UNIVERSITÀ DEGLI STUDI DI PADOVA

NOTES OF

Automata

REGULAR, CONTEXT FREE and RECURSIVE LANGUAGES

(Version 09/02/2022)

Edited by:
Stefano Ivancich

CONTENTS

1. Introduction

One of the main goals of theoretical computer science is the mathematical study of computation

- computability : what can be computed?
- tractability : what can be efficiently computed?

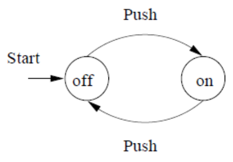
The mathematical study of computation requires

- abstract models of computation : automata theory
- abstract representations of problems/data : formal language theory

1.1. Introduction to finite automata

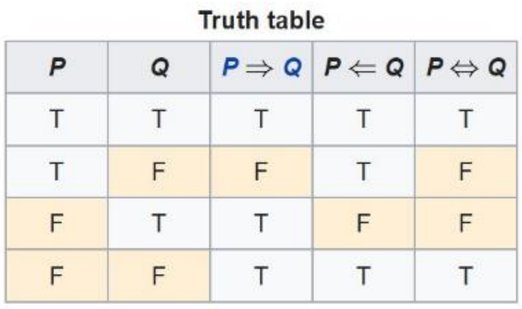
Finite Automata (FA): finite set of **states** with **transitions** from one state to another.
Representation with a graph where:

- **Nodes**: represent states
- **Arcs**: represent transitions
- **Labels**: on each arc indicate what is causing the transition



Recognition model: it takes as input a sequence (string) and either accepts or rejects. Ex. FA. Are operational.
Generative model: generates all the desired sequences (no input). Ex. grammars, regular expressions. Are declarative.

1.2. Formal proof techniques


Truth table

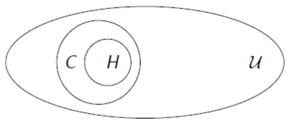
P	Q	$P \Rightarrow Q$	$P \Leftarrow Q$	$P \Leftrightarrow Q$
T	T	T	T	T
T	F	F	T	F
F	T	T	F	F
F	F	T	T	T

1.2.1. Deductive

If H, then C H=hypothesis(true/false), C=conclusion

- H is a sufficient condition for C
- C is a necessary condition for H
- Also written as **C if H**

Insiemistic interpretation:



$H \Rightarrow C$ is equivalet to $H \subseteq C$: if H is true, C can't be false

Deduction: Sequence of statements that starts from one or more hypotheses and leads to a conclusion
Each step of the deduction uses some **logical rule**, applying it to the hypotheses or to one of the previously 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.

C_1 **if and only if** C_2
require proofs for both directions

- if C_2 then C_1
- if C_1 then C_2

Additional techniques

- **Reduction to definitions:** Convert all terms in the assumptions using the corresponding definitions
- **Proof by contradiction:** To prove “if H then C", prove “H and not C implies falsehood"
- **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 or the quantifiers affect the meaning of the statement

Set Equality: To prove $E = F$ we have to prove both $E \subseteq F$ and $F \subseteq E$

- if x is in E then x is in F
- if x is in F then x is in E

Contrapositive (modus tollens): The statement “if H then C" is equivalent to the statement “if C is false then H is false". Proof of equivalence uses truth table.

1.2.2. Induction

Inductive proof: used with objects defined recursively

Induction on integers: we need to prove statement $S(n)$, for non-negative integer numbers n

- in the **base** case we show $S(i)$ for some specific integer i (usually $i = 0$ or $i = 1$). Or a finite number of cases.
- in the **inductive** step, for $n \gg i$ prove statement “if $S(n)$ then $S(n+1)$“

We can then conclude that $S(n)$ is true for every $n \gg i$, where i is the base case.
We can extend the inductive step and demonstrate for a certain k ¡ 0 : \if Spn kq, Spn k ⍰ 1q, ..., Spn 1q, Spnq then Spn ⍰ 1q"

Structural induction:
To prove theorems for structure X which is recursively defined:

- show the statement for the base case of the definition of X
- show the statement for X on the basis of the same statement holding for the subparts of X, according to X's definition

Mutual Induction
Sometimes it is not possible to prove a statement $S_1(n)$ by induction, because the statement depends on statements $S_2(n), \dots, S_k(n)$ of different types
We then need to prove jointly a family of statements $S_1(n), \dots, S_k(n)$ by mutual induction on n

1.3. Basic concepts of automata theory

Alphabet Σ: finite and nonempty set of atomic symbols

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

String: finite sequence of symbols from some alphabet
Empty string ϵ: composed of 0 symbols. Can be chosen from each alphabet.
Length $|w|$: Number of occurrences (standpoints) for the symbols in the string.
Concatenation: xy

- $x\epsilon = \epsilon x = x$

Powers of an alphabet Σ^{k}: is the set of all k-length strings with symbols from Σ

- $\Sigma = \{0,1\}$
- $\Sigma^1 = \{0,1\}$
- $\Sigma^2 = \{00, 01, 10, 11\}$
- $\Sigma^0 = \{\epsilon\}$ for each alphabet
- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$
- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \dots$
- $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

Language L: set of strings arbitrarily chosen from Σ^*

- $L \subseteq \Sigma^*$ is a language
- Extensive representation: $L = \{\epsilon, 01, 0011, \dots\}$
- Intensive representation: $L = \{w | \text{statement specifying } w\}$

Let $P(x)$ be a predicate expressing some mathematical property of element x
Decision problem associated with P: on input x, decide whether $P(x)$ holds true.
Associated formal language: $L_P = \{x | P(x) \text{ holds true}\}$
Can be reformulated as: Given as input an element x (viewed as a string), $x \in L_P$?

Many mathematical problems are not decision problems, but require instead a computation that constructs an output result. Solving a general (non-decision) problem cannot be simpler than solving the associated decision problem.

2.Finite Automata

2.1. Deterministic finite automata (DFA)

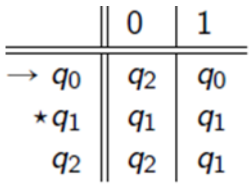
Read input from left to right. They can only store a limited quantity of information.

Definition: $\boldsymbol{A = (Q, \Sigma, \delta, q_0, F)}$

- Q: finite set of states
- Σ: finite set of input symbols
- δ: transition function $Q \times \Sigma \to Q$
- $q_0 \in Q$: initial state
- $F \subseteq Q$: set of final states

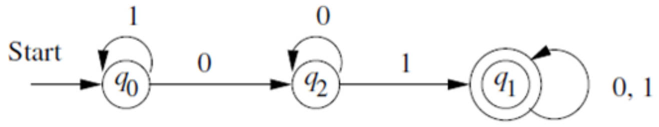
Notations:

- **Transition table:**
    - rows: states
    - columns: input alphabet symbols
    - Arrow: Start State
    - asterisks: final states



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

- **Transition Diagram:**
    - Node: state
    - Arc: $\delta(q, a)$
    - Strat arrow on q_0
    - Final states: nodes with double circle
    - #states*#alphabet_symbols = #arrows
    - For each state there must be #outgoing_arcs=Σ



Acceptance of a string $w = a_1 a_2 \dots a_n$: $\delta(q_{i-1}, a_i) = q_i$, if $q_n \in F$ then $a_1 a_2 \dots a_n$ is accepted.
Or if there is a path in the transition diagram that starts in the initial state, ends in a final state and has a sequence of transitions with labels $a_1 a_2 \dots a_n$

Extended transition function $\boldsymbol{\hat{\delta}}$: operates on entire strings

- **Base**: $\hat{\delta}(q, \epsilon) = q$
- **Induction:** $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$

a is the last symbol of w

Language accepted by DFA $L(A) = \{w | \hat{\delta}(q_0, w) \in F\}$ set of strings w that starting from the initial state reach one of the final states.
Those languages are called **regular languages**.

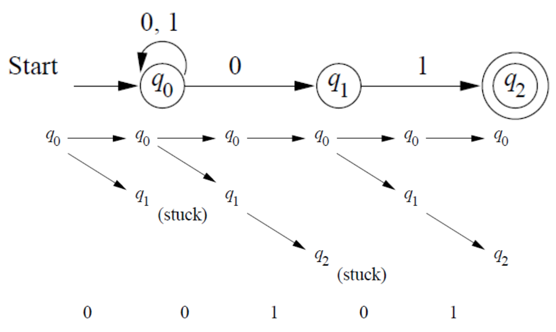
Prove that an automaton $\boldsymbol{A}$ accept the language $\boldsymbol{L}$. Prove that $L = L(A)$. Mutual induction.

- Define a family of properties P_q, one for each state q of A. ($\forall x \in \Sigma^*, P_{q_i}(x)\ holds\ IFF \dots$)
- Prove that, for each property P_q: $\forall x \in \Sigma^*$, $P_{q_i}(x)$ holds IFF $\hat{\delta}(q_0, x) = q_i$
  This means that $P_{q_i}(x)$ is true iff, starting from the initial state and reading x, we reach q_i.
- Proof IF $P_{q_i}(x)$ THEN $\hat{\delta}(q_0, x) = q_i$:
    - base: $|x| = 0$. This implies that $x = \epsilon$. So $P_{q_i}(\epsilon)$ holds. We can write $\hat{\delta}(q_0, x) = q_i$

- Induction: $|x| = n > 0$.
  - If $P_{q_i}(x)$ is false, then the implication is true.
  - If $P_{q_i}(x)$ is true, then $x = wa$, assume w correct, apply the inductive hypotheses, we can write $\hat{\delta}(q_0, x) = \delta(\hat{\delta}(q_0, w), a) = \delta(q_{i..}, a) = q_i$
- Proof IF $\hat{\delta}(q_0, x) = q_i$ THEN $P_{q_i}(x)$:
  - base: $|x| = 0$. This implies that $x = \epsilon$. So $\hat{\delta}(q_0, \epsilon) = q_0$ true, $P_{q_i}(x)$ true, so the implication is true. (If $\hat{\delta}(q_0, \epsilon) = q_i$ false then the implication is true)
  - Induction: $|x| = n > 0$.
    - If $P_{q_i}(x)$ is false, then the implication is true.
    - If $P_{q_i}(x)$ is true, then $x = wa$. ....?????

2.2. Nondeterministic finite automata (NFA)

Accept only regular languages. Easier to design than DFAs. Useful to search for a pattern in a text. Can simultaneously be in different states.
Accepts if at least one final state is reached at the end of the scan of the input string.
Equivalently, in a given state the automaton can "guess" which next state will lead to acceptance
Can be seen as set of states that exists simultaneously, and when a new character is read each state is updated.



Definition: $\boldsymbol{A} = (\boldsymbol{Q}, \boldsymbol{\Sigma}, \boldsymbol{\delta}, \boldsymbol{q_0}, \boldsymbol{F})$

- Q: finite set of states
- Σ: finite set of input symbols
- δ: transition function $Q \times \Sigma \to 2^Q$, where 2^Q is the set of all subsets of Q (power set)
- $q_0 \in Q$: initial state
- $F \subseteq Q$: set of final states

Extended transition function $\hat{\boldsymbol{\delta}}$:

- **Base**: $\hat{\delta}(q, \epsilon) = \{q\}$
- **Induction:** $\hat{\delta}(q, xa) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a)$

transition function δ

	0	1
$\to q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$\star q_2$	$\varnothing$	$\varnothing$

Computation of $\hat{\delta}(q_0, 00101)$

- $\hat{\delta}(q_0, \epsilon) = \{q_0\}$
- $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$
- $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$
- $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
- $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$
- $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

Language accepted by NFA: $L(A) = \{w | \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$ set of strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ contains at least one final state. This means that at least one computation for w leads to acceptance.

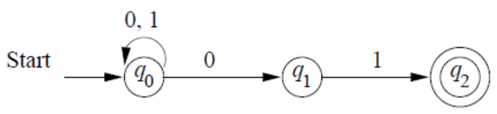
2.2.1. Equivalence DFA - NFA

NFAs are easier than DFAs to program, since nondeterminism makes it possible to simplify the structure of the automaton.

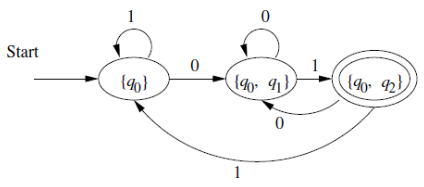
For every NFA N there exist some DFA D such that $L(D) = L(N)$. Proof with subset construction.

NFA to DFA with subset construction:

- Q_D = set of subsets of Q_N, for a total of 2^n states. But a lot are not reachable so we can use lazy evaluation.
- $F_D = \{S \subseteq Q_N | S \cap F_N \neq \emptyset\}$, sets of states that contain at least an accepting state.
- $\forall S \subseteq Q_N$ and $\forall a \in \Sigma$: $\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$



	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$\star\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\star\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\star\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$\star\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

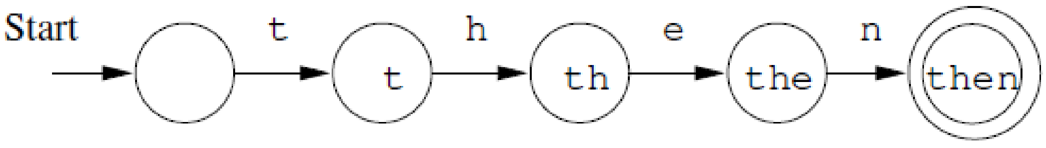


NFA to DFA with Lazy Evaluation: used to avoid writing all states Q_D

- **Base**: $S = \{q_0\}$ is accessible in D
- **Induction:** if state S is accessible in D, then it's accessible also the state $\delta_D(S, a)$ $\forall a \in \Sigma$ with $\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$

$\emptyset$ is considered a single trap state if it appears as final unique solution of $\delta_D(S, a)$. If it appears with other it's like it doesn't exist.

Partial DFA: it has at maximum 1 outgoing transition for each state in Q and for each symbol in Σ. It can be transformed in an DFA by adding some **trap states**: non-accepting states that have a transition on themselves.



2.2.2. Theorems

Given $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$, built $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ using subset construction, then $L(D) = L(N)$

Proof We first prove that, for every string $w \in \Sigma^*$, we have

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

We use induction on $|w|$

Base $w = \epsilon$. The claim follows from the definition

Induction

$$\begin{array}{rcll} \hat{\delta}_D(\{q_0\}, xa) & = & \delta_D(\hat{\delta}_D(\{q_0\}, x), a) & \text{definition of } \hat{\delta}_D \\ & = & \delta_D(\hat{\delta}_N(q_0, x), a) & \text{induction} \\ & = & \bigcup_{p \in \hat{\delta}_N(q_0, x)} \delta_N(p, a) & \text{definition of } \delta_D \\ & = & \hat{\delta}_N(q_0, xa) & \text{definition of } \hat{\delta}_N \end{array}$$

$L(D) = L(N)$ now follows from the definition of F_D □

Language L is accepted by a DFA IIF L is accepted by an NFA.

Proof (If) Previous theorem

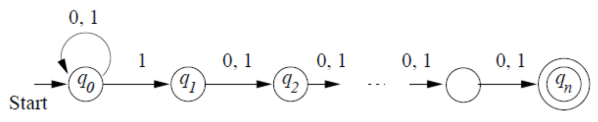
(Only if) Any DFA can be converted into an equivalent NFA by modifying δ_D into δ_N according to the following rule

If $\delta_D(q, a) = p$, *then* $\delta_N(q, a) = \{p\}$

By induction on $|w|$ one can show that $\hat{\delta}_D(q_0, w) = p$ if and only if $\hat{\delta}_N(q_0, w) = \{p\}$ □

Unlucky case: there exist a NFA N with $n + 1$ states that has no equivalent DFA with less than 2^n states.

