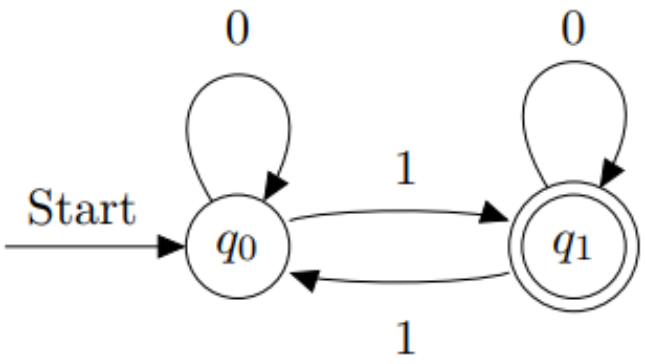
$$(p, \alpha) \in \delta(q, a, X) \Rightarrow (q, aw, X\beta) \vdash (p, w, \alpha\beta)$$
$$(p, \alpha) \in \delta(q, \epsilon, X) \Rightarrow (q, w, X\beta) \vdash (p, w, \alpha\beta)$$

We define $\vdash_P^*$ as the reflexive and transitive closure of $\vdash_P$. We use $\vdash_P^*$ to define a computation of a PDA.

EXERCISES

Exercise 1.1

Let L = $\{w|w \in \{0,1\}^*, \#_1(w)odd\}$, we want to prove that the automaton in the figure recognize the language L s.t L = L(A). We can define the following properties as it follows:



P_{q_0} := true if and only if $\#_1(x)iseven$
P_{q_1} := true if and only if $\#_1(x)isodd$
We prove the following statements:

- (i) $\forall x \in \{0,1\}^*,\ P_{q_0} is\ true\ \Leftrightarrow \hat{\delta}(q_0, x) = q_0$
- (ii) $\forall x \in \{0,1\}^*,\ P_{q_1} is\ true\ \Leftrightarrow \hat{\delta}(q_0, x) = q_1$

Proof: "(i) $\Rightarrow$" by using induction.
BASE: (x = ϵ) then P_{q_0} is true since $\hat{\delta}(q_0, \epsilon) = q_0$; The induction holds for ϵ
INDUCTION $|x| > 1$. If P_{q_0} is false, then the string has an odd number of 1's and we are done. Let's assume P_{q_0} true. Because $|x| > 0$, we can write x = $y \cdot a,\ a \in \{0,1\}$. We now have two cases:

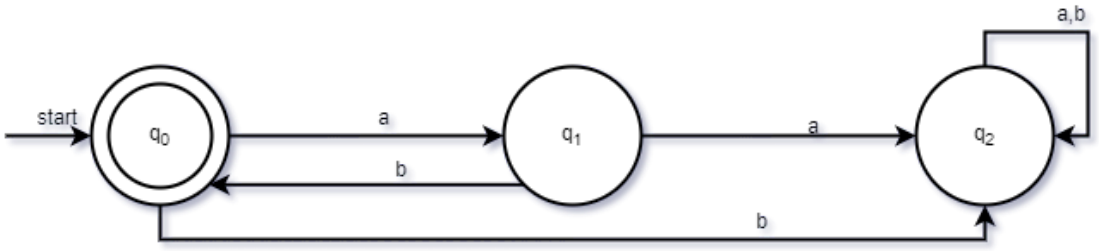
- a = 0. The $\#_1(x) = \#_1(y)$ which is even. By applying "$(i) \Rightarrow$", we can conclude that $\hat{\delta}(q_0, y) = q_0$. From definition of A, $\delta(q_0, 0) = q_0$. Then: $\hat{\delta}(q_0, x) = \delta(\hat{\delta}(q_0, y), 0) = \delta(q_0, 0) \overset{A}{\Rightarrow} q_0$.
- a = 1. The $\#_1(y) = \#_1(x) - 1$ where $\#_1(x)$ is even. P_{q_1} true. By induction on y and applying "$(ii) \Rightarrow$", we get $\hat{\delta}(q_0, y) = q_1$. Then: $\hat{\delta}(q_0, x) = \delta(\hat{\delta}(q_0, y), 1) = \delta(q_1, 1) \overset{A}{\Rightarrow} q_0$.

Exercise lecture 11

Let L = $\{a^n b^n | n \geq 1\}$, use the pumping lemma to prove that the L is not regular. By imposing y = a^m, with $1 \leq m \leq n$, for k = 0 $\Rightarrow w_k = xy^K z \Rightarrow w_0 \notin L$

Exercise lecture 12

Let L = $\{(ab)^n | n \geq 0\}$. Write an automaton A and prove that L(A)=L by using mutual induction. We



can define the following properties:

- $P_{q_0}(w) \stackrel{\text{d}}{\Rightarrow} w \in L$
- $P_{q_1}(w) \stackrel{\text{d}}{\Rightarrow} w \in L \cdot \{a\}$
- $P_{q_2}(w) \stackrel{\text{d}}{\Rightarrow} w \notin L \cap w \notin \{a\}$

And we have the following theorems to prove:

