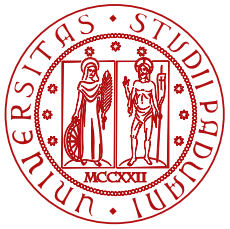


Università
degli Studi
di Padova

Dipartimento
di Ingegneria
dell'Informazione

Master degree in Computer Engineering

Automata, Languages and Computation

University of Padova

Academic Year
2023/2024

Contents

1 Introduction

1.0.1 Introduction

Mathematically study of **computation** is so important: we have to study **computability**, in other words what can be computed (from the italian word "calcolo") and **tractability**, in other words what can be efficiently computed.
To do this we'll use two things:

- **Automata Theory**: abstract models of machine
- **Formal Language Theory**: abstract representations of data

There is different models of computation: we'll see Finite Automata, Turing Machines and Formal Grammars.
There are two types of automaton: **recognition model** that it has an input like a string and gives one output, and **generative model** that it hasn't an input and generates all of the desired sequences.
We will see that there will be a lot of techniques for demonstrating our theorems.
However there is some prerogative concepts about **alphabet and strings** that we have to know.
One alphabet is a finite and nonempty set of atomic (indivisibili) symbols like: $\Sigma = \{0, 1\}$, the binary alphabet, and string is a finite sequence of symbols from some alphabet.
From any alphabet there will be chosen the empty string, ϵ, that is the string with zero symbols. The length of a string w is $|w|$.
The power Σ^k is the set of all k-lenght strings with the symbols from the alphabet

and the number of string is $\|w\|^k$ and the set of all strings of Σ is Σ^*.
So holds that:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 ...$$

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 ...$$

$$\Sigma^* = \Sigma^+ \cup \{\epsilon\} ...$$

There are useful conventions like:

- a, b, c... alphabet symbols
- u, w, x, y, z strings
- a^n
- $a^0 = \epsilon$

At the end there is the **language L** that is a set of strings chosen from Σ^*.
For example set of all the words in some English dictionary is a language, or set of strings consisting of n zeros followed by n ones, with $n \geq 0$.

$$\{\epsilon, 01, 0011, 000111, ...\}$$

It's important do not confuse between empty language $\emptyset$ that contains no string and language $\{\epsilon\}$ that contains only empty string.
A language can be described by using a **extensive** representation like the language described above composed by a list of strings or an **intensive** representation, using a **set former** like $\{w \mid \text{statement specifying } w\}$:

$$\{W \mid W\ consists\ of\ an\ equal\ number\ of\ 0's\ and\ 1's\}$$

and could be expressed in mathematical form like:

$$\{0^n 1^n \| n \geq 0\}$$

and is important remember that the product between 0 and 1 is a **concatenation**.
It's important not confuse the two notations below:

$$\{0^n 1^n \mid n \geq 0\}$$

represent a single language being the condition of n greater equal zero inside the graphs.

Instead

$$\{0^n 1^n\},\ n \geq 0$$

represent different languages, each one with one value of n.

2

Regular Languages

2.1 Finite Automata

A finite automata is maybe one of the simplest computer that we can mind. Finite Automata or FA represent a finite set of **states** with **transitions** from one state to another.
The simplest representation for an FA is a graph with:

- Nodes that represent the states
- Arcs that represent transitions
- Labels on each arc that represent the reason of the transition

2.1.1 Deterministic Finite Automata

These devices read input from left to right and never go back. They are **recognition devices**, don't giving an output structure like a string but only an answer like yes or not, true or false. The response true means that the string that we've as input is in the language L.
These devices can store can only store limited information, using the states that have fixed means.
They're very simple to implement in a computer (table).
A Deterministic Finite Automaton (DFA) is composed by 5 mathematical-object

(5-tuple):

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

where:

- Q is a set of finite states
- Σ is the input symbols, for example $\Sigma = \{0, 1\}$
- δ is a transition function $(Q\ x\ \Sigma) \hookrightarrow Q$ that is the set of instructions that allow to change from one state to another state. If you are in a couple q, a, the transition function gives in which state p the automaton goes
- q_0 is the initial state
- F is a set of final states, where we have to be at the end.