Proof Let N the NFA



$$L(N) = \{x1c_2c_3 \cdots c_n \mid x \in \{0,1\}^*, c_i \in \{0,1\}\}$$

Intuitively, an equivalent DFA must "remember" the last n symbols it has read

Suppose there exists a DFA D equivalent to N with fewer than 2^n states

There are 2^n binary strings of length n. Since D has fewer that 2^n states, there must be

- a state q,
- binary strings $a_1a_2 \cdots a_n \neq b_1b_2 \cdots b_n$,

such that

$$\hat{\delta}_D(q_0, a_1a_2 \cdots a_n) = \hat{\delta}_D(q_0, b_1b_2 \cdots b_n) = q$$

Since $a_1a_2 \cdots a_n \neq b_1b_2 \cdots b_n$, there exists i with $1 \leqslant i \leqslant n$ such that $a_i \neq b_i$; we assume $a_i = 1$ and $b_i = 0$ (the other case being symmetrical)

Case 1: $i = 1$; we have

$$\hat{\delta}_D(q_0, 1a_2 \cdots a_n) \in F$$
$$\hat{\delta}_D(q_0, 0b_2 \cdots b_n) \notin F$$

which is a contradiction

Case 2: $i > 1$; since $\hat{\delta}_D(q_0, a_1a_2 \cdots a_n) = \hat{\delta}_D(q_0, b_1b_2 \cdots b_n)$ and D is deterministic, we have

$$\hat{\delta}_D(q_0, a_1 \cdots a_{i-1}1a_{i+1} \cdots a_n0^{i-1}) = \hat{\delta}_D(q_0, b_1 \cdots b_{i-1}0b_{i+1} \cdots b_n0^{i-1})$$

From the definition of L, we must have

$$\hat{\delta}_D(q_0, a_1 \cdots a_{i-1}1a_{i+1} \cdots a_n0^{i-1}) \in F$$
$$\hat{\delta}_D(q_0, b_1 \cdots b_{i-1}0b_{i+1} \cdots b_n0^{i-1}) \notin F$$

which is a contradiction

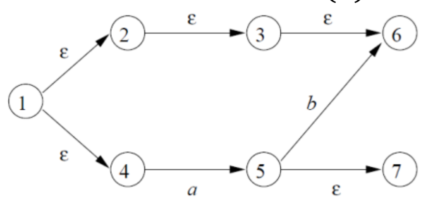
2.3. ε-NFA

NFA with special moves that do not consume the input.
They accept all and only the regular languages.
Easier to design than NFAs.
Same notation of NFA but with $\Sigma \cup \{\varepsilon\}$

ε-closure of a **state** q, $\text{ECLOSE}(q)$: states reachable from q through a sequence of ε

- **Base**: $q \in \text{ECLOSE}(q)$
- **Induction**: $\big(p \in \text{ECLOSE}(q) \text{ and } r \in \delta(p, \epsilon)\big) \Rightarrow r \in \text{ECLOSE}(q)$

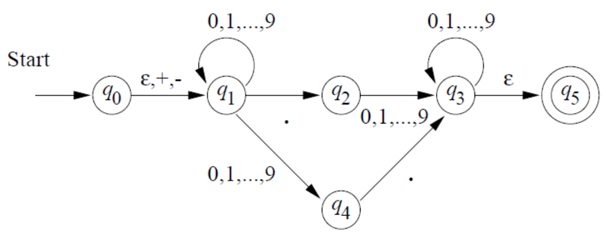
ε-closure of a **set** of states: $\text{ECLOSE}(S) = \bigcup_{q \in S} \text{ECLOSE}(q)$



$\text{ECLOSE}(1) = \{1, 2, 3, 4, 6\}$
$\text{ECLOSE}(\{4, 5\}) = \{4\} \cup \{5, 7\} = \{4, 5, 7\}$

Extended transition function $\hat{\delta}$:

- **Base**: $\hat{\delta}(q, \varepsilon) = \text{ECLOSE}(q)$
- **Induction**: $\boldsymbol{\hat{\delta}(q, xa)} =$
  - $\boldsymbol{\{p_1, \dots, p_k\} = \hat{\delta}(q, x)}$ reachable states from q through path x
  - $\boldsymbol{\{r_1, \dots, r_m\} = \bigcup_{i=1}^{k} \delta(p_i, a)}$ follow the transitions with label a from the states that are reachebly from q through paths labeled x
  - $\boldsymbol{\hat{\delta}(q, xa) = \text{ECLOSE}(\{r_1, \dots, r_m\})}$



We compute $\hat{\delta}(q_0, 5.6)$ for the ϵ-NFA accepting fractional numbers

$\hat{\delta}(q_0, \epsilon) = \text{ECLOSE}(q_0) = \{q_0, q_1\}$

Computation of $\hat{\delta}(q_0, 5)$:

- $\delta(q_0, 5) \cup \delta(q_1, 5) = \varnothing \cup \{q_1, q_4\} = \{q_1, q_4\}$
- $\text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\} = \hat{\delta}(q_0, 5)$

Computation of $\hat{\delta}(q_0, 5.)$:

- $\delta(q_1, .) \cup \delta(q_4, .) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$
- $\text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\} = \hat{\delta}(q_0, 5.)$

Computation of $\hat{\delta}(q_0, 5.6)$:

- $\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \varnothing = \{q_3\}$
- $\text{ECLOSE}(q_3) = \{q_3, q_5\} = \hat{\delta}(q_0, 5.6)$

Language accepted by ε-NFA: $L(E) = \big\{w \big| \hat{\delta}(q_0, w) \cap F \neq \emptyset\big\}$, set of strings w that leads from the initial state to a final state.

Translation ε-NFA to DFA:

- $Q_D = \{S | S \subseteq Q_E, S = \text{ECLOSE}(S)\}$
- $q_D = \text{ECLOSE}(q_0)$
- $F_D = \{S | S \in Q_D, S \cap F_E \neq \emptyset\}$, set of states that contains at least one final state
- For each $a \in \Sigma, \mathrm{S} \in \mathrm{Q_D}$ compute $\delta_D(S, a)$:
  - $S = \{p_1, \dots, p_k\}$
  - $\bigcup_{i=1}^{k} \delta_E(p_i, a) = \{r_1, \dots, r_m\}$
  - $\delta_D(S, a) = \text{ECLOSE}(\{r_1, \dots, r_m\})$

Language L accepted by ε-NFA IIF L accepted by DFA

3. Regular Expressions

Are a declarative way of describing a regular language.

Operations on languages:

- **Union**: $L \cup M = \{w | w \in L \text{ or } w \in M\}$
- **Concatenation**: $L.M = \{w | w = xy, x \in L, y \in M\}$
- **Powers**:
  - $L^0 = \{\epsilon\}$
  - $L^k = L.L^{k-1}$ for $k \geq 1$
- **Klenee closure**: $L^* = \bigcup_{i=0}^{\infty} L^i$???Strings formed by strings that belongs to L????

Operators' precedence: closure, concatenation, union (less important)

Let $L = \{0, 11\}$. In order to construct L^* :

- $L^0 = \{\epsilon\}$
- $L^1 = L = \{0, 11\}$
- $L^2 = L.L^1 = L.L = \{00, 011, 110, 1111\}$
- $L^3 = L.L^2 =$

$$\{000, 0011, 0110, 01111, 1100, 11011, 11110, 111111\}$$

Therefore

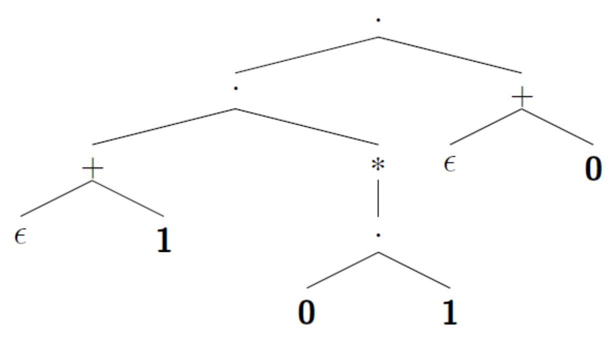
$$L^* = \{\epsilon, 0, 11, 00, 011, 110, 1111, 000, \\ 0011, 0110, 01111, 1100, 11011, 11110, 111111, \ldots\}$$

Definition of **regular expression** E and its generated language $L(E)$:

- **Base**:
  - ϵ is a regular expression, and $L(\epsilon) = \{\epsilon\}$
  - $\emptyset$ is a regular expression, and $L(\emptyset) = \emptyset$
  - If $a \in \Sigma$, then $\boldsymbol{a}$ is a regular expression, and $L(\boldsymbol{a}) = \{a\}$
- **Induction**: If E and F are regular expressions, then
  - $E + F$ is regular expression, and $L(E + F) = L(E) \cup L(F)$
  - EF is regular expression, and $L(EF) = L(E)L(F)$
  - E^* is regular expression, and $L(E^*) = \big(L(E)\big)^*$
  - (E) is regular expression, and $L\big((E)\big) = L(E)$

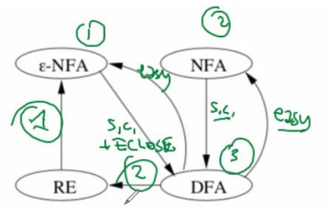
Tree structure of a regular expression:

$$(\epsilon + \mathbf{1})(\mathbf{01})^*(\epsilon + \mathbf{0})$$



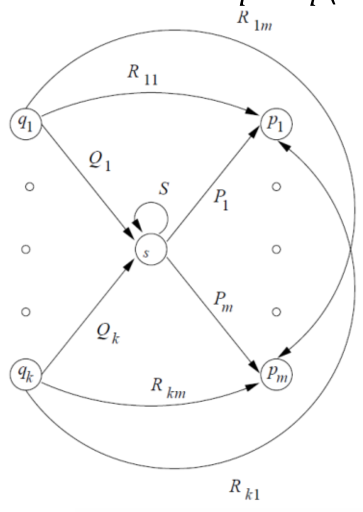
To show that FA and regular expressions are equivalent, we will show that:

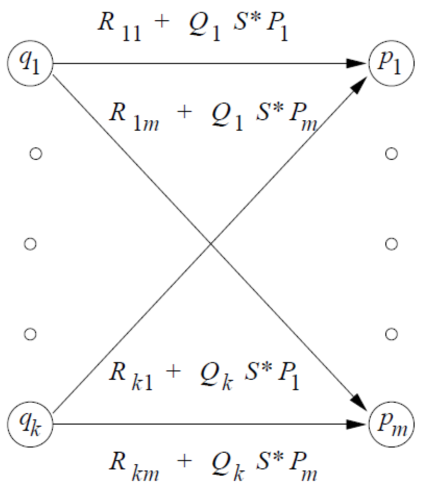
- for each DFA A there is a regular expression R such that $L(R) = L(A)$
- for each regular expression R there is a ϵ-NFA A such that $L(A) = L(R)$

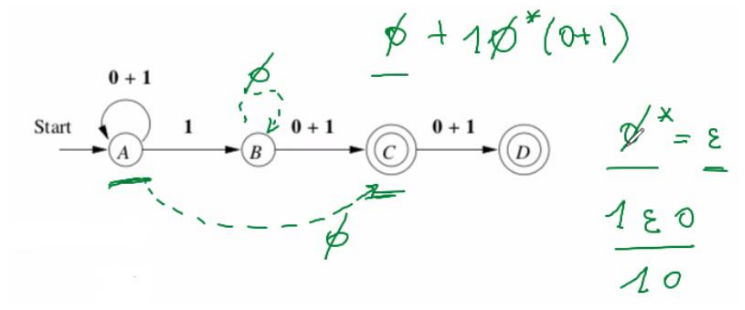


Convert FA in Expressions by state elimination:

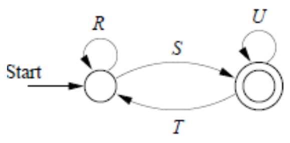
- Replace arc labels with equivalent regular expressions (e.g. 0->**0**, 0,1->**0+1**)
- Delete non-accepting and non-initial states with the **Elimination process**: for each state s to remove, add on arcs $p \to q$ labels of paths $p \to s \to q$: $+\boldsymbol{PS^*Q}$. (If there wasn't recursion on state s or there wasn't the arc $p \to q$ (create it) add only $+PQ$)



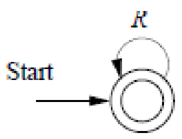




- For each final state q:
  - Remove all states except q and q_0 with the elimination process.
    - If $q \neq q_0$: $E_q = (R + SU^*T)^*SU^*$



    - If $q = q_0$: $E_q = R^*$

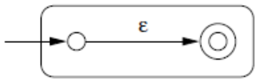


- Final regular expression is the union (+): $E = \sum_{q \in F} E_q$

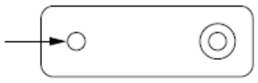
Convert Regular Expression in ϵ-NFA:
For every regular expression R we can construct an ϵ-NFA such that $L(E) = L(R)$
Proof: e construct E with

- only one final state
- no arc entering the initial state
- no arc exiting the final state

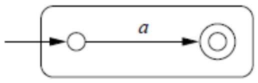
Base: Automata for regular expressions $\epsilon, \emptyset, \boldsymbol{a}$



$\epsilon \Rightarrow L = \{\epsilon\}$

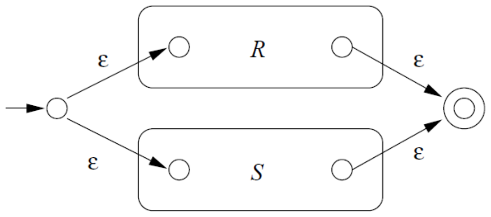


$\emptyset \Rightarrow L = \emptyset$

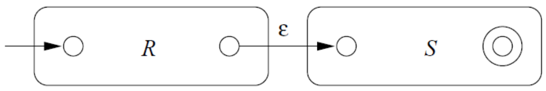


$\boldsymbol{a} \Rightarrow L = a$

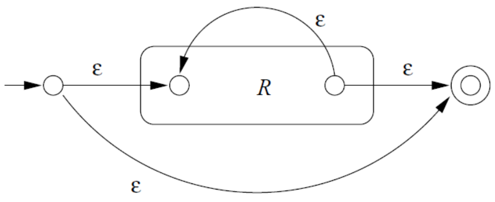
Induction:



$R + S \Rightarrow L(R) \cup L(S)$



$RS \Rightarrow L(R)L(S)$



$R^* \Rightarrow L(R^*)$

Algebraic properties:

- Union Commutative: $L + M = M + L$
- Union Associative: $(L + M) + N = L + (M + N)$
- Union Idempotent: $L + L = L$
- Concatenation Associative: $(LM)N = L(MN)$ (Not commutative)
- Concatenation Distributive left: $L(M + N) = LM + LN$
- Concatenation Distributive right: $(M + N)L = ML + NL$
- Closure:
    - $(L^*)^* = L^*$
    - $\emptyset^* = \epsilon$
    - $\epsilon^* = \epsilon$
    - $L^+ = LL^* = L^*L$
    - $L^* = L^+ + \epsilon$
    - $L? = \epsilon + L$
- $\emptyset \cup L = L \cup \emptyset = L$
- $\epsilon L = L\epsilon = L$
- $\emptyset L = L\emptyset = \emptyset$ Annihilator

4. Properties of Regular Languages

4.1. Pumping Lemma

Used to show that some languages are **not** regular.

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:

- $y \neq \epsilon$
- $|xy| \leq n$
- $\forall k \geq 0, xy^k z \in L$

We can always find a non-empty string y, not too distant from the start of w, to replicate or delete without exiting from L.