- (i) $P_{q_0}(w) \Leftrightarrow \hat{\delta}(q_0, w) = q_0$
- (ii) $P_{q_1}(w) \Leftrightarrow \hat{\delta}(q_0, w) = q_1$
- (iii) $P_{q_2}(w) \Leftrightarrow \hat{\delta}(q_0, w) = q_2$

Proof: "(i)⇒" by induction.
BASE w = ϵ then P_{q_0} is true since $\delta(q_0, \epsilon) = q_0$.
INDUCTION $|w| > 1$. Let us write $w = x \cdot Z$, where $x \in \Sigma^*, Z \in \Sigma$. Let us assume $P_{q_0}(w)$ is true. Then $w \in L$.
We have Z = b, $x \in L \cdot \{a\}$ then $P_{q_1}(x)$ is true. Use inductive hypothesis on x, "$(ii) \Rightarrow$", we conclude that $\hat{\delta}(q_0, x) = q_1$. Then we have $\hat{\delta}(q_0, w) = \delta(\hat{\delta}(q_0, x)b) = \hat{\delta}(q_1, b) = q_0$.
Proof: "(i) ⇐"
BASE as before
INDUCTION $|w| > 1$. Let w = $x \cdot Z$, where $x \in \Sigma^*, Z \in \Sigma$. $\hat{\delta}(q_0, w) = q_0$ then $\hat{\delta}(q_0, x) = q_1$. Use $\hat{\delta}(q_0, x) = q_1$ in statement (ii) ⇐, we conclude that P_{q_1}(x) is true. Then $x \in L \cdot \{a\}$. We have $x \cdot b \in L \cdot \{a\} \cdot \{b\} \subseteq L$. By definition of P_{q_0} we have $w = x \cdot b$ has property P_{q_0}.

Exercise lecture 13

Using the pumping lemma prove that the following language is not regular:

$$L = \{w | w \in \{a, b\}^*, \#_a(w) = \#_b(w) \leq 2\#_b(w)\}$$

By choosing the string w = $a^n b^n$ we obtain that the number of a's is less than the number of b's for k = 0, henceforth the language is not regular.

Exercise lecture 15

Given the following language L = $\{w | w \neq \epsilon, w \textit{ is a string of balanced parenthesis}\}$. G is a CFG with the following productions:

$$S \to SS \; S \to (S) \; S \to ()$$

Prove that L = L(G).
We have the following property:
$\forall x \in \Sigma^* \; P_s(x) := x$ balanced parenthesis $\neq \epsilon$.
The theorem that follows is the following:
(i) $P_s(x) \Leftrightarrow S \overset{*}{\underset{\text{G}}{\Rightarrow}} x$ (x is generated by G in 0 or more steps).
Proof: ($\Rightarrow$ part) induction on $|x|$
BASE: $|x| = 2 \; \to x = ()$; Then $S \underset{\text{G}}{\Rightarrow} ()$. $() \in L(G)$.
INDUCTION: $|x| \geq 2$. Two cases arises:

1. x = (z), then we have $P_s(z)$ true and $|z| < |x|$. By inductive hypothesis: from $P_s(z)$ we get $\overset{*}{\underset{\text{G}}{\Rightarrow}}$. Then we can write, with the help of composition, the following: $S \Rightarrow (S) \overset{*}{\Rightarrow} (z) = x$. We proved that $x \in L(G)$.

2. $x = z_1 z_2$ with $P_s(z_1), P_s(z_2)$ true. Using the inductive hypothesis we get $|z_1| < |x| \; and \; |z_2| < |x|$: $S \overset{*}{\underset{G}{\Rightarrow}} z_1, \; S \overset{*}{\underset{G}{\Rightarrow}} z_2$. Using composition we obtain $s \Rightarrow SS \overset{*}{\Rightarrow} z_1 S \overset{*}{\Rightarrow} z_1 z_2 = x$. Then $x \in L(G)$

Proof: ($\Leftarrow$ part) induction on the umber of derivation steps.
BASE: length 1. $S \underset{G}{\Rightarrow} () = x$. We know that $P_s('()')$ is true.
INDUCTION: length > 1. Two cases arise again:

1. rule $S \to (S)$. We have that $S \underset{G}{\Rightarrow} (S) \overset{*}{\underset{G}{\Rightarrow}} (z) = x$. Here we have used factorization!

2. rule $S \to SS$. Then $S \underset{G}{\Rightarrow} SS \overset{*}{\underset{G}{\Rightarrow}} x$. Using factorization, we split x = $z_1 z_2$ s.t $S \overset{*}{\underset{G}{\Rightarrow}} z1, S \overset{*}{\underset{G}{\Rightarrow}} z2$. Those derivations are shorter than the original derivation. Using the Inductive Hypothesis we get that $P_s(z_1) \; and \; P_s(z_2)$ are true. Then $z_1 \cdot z_2$ must still be balanced. So $x = z_1 z_2$ has the property P_s. $P_s(x)$ is true.

Exercise lecture 17

Given the following language L = $\{a^i b^j c^k | i, j, k \geq 1\}$ write the CFG which generates L. We get the following productions: $S \to aS|aB \; B \to bB|bC \; C \to cC|c$. We want to prove that L(G) = L.