How to use it:

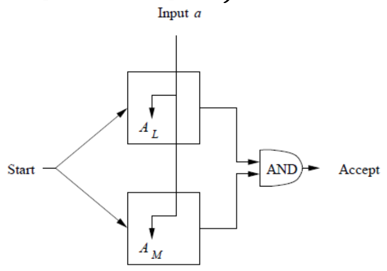
- Suppose L regular. Then $\exists n \in \mathbb{N}$…
- Invent a $w \in L$ with $|w| \geq n$. Make some symbols repeats n times.
- Decompose $w = xyz$. Respecting $y \neq \epsilon$ and $|xy| \leq n$
- If I can choose a k so that $xy^k z \notin L$, then the language is not regular.

4.2. Closure properties

Used to create complex automata starting from other languages.

Returns Regular languages:

- Union: $L \cup M$ (but split a regular in 2 can be not regular)
- Intersection: $L \cap M$
  Intersection Automation: $A = (Q_L \times Q_M, \Sigma, \delta_{L\cap M}, (q_L, q_M), F_L \times F_M)$
  - States: pairs of states of A_L and A_M
  - Initial state: pair of initial states of A_L and A_M
  - Final states: pairs of finial states of A_L and A_M, because the automata accept only if both automatons accept.
  - $\delta_{L\cap M}\big((p,q),a\big) = \big(\delta_L(p,a), \delta_L(q,a)\big)$



- Complement: $\bar{L} = \Sigma^* - L$
- Difference: $L - M$
- Inversion: $L^R = \{w^R | w \in L\}$
- L^*
- Concatenation: $L.M$
- Homomorphism: $h(L) = \{h(w) | w \in L\}$ function that substitute a symbol with a string. $h: \Sigma \to \Delta^*, h(w) = \begin{cases} \epsilon, & w = \epsilon \\ h(x)h(a), & w = xa, x \in \Sigma^*, a \in \Sigma \end{cases}$

4.3. Conversion's complexities, decide if $L = \emptyset$ and if $w \in L$

Conversion's complexities: (n: number of states FA or number of operators in EXPR)

- ϵ-NFA in DFA: $O(n^3 s)$ with s reachable states, usually s is at most 2^n
- DFA in NFA: $O(n)$
- DFA in expression: $O(n^3 4^n)$. From ϵ-NFA $O\left(n^3 4^{n^3 2^n}\right)$
- Expression in ϵ-NFA: $O(n)$

Decide if a language is empty for an FA: if it exists a path from the initial state to a final state, the language is not empty.

- **Base**: The initial state is reachable
- **Induction**: If q is reachable and there exists a transition from q to p, then p is reachable.

Time: $O(n^2)$

Decide if a language is empty for an expression: induction on the structure of E

- **Base:**
  - If $E = \epsilon$ or $E = \boldsymbol{a}$, then $L(E)$ is non-empty
  - If $E = \emptyset$, then $L(E)$ is empty
- **Induction**:
  - $E = F + G$, then $L(E)$ is empty iff both $L(F)$ and $L(G)$ are empty
  - $E = F.G$, then $L(E)$ is empty iff either $L(F)$ or $L(G)$ are empty
  - $E = F^*$, then $L(E)$ is not empty, since $\epsilon \in L(E)$

Decide if a string w is in a language L:

- If L is represented by a DFA A: we simulate the input of the string in the DFA. If it ends in a final state, then the string is in the language.
  Time: $O(n)$ with $n = |w|$
- If L is represented by an NFA: simulate. $O(ns^2)$, s=#states A
- If L is represented by a ϵ-NFA: simulate. $O(ns^3)$,
- Se L è rappresentato da un Espressione di dimensione s: la si converte in un ϵ-NFA. $O(ns^3)$

4.4. Automata minimization

Equivalent states: $p \equiv q \iff \forall w \in \Sigma^*: \hat{\delta}(p,w) \in F \iff \hat{\delta}(q,w) \in F$
If for every string w, $\hat{\delta}(p,w)$ is a final state IIF $\hat{\delta}(q,w)$ is a final state.
Distinguishable states: $p \not\equiv q \iff \exists w: \hat{\delta}(p,w) \in F$ and $\hat{\delta}(q,w) \notin F$ or viceversa
If exist a string w that brings to p to a final state and q not (or viceversa).

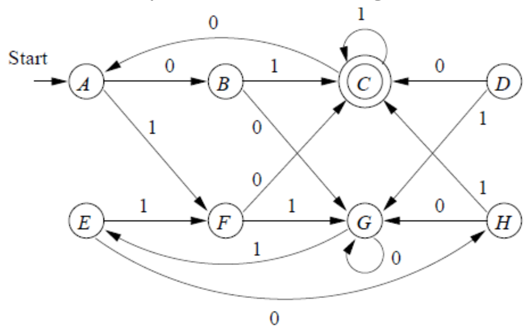
Transitivity of the equivalence: If $p \equiv q$ and $q \equiv r$, then $p \equiv r$

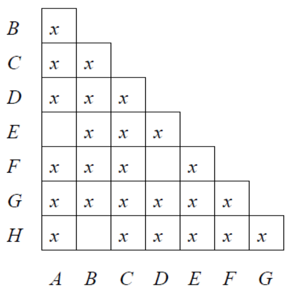
Equivalence algorithm between states of DFA:

- **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 $p \not\equiv q$

If 2 states are not distinguished by the algorithm, then they are equivalent.

- initialize table with pairs that are distinguishable by string ϵ: put the X in all cells of final states (because the other ones are not).
- for all not yet visited pairs, try to distinguish them using one symbol string: if you reach a pair of already distinguishable states, then update table
- iterate until no new pair can be distinguished.



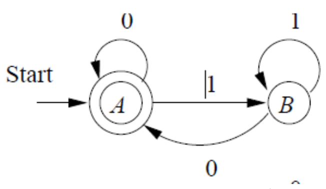


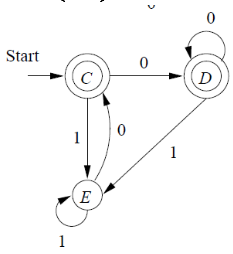
	A	B	C	D	E	F	G
B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x

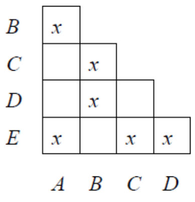
Verify if two languages are equal: $O(n^4)$

- Convert them in DFA
- Build a DFA with the union of both (it will have 2 initial states)
- Verify with the equivalence algorithm if the 2 initial states are distinguishable. If they are not distinguishable then they are equivalent

There exist also an algorithm in $O(n^2)$. Look in the book.



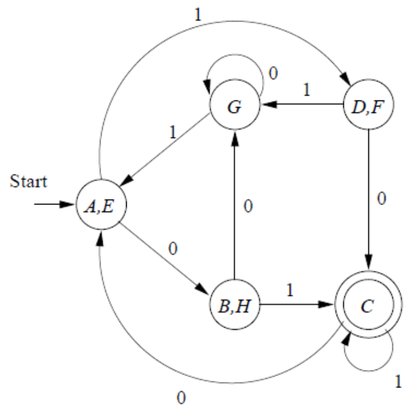




	A	B	C	D
B	x			
C		x		
D		x		
E	x		x	x

DFA Minimization: $A = (Q_A, \Sigma, \delta_A, q_{0_A}, F_A)$

- Determine the pairs of equivalent states with the equivalence algorithm
- Partition Q_A in groups of equivalent states:
  - Q_B: are the groups
  - Initial state q_{0_B}: group that contains initial state of A
  - Final states F_B: groups that contains final states of A
  - δ_B: for each group in Q_B look in the graph of A where the arcs go.



5.Context-free grammars and Languages

5.1. Context-free grammars (CFG)

Definition: $G = (V, T, P, S)$

- V finite set of variables (**nonterminals**)
- T finite set of terminal symbols (**terminals**)
- P finite set of **productions**. Production: $A \to \alpha$
  - Head: $A \in V$
  - Production symbol: $\to$
  - Body: string $\alpha \in (V \cup T)^*$
  - Compact notation: $A \to \alpha_1, \dots A \to \alpha_n$ written as $A \to \alpha_1| \dots |\alpha_n$
- $S \in V$ variable (**initial symbol**)

Es: T= $\{+, *, (,), a, b, 0, 1\}$

P=

1. $E \to I$	6. $I \to b$
2. $E \to E + E$	7. $I \to I\,a$
3. $E \to E * E$	8. $I \to I\,b$
4. $E \to (E)$	9. $I \to I\,0$
5. $I \to a$	10. $I \to I\,1$

2 Ways to define a language of a CFG:

- Recursive inference: use production from the body to the head
- **Derivation**: use production from the head to the body.
  Expand the initial symbol with one of its productions, then recursively expand one of it's variables with one of it's production till derive a string.

Derivation step: $\alpha A\beta \underset{G}{\Rightarrow} \alpha\gamma\beta$ (substitute variable A with one of its productions)

Multiple steps (0 or more): $\alpha A\beta \overset{*}{\underset{G}{\Rightarrow}} \alpha\gamma\beta$

- Base: $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$ with $\alpha \in (V \cup T)^*$
- Induction: If $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ e $\beta \underset{G}{\Rightarrow} \gamma$ then $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$

Leftmost$\underset{lm}{\Rightarrow}$/**Rightmost**$\underset{rm}{\Rightarrow}$ **derivation**: we always substitute the variable on the leftmost/rightmost
Every terminal string has leftmost=rightmost derivation.

Generated Language CFL: $L(G) = \left\{w \in T^* \middle| S \overset{*}{\underset{G}{\Rightarrow}} w\right\}$

Sentential form: derivation from the initial symbol. $\alpha \in (V \cup T)^*$

- **Sentential form** α: $S \overset{*}{\underset{G}{\Rightarrow}} \alpha$
- **Left Sentential form** α: $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$
- **Right Sentential form** α: $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$

$L(G)$ contains sentential forms that are in T^*

Prove that a Grammar generates a Language:

Ind. hyp.: for each variable A in the CFG, define some property P_A of all strings w such that $A \overset{*}{\underset{G}{\Rightarrow}} w$.

Examples of Properties for each variable A (start with the ones nearest to terminals, not S):

- $\forall w \in \Sigma^*, P_A(w)$ holds true IFF w is a sequence of n>=1 "1" followed by m>n "0"
- …

Prove for every property P_A: $\forall w \in \Sigma^*, P_A(w)$ holds true IIF $A \overset{*}{\Rightarrow} w$

- **IF part:** IF $A \overset{*}{\Rightarrow} w$ THEN $P_A(w)$

  Induction on the length of derivation $A \overset{*}{\Rightarrow} w$ (number of steps)
  - **Base**: shortest derivation (that led to terminal) of $A \overset{*}{\Rightarrow} w$ is $A \overset{1}{\Rightarrow} \dots \overset{1}{\Rightarrow} 1$, we have that w is a sequence composed by …, so $P_A(1)$ holds
  - **Induction:** assume derivation length $>$…

    If derivation starts with the production $A \rightarrow 1A$ … than we can write $A \overset{1}{\Rightarrow} 1A \overset{*}{\Rightarrow} 1x = w$ and $A \overset{*}{\Rightarrow} x$ holds.

    Apply inductive hypothesis to $A \overset{*}{\Rightarrow} x$ obtaining $P_A(x)$ holds true, that means x is a sequence composed of … So $w = \cdots x$ is composed of … So $P_A(w)$ true.

  If there are k parts in the inductive enunciate, in some parts of the induction we use mutual induction:
  - Focus on the first production of the derivation:
$$\begin{aligned} A &\Rightarrow B_1 \dots B_k \\ &\overset{*}{\Rightarrow} x_1 B_2 \dots B_k \\ &\vdots \\ &\overset{*}{\Rightarrow} x_1 \dots x_k = w \end{aligned}$$
  - Use the inductive hypothesis on $B_i \overset{*}{\Rightarrow} x_i$ to obtain that $P_{B_i}(x_i)$ holds for each i
  - Use P_A definition to show that $P_A(w)$ is true
- **ONLY IF part:** IF $P_A(w)$ THEN $A \overset{*}{\Rightarrow} w$

  Induction on length of $|w|$:
  - **Base**: $|w|$ minimum length (0 if there is ϵ, 1 if there is 1 symbol, 2…). Check every possible value of w of that length:
    - case w="0" then $P_A(w)$ is true and the required derivation is $A \overset{1}{\Rightarrow} 0$
    - case w="1" then $P_A(w)$ is false. So the implication is true.
  - **Induction**: assume $|w| >$ min length. Consider $P_A(w)$ true and we prove that $A \overset{*}{\Rightarrow} w$. Decompose w…, if w starts with … . We apply inductive hypothesis, and conclude $A \overset{*}{\Rightarrow} x$. Using production $A \rightarrow 1A$ we can write $A \overset{1}{\Rightarrow} 1A \overset{*}{\Rightarrow} 1x = w$

  If there are k parts in the inductive enunciate, in some parts of the induction we use mutual induction:
  - Using P_A definition, choose a factorization $w = x_1 \dots x_k$ such that $P_{B_i}(x_i)$ holds for each i
  - Use the inductive hypothesis on $P_{B_i}(x_i)$ to obtain $B_i \overset{*}{\Rightarrow} x_i$ for each i
  - Choose a production $A \rightarrow B_1 \dots B_k$ and obtain
$$\begin{aligned} A &\Rightarrow B_1 \dots B_k \\ &\overset{*}{\Rightarrow} x_1 B_2 \dots B_k \\ \dots &\overset{*}{\Rightarrow} x_1 \dots x_k = w \end{aligned}$$

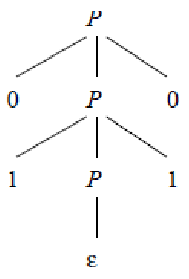
5.2. Parse Tree

Tree representation of a derivation. Represent the syntactic structure of a sentence according to the grammar.

Construction:

- each internal node is labeled with a variable in V
- each leaf node is labeled with a variable, a terminal or ϵ. $V \cup T \cup \{\epsilon\}$
  each leaf labeled with ϵ is the only child of its parent
- if an internal node is labeled A and its children (from left to right) are labeled $X_1, \dots, X_k$, then $A \rightarrow X_1 \dots X_k \in P$
  X can be ϵ only if $A \rightarrow \epsilon \in P$

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



Yield: is the string obtained by reading the leaves from left to right.
Complete parse trees:

- the yield is a string of terminal symbols
- the root is labeled by the initial symbol

The set of yields of all complete parse trees is the language generated by the CFG.

Following statements are equivalent: $A \in V, w \in (V \cup T)^*$

- $A \overset{*}{\Rightarrow} w$
- $A \underset{lm}{\overset{*}{\Rightarrow}} w$
- $A \underset{rm}{\overset{*}{\Rightarrow}} w$
- there exists a parse tree for G with root label A and yield w

We can always **compose 2 derivations**: $A \overset{*}{\Rightarrow} \alpha B \beta$ e $B \overset{*}{\Rightarrow} \gamma$ into a single derivation $A \overset{*}{\Rightarrow} \alpha\gamma\beta \overset{*}{\Rightarrow} \alpha\gamma\beta$
Given $A \Rightarrow X_1 \dots X_k \overset{*}{\Rightarrow} w$ we can always **factorize** w in $w_1 \dots w_k$ such that $X_i \overset{*}{\Rightarrow} w_i, 1 \leq i \leq k$
Substring w_i can be identified from derivation $A \overset{*}{\Rightarrow} w$ by considering only those derivation steps that rewrite X_i

5.3. Ambiguity and relation with Regular languages

Some strings might have more than one parse tree. And a parse tree can have several derivations.

Grammar G is **ambiguous:** if there exists a string in $L(G)$ with more than one parse tree
Grammar G is **unambiguous:** If every string in $L(G)$ has only one parse tree.
There is no way to remove the ambiguity.
A terminal string w had 2 distinct parse trees IFF w has 2 different derivations to the left of S.
Language L is **inherently ambiguous**: when every CFG such that $L(G) = L$ is ambiguous.

A regular language is always a CFL.
From a regular expression or from an FA we can always construct a CFG generating the same language.
CFGs can simulate FAs or regular expressions.

From regular expression to CFG: given E, use variable E (start symbol) and a variable for each subexpression of E and do structural induction:

- $E = \boldsymbol{a}$: add production $E \to a$
- $E = \epsilon$: add production $E \to \epsilon$
- $E = \emptyset$: the production set is empty
- $E = F + G$: add production $E \to F \mid G$
- $E = FG$: add production $E \to FG$
- $E = F^*$: add production $E \to FG \mid \epsilon$

Regular expression : $\mathbf{0}^*\mathbf{1}(\mathbf{0}+\mathbf{1})^*$

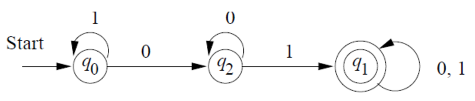
CFG :

$$
\begin{aligned}
E &\to AR \\
R &\to BC \\
A &\to 0A \mid \epsilon \\
B &\to 1 \\
C &\to DC \mid \epsilon \\
D &\to 0 \mid 1
\end{aligned}
$$

From FA to CFG: $A = (Q, \Sigma, \delta, q_0, F)$ --> $G = (V, T, P, S)$

- V = create a variable Q_i for each state in Q
- $S = Q_0$ is the variable of q_0
- P = for each transition from q_i to q_j under symbol a, add the production $Q_i \to aQ_j$
- If q_k is a final state, add production $Q_k \to \epsilon$
- $T = \Sigma$

Automaton :



CFG :

$$
\begin{aligned}
Q_0 &\to 1Q_0 \mid 0Q_2 \\
Q_2 &\to 0Q_2 \mid 1Q_1 \\
Q_1 &\to 0Q_1 \mid 1Q_1 \mid \epsilon
\end{aligned}
$$

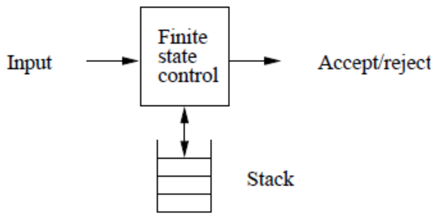
6. Push-Down Automata (PDA)

6.1. Definition

Is a ϵ-NFA with a stack (LIFO) that can memorize symbols. Recognize all and only the CFL.

PDA: $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

- Q: finite set of states
- Σ: finite set of symbols
- Γ: stack alphabet. Finite set of symbols that can be inserted into the stack.
- $\delta: Q \times \Sigma \cup \{\epsilon\} \times \Gamma \to 2^{Q\times\Gamma^*}$: transition:
  - Input the triplet (q, a, X) with $q \in Q, a \in \Sigma \cup \{\epsilon\}, X \in \Gamma$
  - Output: finite set of pairs (p, γ) where p is the new state and γ is the string that replace X on the top of the stack.

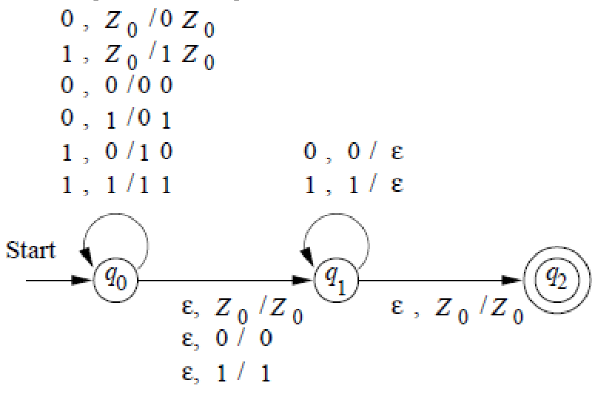


- q_0: initial state
- $Z_0 \in \Gamma$: initial stack symbol.
- $F \subseteq Q$: set of final states.

Transition the PDA:

- Consumes a single symbol from the input or is an ϵ-transition
- Update the current state
- replaces the top-most symbol of the stack with a string of symbols, including ϵ that is equivalent to popping the element on the top.

Transition **Graphic notation**: for $(p, \alpha) \in \delta(q, a, X)$ we label the arc from p to q with $a, X/\alpha$



Computation: is a sequence of "configurations" of the automaton obtained one from the other by consuming an input symbol or else by reading ϵ

Instantaneous description (ID): (q, w, γ)

- q: state
- w: remaining input
- γ: stack content

Computational move $\vdash$: binary relation between instantaneous descriptions

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

The computation is represented by the Closure $\overset{*}{\underset{P}{\vdash}}$: zero or more PDA moves

Properties of computations: if an ID sequence is valid (relation $\vdash$) for a PDA P:

- then so is the sequence obtained by adding any string to the tail of the input
- then so is the sequence obtained by adding any string to the bottom of the stack
- and some tail of the input is not consumed, then so is the sequence obtained by removing that tail in every ID in the sequence

It means that **symbols that are not read/consumed by the PDA do not affect the computation.**

- $\forall w \in \Sigma^*, \gamma \in \Gamma^*: (q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta) \Longrightarrow (q, xw, \alpha\gamma) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta\gamma)$
  If $\gamma = \epsilon$ we have property 1 and if $w = \epsilon$ we have the 2
- $\forall w \in \Sigma^*: (q, xw, \alpha) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta) \Longrightarrow (q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$

6.2. Accepted language

Language accepted by final state: $\boldsymbol{L(P)} = \left\{ w \middle| (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \alpha), q \in F \right\}$

Starting from the initial ID, P consumes the input w till it reaches a final state. The stack does not necessarily need to be empty at the end of the computation.

Language accepted by empty stack: $\boldsymbol{N(P)} = \left\{ w \middle| (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \epsilon) \right\}$

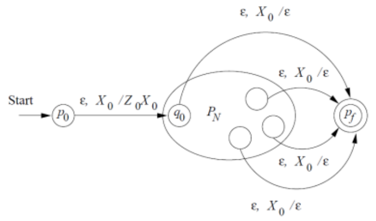
$N(P)$ is the set of the inputs w that P consumes making the stack empty.
$L(P) = N(P)$

From empty stack to final state:
If $L = N(P_N)$ then exist a PDA P_F such that $L = L(P_F)$. Proof:

$$P_F = \left(Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0 X_0, \{p_f\}\right)$$

with δ_F:

- $\delta_F(p_0, \epsilon, X_0) = \{(q_0, Z_0 X_0)\}$
- $\delta_F(q, a, Y) = \delta_N(q, a, Y)\ \forall q \in Q, a \in \Sigma \cup \{\epsilon\}, Y \in \Gamma$
- $\delta_F(q, \epsilon, X_0)$ contains $(p_f, \epsilon)\ \forall q \in Q$

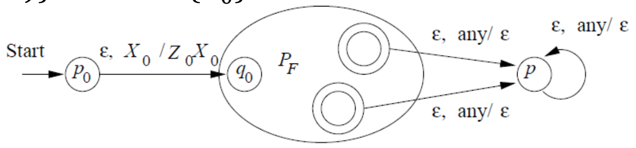


From final state to empty stack:
If $L = L(P_F)$ then exist a PDA P_N such that $L = N(P_N)$. Proof:

$$P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$$

with δ_N:

- $\delta_N(p_0, \epsilon, X_0) = \{(q_0, Z_0 X_0)\}$
- $\delta_N(q, a, Y) = \delta_F(q, a, Y). \forall q \in Q, a \in \Sigma \cup \{\epsilon\}, Y \in \Gamma$
- $(p, \epsilon) \in \delta_N(q, \epsilon, Y). \forall q \in F, Y \in \Gamma \cup \{\mathrm{X}_0\}$
- $\delta_N(q, \epsilon, Y) = \{(p, \epsilon)\}\ \forall Y \in \Gamma \cup \{\mathrm{X}_0\}$



We now prove $N(P_N) = L(P_F)$

(part $\subseteq$) By inspecting the diagram

(part $\supseteq$) Let $w \in L(P_F)$. Then

$$(q_0, w, Z_0) \overset{*}{\underset{F}{\vdash}} (q, \epsilon, \alpha)$$

for some $q \in F, \alpha \in \Gamma^*$

Since $\delta_F \subseteq \delta_N$, and from a previous theorem stating that X_0 can be added to the bottom of the stack, we have

$$(q_0, w, Z_0X_0) \overset{*}{\underset{N}{\vdash}} (q, \epsilon, \alpha X_0)$$

Then P_N can compute

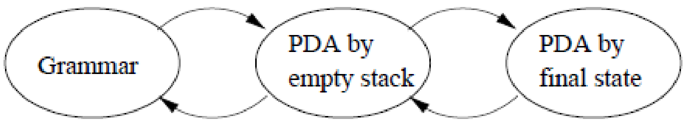
$$(p_0, w, X_0) \underset{N}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{N}{\vdash}} (q, \epsilon, \alpha X_0) \overset{*}{\underset{N}{\vdash}} (p, \epsilon, \epsilon)$$

□

6.3. Equivalence of PDAs e CFGs

The following statements are equivalent

- L is generated by a CFG
- L is accepted by a PDA by empty stack
- L is accepted by a PDA by final state



From CFG to PDA: $G = (V, T, R, S)$ -----> $P = (\{q\}, T, V \cup T, \delta, q, S)$
with δ:

- $\delta(q, \epsilon, A) = \{(q, \beta) | (A \rightarrow \beta) \in R\}$ for each $A \in V$
- $\delta(q, a, a) = \{(q, \epsilon)\}$ for each terminal $a \in T$

So $L(G) = L(P)$

From PDA to CFG: $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ ----> $G = (V, T, R, S)$

- $V =$
  - initial symbol S
  - the symbol $[pXq]$ for each $p, q \in Q$, $X \in \Gamma$
- Productions $R =$
  - The production $S \rightarrow [q_0 Z_0 p]$ for each $p \in Q$
  - If $\delta(q, a, X)$ contains $(r, Y_1 \dots Y_k)$ where $a \in \Sigma \cup \{\epsilon\}$ and $k \in \mathbb{N}$
    Then for each sequence of states $r_1, \dots, r_k \in Q$, R contains the production
    $$[qXr_k] \rightarrow a[rY_1r_1] \dots [r_{k-1}Y_kr_k]$$

7. Properties of CFL

7.1. Normal forms of CFG

7.1.1. Eliminate useless symbols

Given a CFG $G = (V, T, P, S)$. A symbol $X \in V \cup T$ is:

- **reachable** if there exists a derivation $S \overset{*}{\Rightarrow} \alpha X \beta$
- **generating** if there exists a derivation $S \overset{*}{\Rightarrow} w$ for some $w \in T^*$
- **useful** if it is reachable and generating, so if there exist a derivation $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$

Compute generating symbols: $g(G)$

- **Base**: each symbol in T is a generating
- **Induction**: given $A \rightarrow \alpha$, A is a generator if each symbol of α is a generator

Compute reachable symbols: $r(G)$

- **Base**: S is reachable
- **Induction**: if A is reachable, then every production with A as head is reachable

Eliminate useless symbols:

- Build $G_1 = (V_1, T_1, P_1, S)$ by eliminating from G all non-generating symbols and all productions in which they appear.
- Build $G_2 = (V_2, T_2, P_2, S)$ by eliminating from G_1 all non-reachable symbols (in G_1) and all productions in which they appear.

So $L(G_2) = L(G)$

7.1.2. Eliminate ϵ-Productions

If L is a context-free language, then there is a CFG without ϵ-productions that generates $L \backslash \{\epsilon\}$

Variable A is **nullable** if $A \overset{*}{\Rightarrow} \epsilon$

Compute nullable symbols: $n(G)$

- **Base**: if $A \rightarrow \epsilon$, then A is nullable
- **Induction**: if there exist a production $B \rightarrow C_1 \dots C_k$ in which every variable is nullable, then B is nullable.

Eliminate ϵ-productions:

$G_1 = (V, T, P_1, S)$ with P_1 obtained:

- Productions $A \rightarrow \epsilon$ are not inserted in P_1
- For each production $A \rightarrow X_1 \dots X_k$ of P, with $k \geq 1$, if there are m X_i nullable, insert in P_1 all the 2^m versions of the production, where the X_i are present or absent in all possible combinations.
- Exception: if $m = k$, not insert in P_1 the production $A \rightarrow \epsilon$

So $L(G_1) = L(G) - \{\epsilon\}$

7.1.3. Eliminate unary productions

Unary production: $A \rightarrow B$ in which both A and B are variables in V. ($A \rightarrow a, A \rightarrow \epsilon$ are not unary productions)

Unary pair (A, B): if $A \overset{*}{\Rightarrow} B$ using only unary productions

Compute unary pairs: $u(G)$

- **Base**: (A, A) is a unary pair for each variable A
- **Induction**: if (A, B) is a unary pair, $B \rightarrow C$ in which C is a variable, then (A, C) is a unary variable.

Eliminate unary productions: for each unary pair $(A, B) \in u(G)$, for each $B \rightarrow \alpha$ that is a NON unary production of P, add to P_1 the productions $A \rightarrow \alpha$. ($A \rightarrow \alpha$ might not be present before)

7.1.4. Chomsky normal form (CFN)

A CFG is **in Chomsky normal form (CNF)**, if its productions have one of the two forms:

- $A \rightarrow BC$, with $A, B, C \in V$
- $A \rightarrow a$ with $A \in V, a \in T$

and the grammar does not have useless symbols.

Every CFL without the empty string ϵ can be generated by CNF grammar.

CFG simplification: (in order)

- elimination of ϵ-productions
- elimination of unary productions
- elimination of useless symbols

The resulting grammar has productions of the form:

- $A \rightarrow a$
- $A \rightarrow \alpha$, with $\alpha \in (V \cup T)^*, |\alpha| \geq 2$

From simplified CFG to CFN:

- right-hand sides of length ≥ 2 must only have variables.
  For each production with right-hand side α such that $|\alpha| \geq 2$ and for each occurrence in α of $a \in T$:
  - construct a new production $A \rightarrow a$ (A new variable)
  - use A in place of a in α
- right-hand sides of length ≥ 2 must be decomposed into chains of productions with only two variables in their right-hand side.
  For each production of the form $A \rightarrow B_1 B_2 \dots B_k, k \geq 3$
  - introduce new variables $C_1, \dots, C_{k-2}$
  - replace the production with the chain of new productions: $(A \rightarrow B_1\, C_1), (C_1 \rightarrow B_2 C_2), \dots, (C_{k-3} \rightarrow B_{k-2} C_{k-2}), (C_{k-2} \rightarrow B_{k-1}\, B_k)$

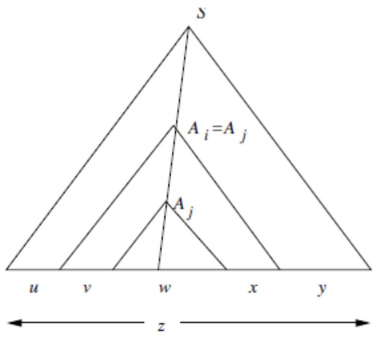
Greibach normal form (GNF): if every production has the form $A \rightarrow a\alpha$ with $a \in T, \alpha \in V^*$

- every nonempty CFL with non-empty strings has only one GNF grammar
- a grammar in GNF generates a string of length n in exactly n steps
- if we turn a GNF grammar into a PDA, we get an automaton without ϵ-transitions

7.2. Pumping lemma for CFL

Used to prove that a Language is not a CFL.

Given a **Parse Tree** T of a string w generated by a CNF. If the longest path in T has n arcs, then $|w| \leq 2^{n-1}$.



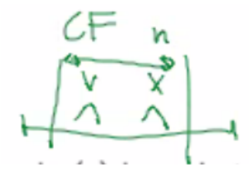
Let L be some CFL. Then $\exists n \in \mathbb{N}$ such that if $z \in L$ with $|z| \geq n$, we can factorize $z = uvwxy$ under the following conditions:

- $|vwx| \leq n$
- $vx \neq \epsilon$ it means that $|v| + |x| \geq 1$
- $\forall k \geq 0, uv^kwx^ky \in L$

It means that, in each sufficiently long string of a CFL we can find two substrings next to each other that

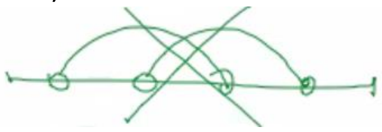
- can be eliminated
- can be iterated (synchronously)

still resulting in strings of the language.



Consequences of the pumping lemma:

- A CFL cannot display crossing pairs with the same arbitrary number of symbols. (Eg. $L = \{0^i1^j2^i3^j | i,j \geq 1\}$ is NOT a CFL)



- A CFL cannot copy strings of arbitrary length if those are defined in an alphabet with more than 1 symbol. (Eg. $L = \{ww | w \in \{0,1\}^*\}$ is NOT a CFL)

How to use it:

- Suppose L CFL. Then $\exists n \in \mathbb{N}$…
- Invent a $z \in L$ with $|z| \geq n$. Make some symbols repeats n times.
- Decompose $z = uvwxy$. Respecting $|vwx| \leq n$ and $vx \neq \epsilon$
- If I can choose a k so that $uv^kwx^ky \notin L$, then the language is not CFL.
    - For any possible choice of v and x there must be a k that falsify the theorem.

Common Not CFL languages:

- $L = \{0^i1^j2^i3^j | i,j \geq 1\}$
- $L = \{a^nb^nc^n | n \geq 0\}$
- $L = \{w | \#_a(w) = \#_b(w) = \#_c(w)\}$

Common NOT REG languages:

- $L = \{a^nb^n | n \geq 0\}$
- $L = \{w | \#_a(w) = \#_b(w)\}$

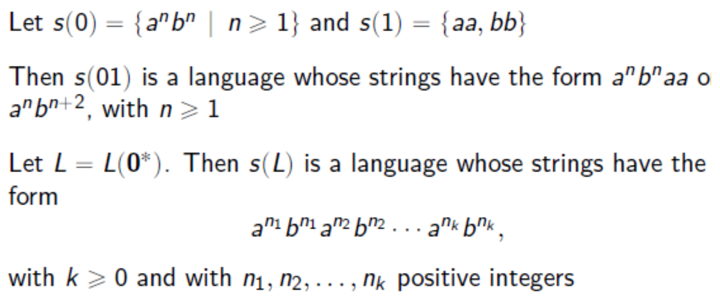
Other tricks:

- First check if you can build a grammar or a PDA that accept the language.
- If the form of the string is not given (not a^nb^n…), do the intersection with a regular, then use pumping on the new language and for closure property also the initial is not CFL.

7.3. Closure properties for CFL

The following properties (substitution, union, …) between CFL, returns CFL.

- **Substitution**: every symbol in the strings of a language is substituted with another language.
    - $s: \Sigma \to 2^{\Delta^*}$, with Σ and Δ finite alphabets, $s(a)$ a CFL
    - Given $w \in \Sigma^*$, $w = a_1 \dots a_n$, $a_i \in \Sigma$
    - $s(w) = s(a_1).s(a_2).\cdots.s(a_n)$
    - $\boldsymbol{s(L)} = \bigcup_{\boldsymbol{w \in L}} \boldsymbol{s(w)}$ union of the $s(w)$ for all the strings $w \in L$. $s(L)$ is a CFL
    - Eg.



Let $s(0) = \{a^n b^n \mid n \geqslant 1\}$ and $s(1) = \{aa, bb\}$

Then $s(01)$ is a language whose strings have the form $a^n b^n aa$ o $a^n b^{n+2}$, with $n \geqslant 1$

Let $L = L(\mathbf{0}^*)$. Then $s(L)$ is a language whose strings have the form

$$a^{n_1} b^{n_1} a^{n_2} b^{n_2} \dots a^{n_k} b^{n_k},$$

with $k \geqslant 0$ and with $n_1, n_2, \dots, n_k$ positive integers

- **Union**: $L_1 \cup L_2$ (but you can't split a CFL in 2)
- **Concatenation**: $L_1 L_2$
- **Kleene closure ($\boldsymbol{L^*}$) and positive closure ($\boldsymbol{L^+}$)**
- **Inversion**: L^R
- **Intersection with regular language:** $L \cap R$ is a CFL
- **Difference with regular**: $L - R$ is a CFL
- $L_1 \cap L_2$ may fall outside of CFL:
  $L_1 = a^n b^n c^i \in CFL, L_2 = a^i b^n c^n \in CFL\ L = L_1 \cap L_2 = a^n b^n c^n \notin CFL$
- $\overline{L} = \Sigma^* - L$ may fall outside of CFL.
  If it was true then: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ but intersection is not closed.
- $L_1 - L_2$ may fall outside of CFL

Tell if CFL is closed under a property $\boldsymbol{P}$:

- Prove that for each $L \in CFL, P(L) \in CFL$
- Or show counterexample

7.4. Computational properties

n the length of the entire representation of a PDA or a CFG
Complexity of conversions:

- Conversion from CFG to PDA: $O(n)$
- Conversion from PDA accepting by final state to accepting by empty stack: $O(n)$ viceversa
- **Conversion from PDA to CFG**: $O(n^3)$
- **Conversion from CFG to CNF**: $O(n^2)$

We can compute in time $O(n)$:

- the set of reachable symbols $r(G)$
- the set of generating symbols $g(G)$
- the elimination of useless symbols from a CFG
- the set of nullable symbols $n(G)$
- the elimination of ϵ-productions using a preliminary binarization of the grammar
- the replacement of terminal symbols with variables (first transformation for CNF)
- the reduction of production with right-hand side length larger than 2 (second transformation for CNF)

We can compute in time $O(n^2)$:

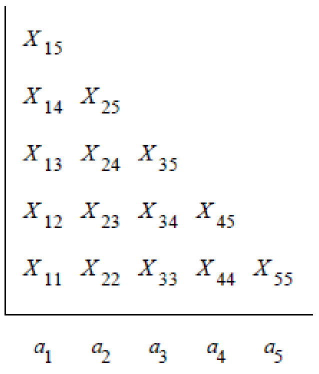
- the set of unary symbols $u(G)$
- the elimination of unary productions from a CFG

7.5. Decision problems for CFL

Check if a CFL is empty: $O(n)$
Using a modified version of the representation of G: $L(G) = \emptyset$ IFF S is not a generator

Check if a string $w \in L(G)$ for a fixed CFG G: $O(n^3)$
Given a grammar in Chomsky normal form.
We can generate all the parse trees of G with $2n - 1$ nodes and test whether some tree yields w.
But with dynamic programming:

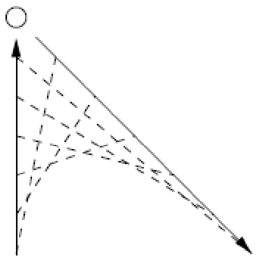
- Assume $w = a_1 \dots a_n$
- construct a triangular parse table where cell X_{ij} contains all variables A such that $A \overset{*}{\Rightarrow} a_i a_{i+1} \dots a_j$
- Iteratively construct the parse table, one row at a time and from bottom to top.
- First row is populated with the base case, the other with induction
- 1 row: strings of length 1. 2 row: strings length 2, ….



X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
a_1	a_2	a_3	a_4	a_5

DP Algorithm:

- **Base**: $X_{ii} = \{A | (A \rightarrow a_i) \in G\}$ first row
- **Induction**: X_{ij} = every A such that
  - $i \leq k < j$
  - $B \in X_{ik}$
  - $C \in X_{k+1,j}$
  - $(A \rightarrow BC) \in G$



To populate the X_{ij} we need to check at most n pairs of previously built cells of the parse table.

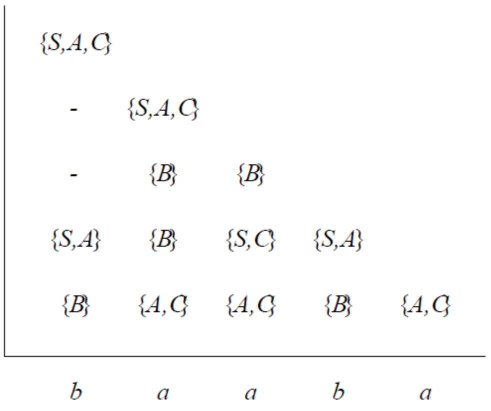
$$(X_{ii}, X_{i+1,j}), (X_{i,i+1}, X_{i+2,j}), \dots, (X_{i,j-1}, X_{jj})$$

Example:

Let G be a CFG with productions

$S \rightarrow AB \mid BC$
$A \rightarrow BA \mid a$
$B \rightarrow CC \mid b$
$C \rightarrow AB \mid a$

and let $w = baba$



{S,A,C}				
-	{S,A,C}			
-	{B}	{B}		
{S,A}	{B}	{S,C}	{S,A}	
{B}	{A,C}	{A,C}	{B}	{A,C}
b	a	a	b	a

Undecidable decision problem for CFLs:

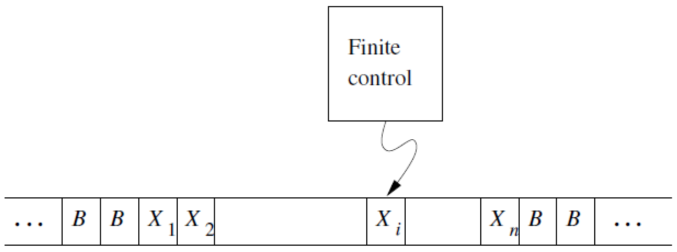
- given a CFG G, test whether G is ambiguous
- given a representation for a CFL L, test whether L is inherently ambiguous
- given a representation for two CFLs L_1 and L_2, test whether the intersection $L_1 \cap L_2$ is empty
- given a representation for two CFLs L_1 and L_2, test whether $L_1 = L_2$
- given a representation for a CFL L defined over Σ, test whether $L = \Sigma^*$

8. Turing Machines

8.1. Definition

A Turing machine is a finite state automaton with the addition of a **memory tape** with unlimited capacity in both tape directions and sequential access.
Input: a finite string, composed by symbols, it's placed in the memory tape. At its right and left there is an infinite series of Blank symbols.
Head: at the beginning is pointing to the leftmost cell of the input.
Move: performed according to its state and the symbol which is read by the tape head. In a move:

- changes its state
- writes a new symbol in the cell read by the tape head
- moves the tape head to the cell to the right or to the le



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

- Q: finite set of states
- Σ: finite set of input symbols
- Γ: finite set of tape symbols. $\Sigma \subseteq \Gamma$
- $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
  - Input (q, X): state and tape symbol
  - Output (p, Y, D): state, symbol that replaces X, direction (Left or Right)
- q_0: initial state
- $B \in \Gamma$: symbol Blank $\notin \Sigma$
- $F \subseteq Q$: finite set of final states

A TM changes its configuration with each move. We use instantaneous description (ID) to describe configurations.
Instantaneous description (ID): string $X_1X_2 \dots X_{i-1}qX_iX_{i+1} \dots X_n$

- q: state
- tape head is reading the i-th tape symbol
- $X_1 \dots X_n$: visited portion of the tape

Computation step $\vdash$: binary relation between instantaneous descriptions

- If $\delta(q, X_i) = (p, Y, L)$ then $X_1X_2 \dots X_{i-1}qX_iX_{i+1} \dots X_n \vdash X_1X_2 \dots pX_{i-1}YX_{i+1} \dots X_n$. Replace X_i with Y and the head move to the left of 1 symbol.
- If $\delta(q, X_i) = (p, Y, R)$ then $X_1X_2 \dots X_{i-1}qX_iX_{i+1} \dots X_n \vdash X_1X_2 \dots X_{i-1}YpX_{i+1} \dots X_n$. Replace X_i with Y and the head move to the right of 1 symbol.
- If the head goes outside, it will add the symbol B.

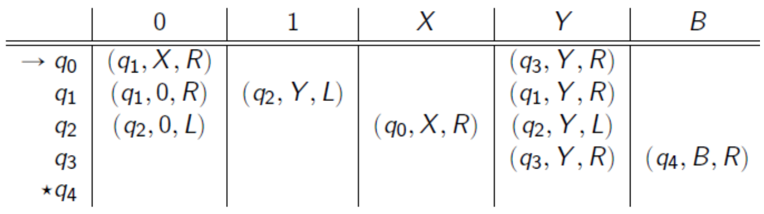
Computation is represented by the Closure $\overset{*}{\underset{M}{\vdash}}$: (zero or more moves)

Initial ID has the form q_0w

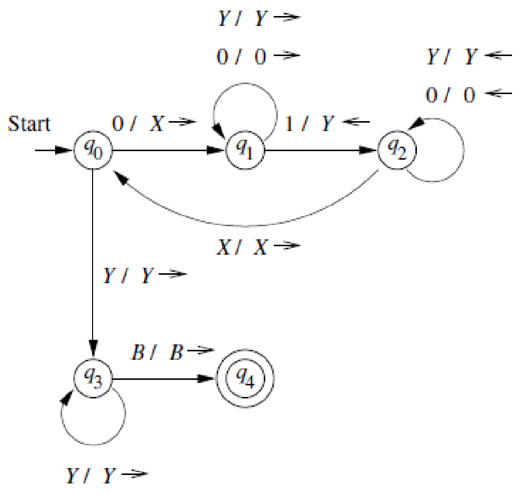
Accepting computation has the form $q_0w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ with $w \in \Sigma^*, p \in F,\ \alpha, \beta \in \Gamma^*$

Transition diagram: arch from q to p is labeled by one or more objects X/YD with

- X: symbol read from the cell of the tape
- Y: symbol that replaces
- $D \in \{L, R\}$ direction, represented also as with an arrow.



	0	1	X	Y	B
$\rightarrow q_0$	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	(q_4, B, R)
$\star q_4$					



We have defined a TM as a recognition device. Alternatively, we can use these devices to compute functions on natural numbers.
We encode each natural number in unary notation according to the scheme $n =_1 0^n$

Language accepted: $L(M) = \left\{w | w \in \Sigma^*, q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta, p \in F,\ \alpha, \beta \in \Gamma^*\right\}$
Called **Recursively Enumerable (RE)**

Halts (stops): if it enters a state q with tape symbol X and $\delta(q, X)$ is not defined (there is no next move)
If a TM accepts a string, we can assume that it always halts: just make $\delta(q, X)$ undefined for every final state q.
If a TM does not accept, we can't assume that it will halt (in a non-final state).

Recursive language (REC): language accepted by a TM that halts on each input string (independently of acceptance).
Recursively enumerable language (RE): language accepted by a TM that halts when the string belongs to the language.

A decision problem P is:

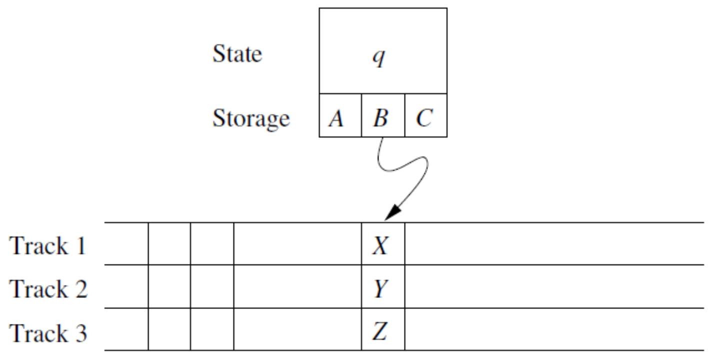
- **Decidable** if its encoding L_P is a **recursive language**.
  Alternatively: if there is a TM M that always halts such that $L(M) = L_P$
- **Undecidable**: if there is no program that can solve it.
- Every problem P can be represented by a language L_P, and solving P means solving the problem of checking if a string belongs to L_P.
- **Intractable**: decidable (solvable) bat requires a huge amount of time to be solved.

8.2. Programming techniques for TM

Techniques to facilitate the writing of programs for TM.

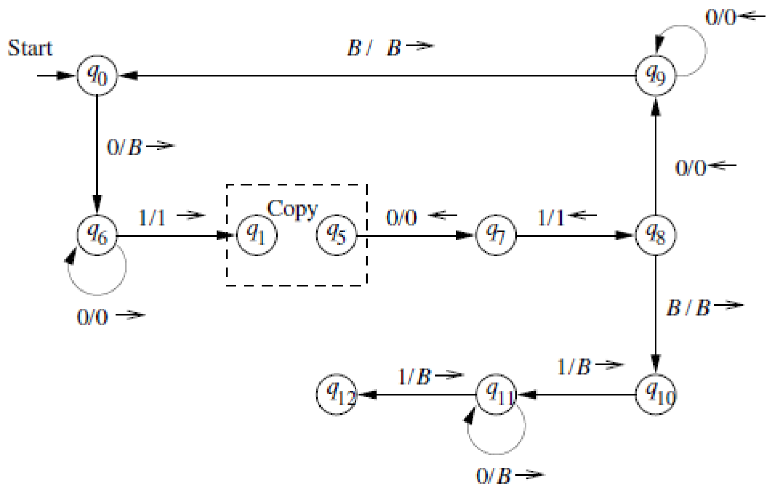
TM with:

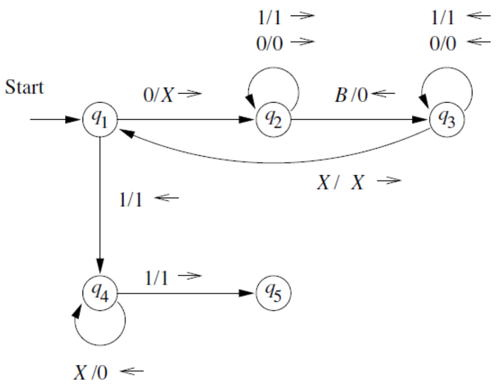
- a finite number of registers with random access, which we place inside each state
- a finite number of tape tracks. (Sometimes used to mark symbols of the first tape)



State is a n-ple $[q, A, B, C]$
Tape alphabet is composed by n-ples with a component for each track $[X, Y, Z]$

Subroutine: set of states that execute a procedure



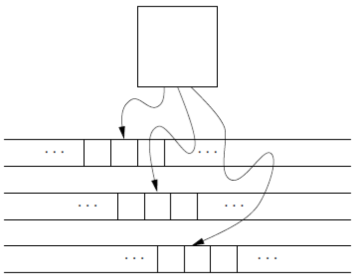


8.3. Extensions

More complex than TM but with the same computational capacity.

Multi-tape TM: finite number of independent tapes for the computation, with the input on the first tape. Useful to simulate a real calculator.
In one move:

- state update
- for each tape:
    - write a symbol in current cell
    - move the tape head independently of the other heads (L = left, R = right, or S = stay)



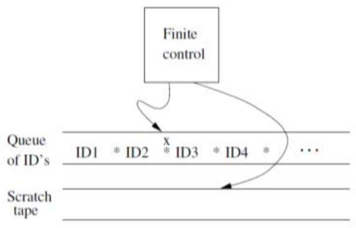
Accept same languages of normal TM. (proof slides 08.33)
Can be simulated by a single-tape TM in $O(n^2)$ time, with n number of moves to simulate.

Nondeterministic TM (NTM): transition function δ returns sets of triplets

$$\delta(q, X) = \{(q_1, Y_1, D_1), \dots, (q_k, Y_k, D_k)\}$$

At each step, the NTM chooses one of the triples as the next move.
Accepts an input w if there exists a sequence of choices that leads from the initial ID for w to an ID with an accepting state.
Accept same languages of normal TM.

Proof (skecth) We specify M_D as a TM with two tapes



A single ID in the queue (first) tape is marked as being processed

M_D performs the following cycle

- copy the marked ID from the queue tape to the scratch (second) tape
- for each possible move of M_N, add a new ID at the end of queue tape
- move the marker in the queue tape to the next ID

Let m be the maximum number of choices for M_N After n moves, M_N reaches a number of ID bounded by

$$1 + m + m^2 + \cdots + m^n \leqslant nm^n + 1$$

M_D explores all the IDs reached by M_N in n steps before each ID reached in $n + 1$ steps, as in a breadth first search

If there exists an accepting ID for M_N on w, M_D reaches this ID in a finite amount of time. Otherwise, M_D does not halt

We therefore conclude that $L(M_N) = L(M_D)$ □

Observe that the TM MD in the previous theorem can take an amount of time exponentially larger than MN to accept an input string.
We do not know if this slowdown is necessary: this very important issue will be the subject of investigation in a next chapter.

8.4. Restrictions

Simpler than normal TM but with the same computational capacity and accept same languages.

Semi-infinite tape:

- the head can not visit the cells to the left of the initial tape position
- a tape symbol can never be overwritten by the blank B

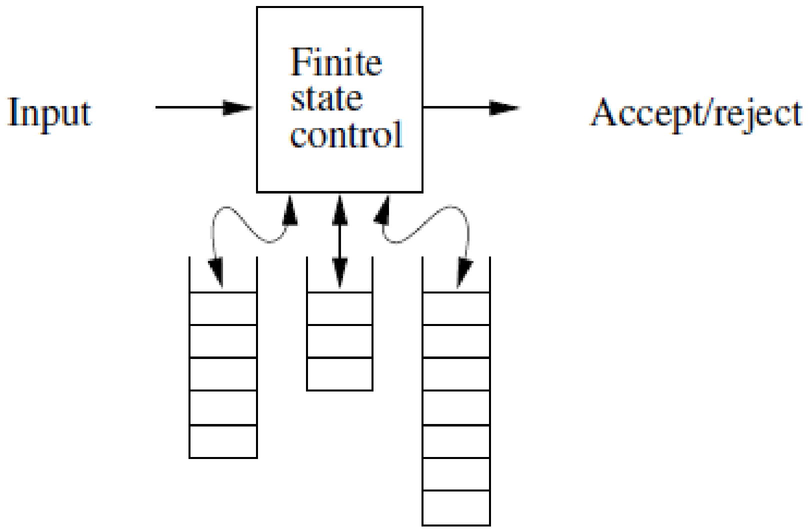
Each ID is a sequence of tape symbols other than B, it means that there are no holes
We can simulate a normal TM with TM with semi-infinite tape that has two tracks:

- the upper track represents the initial position X_0 and all tape cells to its right
- the lower track represents all tape cells to the left of X_0, in reverse order
- a special symbol $*$ is used to mark the initial position

X_0	X_1	X_2	...
$*$	X_{-1}	X_{-2}	...

Accept same languages of normal TM. (proof 08.44)

Multi-Stack machine: multi-tape TM in which every track is used has stack.
Generalization of the PDA: is a PDA with more that 1 stack.



Transition rule (k stacks): $\delta(q, a, X_1, \dots, X_k) = (p, \gamma_1, \dots, \gamma_k)$
It means that: when the machine is in state q and reads input symbol $a \in \Sigma \cup \{\epsilon\}$, and with X_i on top of the i-th stack, $1 \leq i \leq k$, it moves to state p and replaces each X_i with γ_i.
With 2 stack, can simulate a normal TM.
Accept same languages of normal TM. (proof 08.52)

9. Undecidability

9.1. Non-RE languages

Recursively enumerable language **(RE):** if $L = L(M)$ for some TM M. Languages accepted by a TM. M halts if $w \in L(M)$, but M may not halt if $w \notin L(M)$

Recursive (REC) or **Decidable** language: if $L = L(M)$ for some TM M that halts on every input.

- if $w \in L$ then M accept (and halts)
- if $w \notin L$ then M halts in non-final state.

Corresponds to the definition of algorithm, for which we impose that computation halting occurs both for positive and negative instances of the problem.

Binary string indexing (enumeration): associate to each binary string $w \in \{0,1\}^*$ a positive integer index i. w_i denote the i-th string. $w = w_i \Leftrightarrow i = 1w$

string	ϵ	0	1	00	01	$\cdots$
i	1	2	3	4	5	$\cdots$

$\text{enc}(M) =$ **binary string that represents a TM with binary input alphabet**:

- We need to assign integers to each state, tape symbol, and symbols L and R for directions.
- rename the states as $q_1, q_2, \dots, q_r$. With q_1 initial state, q_2 final state (unique)
- rename the tape symbols as $X_1, \dots, X_s$. With $0 = X_1$, $1 = X_2$, $B = X_3$
- rename Directions as $L = D_1, R = D_2$
- transition function $\delta(q_i, X_j) = (q_k, X_l, D_m)$ is encoded as $C_i = 0^i 1 0^j 1 0^k 1 0^l 1 0^m$. It never has two consecutives 1s.
- For a TM, we concatenate the codes C_i for all transitions, separated by 11: $C_1 11 C_2 11 \dots 11 C_n$

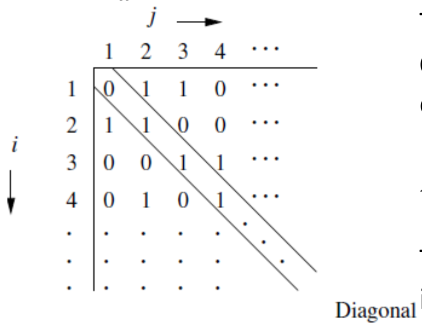
There are several codes for M, obtained by indexing the symbols and/or listing the transitions in different orders. Many binary strings do not correspond to a TM. enc() is not a function.

TM indexing (enumeration): for $i \geq 1$, the i-th string that represent the TM M_i is:

- if w_i is a valid encoding representing TM M, then $M_i = M$
- if w_i is not a valid encoding, then M_i halts immediately for any input. $L(M_i) = \emptyset$

Diagonalization language L_d $= \{w | w = w_i, w_i \notin L(M_i)\}$ with $w_i = \text{enc}(M_i)$

- contains all binary strings w_i such that the i-th TM does not accept w_i.
- **NOT RE** (proof 09.16). There is no TM that accepts L_d
- $\overline{L_d}$ **is RE.**



j
i
Diagonal

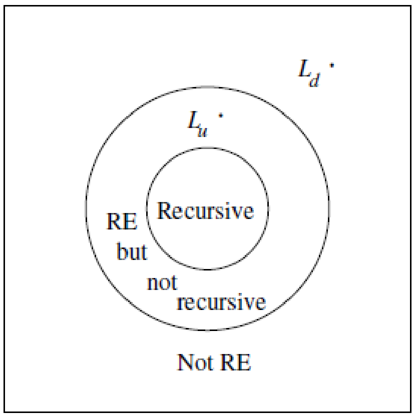
$i \backslash j$	1	2	3	4	$\cdots$
1	0	1	1	0	$\cdots$
2	1	1	0	0	$\cdots$
3	0	0	1	1	$\cdots$
4	0	1	0	1	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

The i-th row indicates if M_i accept (1) the binary string w_j

Characteristic vector (i-th row): each 1 indicates that the corresponding string (column) belongs to the language.

L_d is the complement of the diagonal.
If the i-th element of the diagonal is 0, then $w_i \notin L(M_i), w_i \in L_d$
The table represents the entire class RE. In fact, a language is in RE if and only if its characteristic vector is a row of the table.

9.2. Undecidable languages



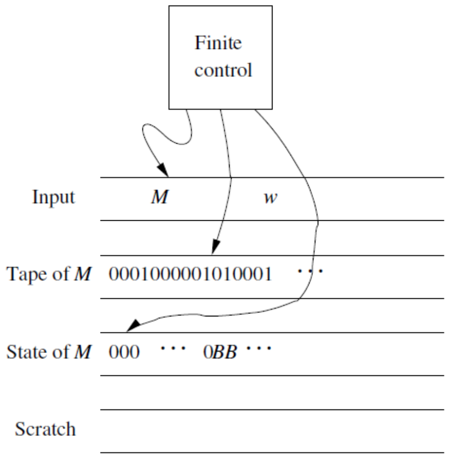
Recursive (REC) = decidable = M always stops. There is an algorithm to solve the problem.
RE = M stops just if accept.
NOT-RE = we can't compute. Ex. L_d
RE\REC=semi-decidibile

If L is recursive, then $\bar{L}$ is recursive. (proof 09.20)
If $L \in RE$ and $\bar{L} \notin RE$, then L is not REC
If L and $\bar{L}$ are in RE, then L is REC. (proof 09.21)
Possible arrangements for L and $\bar{L}$:

- both L and $\bar{L}$ are recursive
- both L and $\bar{L}$ are not in RE
- L is RE but not recursive, and $\bar{L}$ is not RE
- $\bar{L}$ is RE but not recursive, and L is not RE

Universal language L_u $= \{\text{enc}(M,w) | w \in L(M)\}$

- Is the set of binary strings that encode a pair (M, w) such that $w \in L(M)$
- Set of binary strings that represent a TM and its accepted strings.
- $\text{enc}(M)$ followed by 111, followed by w
- **is RE but NOT Recursive** (proof 09.30)
- $\overline{L_u} = \{\text{enc}(M,w) | w \notin L(M)\}$ is **NOT RE**



Universal TM U: $L(U) = L_u$
4 tapes:

- tape 1: contains the input string $\text{enc}(M,w)$
- tape 2: simulates M's tape, using the 0^j format for each X_j tape symbol, and 1 as cell separator
- tape 3 records M's state, using the 0^j format for each state q_j
- tape 4: auxiliary copying tape, used to enlarge or shrink the available space for the 0^j representations in tape 2

Strategy exploited by U:

- if $\text{enc}(M)$ is invalid, U halts and rejects (in this case $L(M) = \emptyset$)
- write $\text{enc}(w)$ on tape 2, using 10 for $0 = X_1$ and 100 for $1 = X_2$
- write the initial state on tape 3, using 0 for q_1, and place the tape 2 head on the first cell
- search on tape 1 for a transition of the form $0^i10^j10^k10^l10^m$, where:
  - 0^i is the state on tape 3
  - 0^j is M's tape symbol under the tape head of tape 2
- in order to simulate transition $0^i10^j10^k10^l10^m$, the TM U:
  - replaces the content of tape 3 with 0^k (new state)
  - replaces 0^j on tape 2 with 0^l (new tape symbol); if needed, we can enlarge or shrink U's tapes using the auxiliary tape (tape 4)
  - move the tape head of tape 2 to the left if $m = 1$ or to the right if $m = 2$, until the next 1 is reached (separator)
- if there is no transition $0^i10^j10^k10^l10^m$, M halts and U halts
- if M reaches a final state, then U halts and accepts

Halting problem L_H $= \{\text{enc}(M, w) | w \in H(M)\}$

- $H(M) =$ set of strings w such that M halts with input w
- There exist a TM M such that $L(M) = L_H$: M takes as input a pair $\text{enc}(M', w)$ and simulates a computation of M' on w.
- **L_H is RE but not Recursive**

It means that there is no algorithm that can state whether a given program ends or not on a given input.
However, there exists a procedure that

- halts, if a given program ends on a given input
- cycles, if a given program does not end on a given input

9.3. Undecidable problems

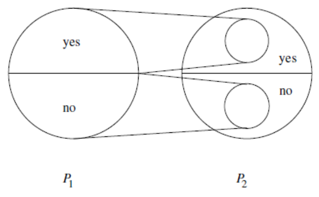
Given a problem **P_1 known** to be difficult, we want to know whether a second problem P_2 under investigation is as hard as, or even harder than, P_1
If we could solve P_2, then we could also solve P_1, written as $P_1 \leq_m P_2$

Reduction of P_1 to P_2: $P_1 \leq_m P_2$

- If P_1 is undecidable, so is P_2 (proof 09.37)
- If P_1 is not RE, so is P_2 (proof 09.38)

is an **algorithm (TM) that converts an instance x of P_1 into an instance y of P_2**, such that

- if x has positive answer then y has positive answer (yes to yes)
- if x has negative answer then y has negative answer (no to no)



Solve P_1 by converting it to P_2 and using a subroutine for P_2
P_1 is converted in a subset of instances of P_2 so P_2 can be larger (harder) than P_1

Let $P_1 \leq_m P_2$, and assume there exists an algorithm that solves P_2. Given an instance x for P_1:

- we use the reduction to convert x to an instance y for P_2
- we use the algorithm for P_2 to decide whether y in P_2 or not

Whatever the answer is, it is also valid for x in P_1
We have built an algorithm that solves P_1. Thus solving P_2 is at least as difficult as solving P_1

Let $P_1 \leq_m P_2$: (proof 09.37)

- If P_1 is undecidable, so is P_2
- If P_1 is not RE, so is P_2

Language L_e $= \{\text{enc}(\boldsymbol{M}) | \boldsymbol{L}(\boldsymbol{M}) = \emptyset\}$

- **NOT RE**. (proof 09.44)
- Set of strings that represents encodings of TMs that accepts empty languages.

Language $L_{ne} = \overline{L_e}$ $= \{\text{enc}(\boldsymbol{M}) | \boldsymbol{L}(\boldsymbol{M}) \neq \emptyset\}$

- **RE but not Recursive**. (proof 09.40-42)
- Set of strings that represents encodings of TMs that accepts non-empty languages.

Both are undecidable.

Property P of RE languages: subset of RE languages that satisfy the property P
Trivial Property: if it is satisfied by all RE languages or by none of the RE languages.

- $P = RE$ or $P = \emptyset$

Language $L_P = \{\text{enc}(M_i) | L(M_i) \in P\}$ set of encodings of all TMs M_i such that $L(M_i) \in P$
P is decidable if and only if L_P is recursive

Rice's theorem: any nontrivial property of RE languages is undecidable (is not Recursive).
This means that, for any nontrivial property P, there is NO TM that

- always halts
- given as input $\text{enc}(M_i)$, decides whether the language $L(M_i)$ satisfies P

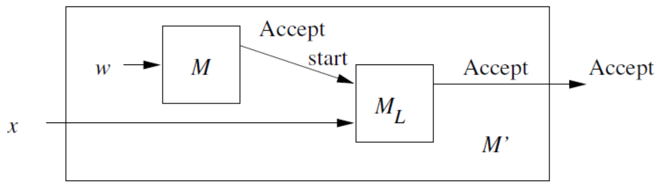
Proof:

Proof Let $\mathcal{P}$ be a nontrivial property of the RE languages. Let us assume by now that $\emptyset \notin \mathcal{P}$

Let $L \in \mathcal{P}$ and let M_L be a TM such that $L(M_L) = L$

We prove that $L_u \leqslant_m L_{\mathcal{P}}$. Then the theorem follows from the fact that L_u is undecidable

Given an instance $\text{enc}(M, w)$ for L_u, we produce an instance $\text{enc}(M')$ of $L_{\mathcal{P}}$



- if M does not accept w, M' does not accept any input string, and thus $L(M') = \emptyset \notin \mathcal{P}$
- if M accepts w, M' simulates M_L on x, and thus $L(M') = L \in \mathcal{P}$

Let us now assume that $\emptyset \in \mathcal{P}$. We consider $\overline{\mathcal{P}}$, the set of RE languages that do not satisfy the property $\mathcal{P}$

Since $\emptyset \notin \overline{\mathcal{P}}$, the above argument proves that $L_u \leqslant_m L_{\overline{\mathcal{P}}}$. Therefore $L_{\overline{\mathcal{P}}}$ is not recursive

Each TM accepts some RE language. Therefore we have

$$\overline{L_{\mathcal{P}}} = L_{\overline{\mathcal{P}}}$$

If $L_{\mathcal{P}}$ were recursive, then $L_{\overline{\mathcal{P}}}$ would be recursive as well. This is a contradiction with respect to what we have previously asserted □

9.4. How to solve Exercises

Show a language L is in REC:

- provide a TM that always halts, and accept the string in the language
- OR prove L and $\overline{L}$ are in RE. So L is REC

Show a language L is in RE: provide a TM that halts for all the string in the language.
Show a language L is in RE\REC:

- show it is in RE
- show it is not in REC (possibly by doing a reduction: $L_{known\ not\ in\ REC} \leq_m L$)

Show a language L is NOT RE: reduction $L_{known\ not\ in\ RE} \leq_m L$

Given property P tell if L_P is not Recursive: show that P is not trivial

- $P \neq \emptyset$: find at least 1 RE language that $\in P$
- $P \neq RE$: find at least 1 RE language that $\notin P$
- By rice theorem, L_P is not Recursive

Given property P tell if L_P is RE: Build a TM that accept L_P

- M_P receives as input a string z and checks if it is a valid encoding enc(M) of some TM M. If not, then M_P halts in a non-final state.
- M_P simulates M on input w. If M accepts, then M_P also accepts.
- Then prove that it has the yes/no property. Map yes->yes and no->not halt.
  - Yes->Yes: if $w \in M$ then …., then,…, then M_P accept
  - No->Not halt: if $w \notin M$ then …., then,…, then M_P not halt

$$\begin{array}{lll}\mathsf{enc}(M) \in L(M_{\mathcal{P}}) & \text{iff} \quad w \in L(M) & \text{(definition of } M_{\mathcal{P}}\text{)} \\ & \text{iff} \quad L(M) \in \mathcal{P} & \text{(definition of } \mathcal{P}\text{)} \\ & \text{iff} \quad \mathsf{enc}(M) \in L_{\mathcal{P}} & \text{(definition of } L_{\mathcal{P}}\text{).}\end{array}$$

Tell if a language L is NOT RE: reduce a non-RE language to L
Reduce a language to another one:

- Draw the schema of the reduction, it is a TM that always halts that has as input an instance of the known language and transform it on an instance the language given.
- Then prove that it has the yes/no property. Map yes->yes and no->no.
  - Yes->Yes: if $w \in L_{known}$ then …., then,…, then $enc(\dots) \in L$
  - No->No: if $w \notin L_{known}$ then …., then,…, then $enc(\dots) \notin L$

OR prove that L is not recursive but its complement is RE, then L is not RE

Given an arbitrary string x, construct a special string y, such that $y \in L$ if and only if $x \in L_{known}$
This proof almost always requires two separate steps:

- Prove that if $x \in L_{known}$ then $y \in L$
- Prove that if $x \notin L_{known}$ then $y \notin L$

Tricks for exercises:

- Universal TM gets in input enc(M) and w, accept if w in L(M)
- NTM can “guess” (generate) a string w. Or guess a factorization w=xy. Or guess multiple strings.
- TM can implement a for loop over the length of the input string, giving the index as input to other TM or Halting if reach the end of the for loop.
- $M_{\emptyset}$ TM that accept the emptyset. Used in reductions when we have more TM than the input.

- M_w TM that accept only w. $L(M_w) = \{w\}$. Used in reduction, having as input w we can return encM_w
- M_ϵ $L(M_\epsilon) = \{\epsilon\}$
- Generator that generate all pairs (i, j) st $i + j = k$ for all possible k. Used to create a string w_i and simulate it on a TM for at maximum j steps. Used to avoid infinite computations.
- A TM can simulate a TM given as input enc(M)
- TM on input w, finds the index i such that$w_i = w$, and returns as output the string $\mathrm{enc}(M_i, w_i)$

9.5. Post's correspondence problem (PCP)

Undecidable problem used to show that other problems are undecidable. (proof 09.71)

Definition: given two equal length lists of strings, tell if it's possible to choose a sequence from each list such that it's concatenation is equal.

Problem Instance (A, B):

- $A = w_1, \dots, w_k$
- $B = x_1, \dots, x_k$
- Where $w_i, x_j \in \Sigma^+$ and Σ is an alphabet with at least two symbols.

Has a solution if there are $m \geq 1$ indices $i_1, \dots, i_m$ such that $w_{i_1} w_{i_2} \dots w_{i_m} = x_{i_1} x_{i_2} \dots x_{i_m}$

	A	B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

A possible solution is provided by the indices :
$m = 4,\ i_1 = 2,\ i_2 = 1,\ i_3 = 1,\ i_4 = 3$

$$w_2 w_1 w_1 w_3 = x_2 x_1 x_1 x_3 = 101111110$$

Modified Post's correspondence problem MPCP: additional constraint that (w_1, x_1) must be the starting string, and m can be 0. $w_1 w_{i1} \dots w_{im} = x_1 x_{i1} \dots x_{im}$

Reduction L_u to MPCP: Transform (M, w) instances of L_u to instances (A, B) of the MPCP.

- semi-infinite tape TM with ID's without any blank.
- represent M's computations as strings of the form $\#\alpha_1\#\alpha_2\#$ … where each α_i is an ID.
- we use fictitious ID's that erase the tape when a final state is reached (needed to realign)
- partial solutions of (A, B) simulate computations of M on w
- in a partial solution, the list obtained by A is always one ID behind with respect to the list obtained by B.

$$l_A: \#\alpha_1 \dots \#\alpha_{i-1}$$
$$l_B: \#\alpha_1 \dots \#\alpha_{i-1}\#\alpha_i$$

- The pairs (w_i, x_i) are used, through several steps, to
  - copy $\#\alpha_i$ from l_B to l_A
  - add to l_B the new string $\#\alpha_{i+1}$, which simulates the next move of M

Transformation of (M, w):

- Pairs of type 1: initial ID

A	B
$\#$	$\#q_0 w\#$

- Pairs of type 2: copy tape symbol and #

A	B	
X	X	for each $X \in \Gamma$
$\#$	$\#$	

- Pairs of type 3: simulate next move for $q \in Q \backslash F$

A	B	
qX	Yp	if $\delta(q, X) = (p, Y, R)$
ZqX	pZY	if $\delta(q, X) = (p, Y, L)$
$q\#$	$Yp\#$	if $\delta(q, B) = (p, Y, R)$
$Zq\#$	$pZY\#$	if $\delta(q, B) = (p, Y, L)$

- Pairs of type 4 : for $q \in F$, erase working tape

A	B
XqY	q
Xq	q
qY	q

- Pairs of type 5: align the two lists, after the tape has been erased

A	B
$q\#\#$	$\#$

Example:

Instance of L_u : $(M, 01)$

$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_3\})$

q_i	$\delta(q_i, 0)$	$\delta(q_i, 1)$	$\delta(q_i, B)$
$\rightarrow q_1$	$(q_2, 1, R)$	$(q_2, 0, L)$	$(q_2, 1, L)$
q_1	$(q_3, 0, L)$	$(q_1, 0, R)$	$(q_2, 0, R)$
$\star\ q_3$	—	—	—

type	w_i	x_i	derived from
(1)	$\#$	$\#q_1 01\#$	
(2)	0	0	
	1	1	
	$\#$	$\#$	

type	w_i	x_i	derived from
(3)	$q_1 0$	$1q_2$	from $\delta(q_1, 0)(q_2, 1, R)$
	$0q_1 1$	$q_2 00$	from $\delta(q_1, 1)(q_2, 0, L)$
	$1q_1 1$	$q_2 10$	from $\delta(q_1, 1)(q_2, 0, L)$
	$0q_1\#$	$q_2 01\#$	from $\delta(q_1, B)(q_2, 1, L)$
	$1q_1\#$	$q_2 11\#$	from $\delta(q_1, B)(q_2, 1, L)$
	$0q_2 0$	$q_3 00$	from $\delta(q_2, 0)(q_3, 0, L)$
	$1q_2 0$	$q_3 10$	from $\delta(q_2, 0)(q_3, 0, L)$
	$q_2 1$	$0q_1$	from $\delta(q_2, 1)(q_1, 0, R)$
	$q_2\#$	$0q_2\#$	from $\delta(q_2, B)(q_2, 0, R)$

type	w_i	x_i	derived from
(4)	$0q_3 0$	$q_3\#$	
	$0q_3 1$	$q_3\#$	
	$1q_3 0$	$q_3\#$	
	$1q_3 1$	$q_3\#$	
	$0q_3$	$q_3\#$	
	$1q_3$	$q_3\#$	
	$q_3 0$	$q_3\#$	
	$q_3 1$	$q_3\#$	
(5)	$q_3\#\#$	$\#$	

M accepts input 01 through the following computation

$$q_1 01 \underset{M}{\vdash} 1q_2 1 \underset{M}{\vdash} 10q_1 \underset{M}{\vdash} 1q_2 01 \underset{M}{\vdash} q_3 101$$

We consider the partial solutions of MPCP associated with the above computation

First pair is mandatory, and simulates the initial ID

$$\ell_A : \ \#$$
$$\ell_B : \ \#q_1 01\#$$

We have only one way to expand the partial solution, that is, use the pair $(q_1 0, 1q_2)$ which simulates the first move

$$\ell_A : \ \#q_1 0$$
$$\ell_B : \ \#q_1 01\#1q_2$$

We apply three pairs for copying, in order to reach the next state

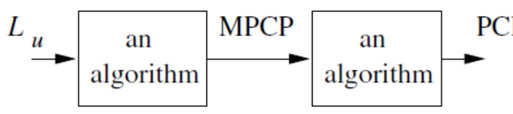
$$\ell_A : \ \#q_1 01\#1$$
$$\ell_B : \ \#q_1 01\#1q_2 1\#1$$

We apply pair $(q_2 1, 0q_1)$ to simulate the second move

$$\ell_A : \ \#q_1 01\#1q_2 1$$
$$\ell_B : \ \#q_1 01\#1q_2 1\#10q_1$$

And so forth ...

Reduction MPCP to PCP: not this year



$L_u \leq_m MPCP$ (proof 09.70)

9.6. Other undecidable problems

CFG ambiguity:

- the instances are the strings $\text{enc}(G)$ where G is a CFG
- the answer is positive if G is ambiguous

We define the corresponding language $L_{AMB} = \{\text{enc}(G)\ |G \text{ is ambiguous}\}$

Reduction (transformation) of PCP to L_{AMB} instances:

- Given an instance of PCP (A, B) over alphabet Σ, where $A = w_1, \dots, w_k$ and $B = x_1, \dots, x_k$
- $G_A \in CFG$
    - Nonterminal set $\{A\}$
    - Alphabet $\Sigma \cup \{a_i | 1 \leq i \leq k\}$ where a_i is an alias for the pair w_i, x_i
    - Production set:
        - $A \to w_1 A a_1 | w_2 A a_2 | \dots | w_k A a_k$
          $\to w_1 a_1 | w_2 a_2 | \dots | w_k a_k$
- $G_B \in CFG$
    - Nonterminal set $\{B\}$
    - Alphabet $\Sigma \cup \{a_i | 1 \leq i \leq k\}$ where a_i is an alias for the pair w_i, x_i
    - Production set:
        - $B \to x_1 B a_1 | x_2 B a_2 | \dots | x_k B a_k$
          $\to x_1 a_1 | x_2 a_2 | \dots | x_k a_k$
- G_A and G_B are unambiguous
- We define $L_A = L(G_A)$ and $L_B = L(G_B)$
- G_{AB} is the set CFG that generates the language $L_A \cup L_B$
    - Nonterminal set $\{S, A, B\}$
    - Alphabet $\Sigma \cup \{a_i | 1 \leq i \leq k\}$
    - Product set $S \to A|B$ and in addition all productions of G_A and G_B

$PCP \leq_m L_{AMB}$ (proof 09.77)

Following CFG problems are undecidable:

- $L(G_1) \cap L(G_2) = \emptyset$?
- $L(G_1) = L(G_2)$?
- $L(G_1) = L(R)$?
- $L(G_1) = T^*$, for a fixed alphabet T?
- $L(G_1) \subseteq L(G_2)$?
- $L(R) \subseteq L(G_1)$?

From Rice theorem the following TM problems are undecidable

- is the language accepted by a TM the empty language?
- is the language accepted by a TM a finite language ?
- is the language accepted by a TM a regular language ?
- is the language accepted by a TM a context-free language ?
- does the language accepted by a TM contain the string "ab"?
- does the language accepted by a TM contain all even numbers ?

10. Intractability

Intractable problem: if the time needed to solve it (decide it) is more than polynomial.
The problems that can be solved in polynomial time on a computer coincide with polynomial time solvable problems on TMs.

10.1. Classes P and NP

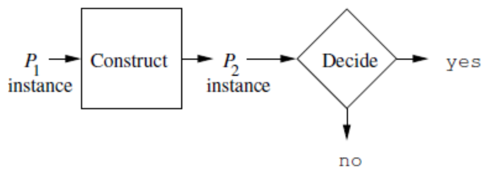
TM M has **time complexity $T(n)$** if, given as input a string w with $|w| = n$, M halts after at most $T(n)$ computational steps.

A language (decision problem) **L belongs to the class P** if there exists a **polynomial** $T(n)$ such that $L = L(M)$ for some **deterministic** TM M with time complexity $T(n)$

A language (decision problem) **L belongs to the class NP** if there exists a polynomial function $T(n)$ such that $L = L(M)$ for some **NON-deterministic** M with time complexity $T(n)$

We can always assume that M performs exactly $T(n)$ moves for every input of length n: to this end, we can simulate a clock function on a special tape track.
$P \subseteq NP$: every TM is also a NTM
A polynomial NTM can perform an exponential number of computations simultaneously. Therefore, it is commonly assumed that $P \neq NP$, but there is not a formal proof yet.

Show that a problem $P_2 \notin P$ (cannot be solved in polynomial time): reduce a problem $P_1 \notin P$ to P_2



(polynomial reduction)
We impose the additional constraint that the reduction operates in polynomial time, $P_1 \leq_p P_2$.
If $P_1 \leq_p P_2$ and $P_1 \notin P$ then $P_2 \notin P$

A language L is ***NP*-complete** if

- $L \in NP$
- for each language $L' \in NP$ we have $L' \leq_p L$

NP-complete problems are the most difficult problems in NP.
If $P \neq NP$ then NP-complete problems are in $NP \backslash P$
If P_1 is NP-complete, $P_2 \in NP$, $P_1 \leq_p P_2$ then P_2 is NP-complete. (proof 10.20)

Proof The polynomial time reduction has the transitive property. For any language $L \in \mathcal{NP}$ we have $L \leqslant_p P_1$ and $P_1 \leqslant_p P_2$, and therefore $L \leqslant_p P_2$ $\square$

If an NP-complete problem is in P then $P = NP$. (proof 10.21)

Proof Assume P is NP-complete and $P \in \mathcal{P}$. For any language $L \in \mathcal{NP}$ we have $L \leqslant_p P$ and therefore we can solve L in polynomial time $\square$

A language L is ***NP*-hard** if, for each language $L' \in NP$ we have $L' \leq_p L$
We do not require membership in NP. L could be much more difficult than the problems in NP.

10.2. Satisfiability problem (SAT)

Deciding whether a Boolean expression is satisfiable is an NP-complete problem.

Boolean expressions are composed by the following symbols

- an infinite set $\{x, y, z, x_1, x_2, \dots\}$ of variables defined on Boolean values 1 (true) and 0 (false)
- binary operators $\wedge$ (logical AND) and $\vee$ (logical OR)
- unary operator $\neg$ (logical NOT)
- round brackets (to force precedence)

Recursively defined as:

- $E = x$, for any Boolean variable x
- $E = E_1 \wedge E_2$ and $E = E_1 \vee E_2$
- $E = \neg E_1$
- $E = (E_1)$

Operator precedence (decreasing): $\neg$, $\wedge$, $\vee$

Truth assignment T for a Boolean expression E assigns a Boolean value $T(x)$ (true or false) to each variable x in E
The Boolean value $E(T)$ of E under T is the result of the valuation of E with each variable x replaced by $T(x)$.
T **satisfies** E if $E(T) = 1$
E is **satisfiable** if there exists at least one T that satisfies E

Satisfiability problem (SAT):

- input: is a Boolean expression E
- output: "YES" if E is satisfiable, "NO" otherwise

Boolean expression encoding:

- We rename the variables as $x_1, x_2, \dots$ and encode them using symbol x followed by a binary representation of the index. Ex: $x_{13} = x1101$
- Logical operators and parentheses are represented by themselves

$\mathrm{enc}(E)$ has length $O(m \log m)$, which is a polynomial function of m
The SAT language is formed by the set of all Boolean expressions that are well-formed, properly coded, and satisfiable.

Cook Theorem: SAT is an NP-complete language. (proof 10.32, NO EXAM)

Simplified version of SAT (3SAT):

- NP-complete problem
- convenient to define reductions

Literal is a variable or else the negation of a variable.
Clause is the disjunction (logical OR) of literals.
Boolean expressions in **conjunctive normal form (CNF)** is a conjunction (logical AND) of clauses.
k-conjunctive normal form (k-CNF): CNF in which every clause has exactly k literals.

CSAT: is some CNF satisfiable ? NP-Complete
kSAT: is some k-CNF satisfiable?

10.3. Other NP-complete problems

Finding out that a decision problem is NP-complete indicates that there are very few chances to discover an efficient algorithm for its solution. It is therefore recommended to look for partial/approximate solutions, using heuristics.

Definition schema:

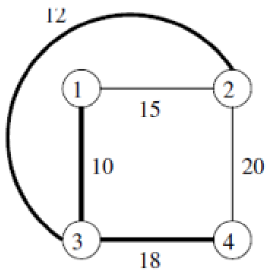
- **Problem**: name of the problem
- **Input**: representation/encoding
- **Output**: when the output is YES
- **Reduction from**:

10.3.1. Independent set (IS) problem

G undirected graph.

Independent set: subset I of nodes of G such that no pairs of nodes in I is connected by arcs of G.

Maximal Independent set: if any other independent set of G has a number of nodes smaller or equal than the former.



$I = \{1{,}4\}$

- **Problem**: independent set (IS)
- **Input**: undirected graph G and lower bound k
- **Output**: YES if and only if G has an independent set with k nodes
- **Reduction**: from 3SAT

NP-complete. (proof 10.62)

10.3.2. Node cover (NC) problem

G undirected graph.

Node Cover: subset C of nodes of G such that each arc of G touch at least one node in C

Minimal Node Cover: every other node cover has at least the size

- **Problem**: node cover (NC)
- **Input**: undirected graph G and upper bound k
- **Output**: YES if and only if G has a node cover with at most k nodes
- **Reduction**: from IS

NP-complete. (proof 10.71)

10.3.3. Directed Hamiltonian circuit (DHC) problem

G directed graph.

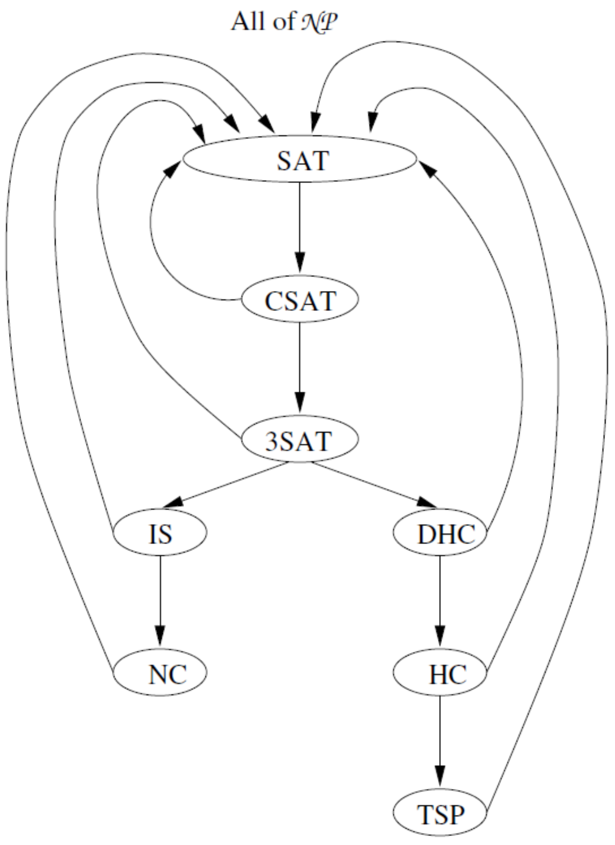
Directed Hamiltonian circuit: oriented cycle that passes through each node of G exactly once

- **Problem**: directed Hamiltonian circuit (DHC)
- **Input**: directed graph G
- **Output**: YES if and only if G has a directed Hamiltonian circuit
- **Reduction**: from 3SAT

- **Problem**: undirected Hamiltonian circuit (HC)
- **Input**: undirected graph G
- **Output**: YES if and only if G has an undirected Hamiltonian circuit
- **Reduction** : from DHC

- **Problem**: traveling salesman problem (TSP)
- **Input**: undirected graph G with integer weights at every arc, and upper bound k
- **Output**: YES if and only if G has an undirected Hamiltonian circuit such that the sum of the weights at each arc is smaller equal than k
- **Reduction**: from HC

Summary of our reductions:



Exam Questions

You can study them for free as flashcard at https://www.brainscape.com/p/3YF5P-LH-BBQSC

Regulars 2-3-4:

- **Theory**:
  - Subset construction. Proof L(DFA)=L(NFA). (19-09-13, 20-01-27, 21-06-28,
  - Definition of equivalent pair states of a DFA (21-01-25,
  - Provide the mathematical definition of language L*. Provide a rigorous proof that (L*)* = L*. (21-06-28, slides 03.38,
  - Tell why Regular languages are closed under reverse operator (theorem 4.11, 22-01-26,
- Given DFA, prove that L(A)=L:
  - Exams: 20-02-17, 21-01-25,
  - PDF: 1.1,
  - Slides: 2.30,
- Given Language, construct NFA, convert NFA in DFA.
  - PDF: 1.3,
  - Slide: 02.50, 02.55,
  - Book: section 2.3
- Given language tell if it's Regular (pumping)
  - Exams: 19-01-22, 19-06-28, 19-09-13, 20-02-17, 21-02-12, 21-09-03,
  - PDF: 1.4, 1.6
  - Slides: 04.16,
- Convert FA in Regular Expression:
  - Exams: 19-01-22, 21-02-12,
  - Slides: 03.28,
  - Book: section 3.2
- Convert Regular Expression in e-NFA:
  - Exams: 19-06-28, 21-09-03,
  - Book: section 3.2
- Given generic L1 and L2, tell if intersection/concatenation/… is regular/not regular
  - Slide 4.31,
- Given DFA, apply Tabular algorithm states equivalence and reduce to its minimum
  - Exams: 19-02-13, 21-01-25,

CFL 5-6-7:

- **Theory**:
  - Tell why CFL is not open to the complement operation. The complement of a CFL is always recursive? (19-09-13,
  - Show that the class of context-free languages is not closed under intersection. Specify in detail the construction that takes as input a PDA P and a DFA A and produces a PDA P' that accepts the language L(P)∩L(A). (21-02-12,
- Given L, build the grammar G. Then prove that L=L(G) by mutual induction.
  - Exams: 19-01-22, 21-09-03,
  - PDF: 2.1
- Given L, tell if its context-free (pumping).

    - Exams: 19-02-13, 20-01-27, 21-01-25, 21-06-28, 22-01-26,
    - PDF: 2.2, 2.3, 2.4, 2.5, 2.6,
    - Slide: 7.2.16
- Given generic L1 and L2, tell if intersection/concatenation/… is regular/context free.
    - Exams: 19-02-13, 20-01-27, 21-09-03, 22-01-26,
    - PDF: 2.7,
    - Slide: 7.2.42
- Given grammar G, remove e-prod, reduce to chomsky
    - Exams: 19-09-13, 20-02-17(22-01-26 same),
    - Slide: 7.1.40
- Algorithm to verify if string is in language (also specify the algorithm)
    - Exams: 19-06-28, 21-06-28,
- Is CFL closed to the operator P(L)={…}?:
    - PDF: 2.8

Recursive 8-9:

- **Theory**:
    - Define the notion of property of the languages generated by TMs and state Rice's theorem. Provide the proof of Rice's theorem that we have developed in class. (21-09-03,
    - definizione di macchina di Turing nondeterministica e la definizione di linguaggio da questa accettato. Dimostri che ogni linguaggio accettato da una macchina di Turing nondeterministica può essere accettato da una macchina di Turing deterministica.
    - Richiamare la definizione del linguaggio Lne. Dimostrare che Lne non appartiene alla classe REC. Attenzione: è richiesta la dimostrazione svolta in classe per questo teorema, non utilizzare il teorema di Rice. (20-02-17,
- Given property P over RE, tell if Lp is REC/RE/NON-RE.
    - Exams: 19-01-22, 20-01-27, 20-02-17, 21-01-25, 21-02-12, 21-06-28,
    - PDF: 3.4, 3.5, 3.6, 3.7, 3.8,
- Given L1 and L2, tell if intersection/concatenation/... are RE/REC/…
    - Exams: 19-01-22, 21-01-25,
    - PDF: 3.10,
- Given L={enc(M),…}, tell if L is RE (reduction)
    - Exams: 19-02-13, 19-01-22, 19-09-13, 21-02-12, 21-06-28, 22-01-26,
    - PDF: 3.9, 3.10.3, 3.11, 3.12,
- Given L, specify Turing machine M that accepts it and stops for each input.
    - Exams: 19-06-28,
    - PDF: 3.1, 3.2, 3.3,

Intractable 10: (this year every exam will have a question of this chapter 2022)

- **Theory**:
    - Siano P1 e P2 due problemi appartenenti alla classe NP. (20-01-27,
        - Richiamare la nozione di riduzione polinomiale di P1 a P2
        - definizione di problema NPcompleto.
        - Dimostrare che se P1 è NP-completo e se esiste una riduzione polinomiale di P1 a P2, allora anche P2 \`e NP-completo. Perchè è cruciale cha la riduzione utilizzata impieghi tempo polinomiale?

    - If P_1 is NP-complete, $P_2 \in NP$, $P_1 \leq_p P_2$ then P_2 is NP-complete. (proof 10.20)
    - If an NP-complete problem is in NP then $P = NP$. (proof 10.21)
    - The class P of languages that can be recognized in polynomial time by a TM is closed under intersection with regular languages. (22-01-26,
- Reduction exercise: 21-02-12.e4