Example: A DFA A that accepts the language of all strings of 0 and 1 containing the substring 01:

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

Are accepted strings with pattern 01 somewhere. The alphabet is composed by 0 and 1 and the strings are composed by concatenation of two other strings x and y come from the language $\{0, 1\}^*$. The shortest string accepted is 01 with x and y that are ϵ.
The set of states Q is $\{q_0, q_1, q_2\}$ and the final state is q_1.
There is two methods to represents the DFA: **graphical representation** or with a **transition table**.
The graphical representation is a transition diagram built starting from a point the first state q_0. Here, the input consumed is ϵ, the empty string.
For each possible symbol, in this case 0 or 1, the transition function δ, that is represented by an arc, gives the instruction for changing the state.
In this example, if we had a 1, we remain in the state, still looking for a 0, instead, if we have a 0, we go to another state.
In the second state, if we have a 0, we remain in the state, waiting for a 1, instead, if we have a 1, we go to the final state that is represented by a double circle.
In the final state, that means that the pattern 01 is found, for each symbol we had, we remain in the state, because for any symbol the output our response doesn't change.
For DFA, for each states there will be a number of arcs same of the number of the symbols of the alphabet and no more: the automata is deterministic and it

means that for each state and each symbol given there is only one possible state where DFA can go.
In Figure 2.1 the correct diagram of this DFA.

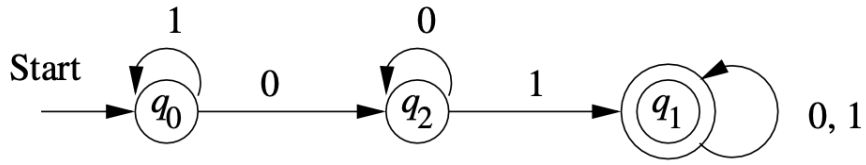


Figure 2.1: Diagram of the DFA

As we said before there is another method to represent the DFA. The transition table of this DFA is illustrate in Figure 2.2:

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

Figure 2.2: Table of the DFA

As in the diagram representation there are six possible combination, from the moment that there are 3 states and only 2 symbols.
Without talking in mathematical words, a DFA accepts a string $w = a_1a_2a_3...a_n$ if there is a path in the transition that starts in the initial state and ends in some final state, having a sequence of transitions for each symbols added.

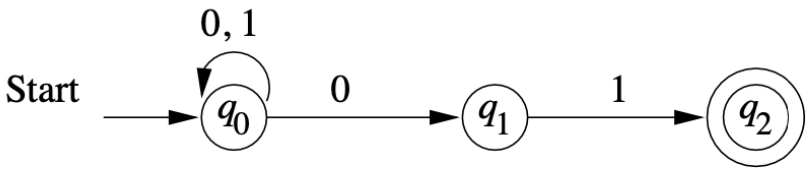


Figure 2.3: Uncorrect diagram

In Figure 2.3 is illustrated a transition diagram that is not correct for two reason:

for the state q_0 there is not an unique instruction for the symbol 0 and means that is not deterministic, for the state q_1 there is not the 0 instruction.

Extended transition function

The δ transition can be extended to function $\hat{\delta}$ defined not in this way $(Q\ x\ \Sigma) \hookrightarrow\ Q$ but in this way $\hat{\delta} : (Q\ x\ \Sigma^*) \hookrightarrow\ Q$ which takes as input a state and a string $w \in \Sigma^*$ and gives the state in which we must move.
It function with recursion:

- **Base** $\hat{\delta}(q, \epsilon) = q$
  It's the zero step and if i have q and the input is ϵ, we remain in q
- **Induction** $\hat{\delta}(q, x \cdot a) = \delta(\hat{\delta}(q, x), a)$
  Where xa is a string formed by a first part x and as last symbol a. Giving a string $y = xa$, in a recursive way the $de\hat{l}ta$ is invoked with only the string x while δ with the symbol a give us the next state. At the end we will have a "Matrioska" of single delta function, each one with one symbol of the string y

Example: Given a string w over the alphabet and a symbol a, let $\#_a(w)$ denote the number of occurrences of a in w.
Specify a DFA A that accepts only the strings in the following language:

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

In words, L contains only the strings that have an even number of 0's and an even number 0f 1's.

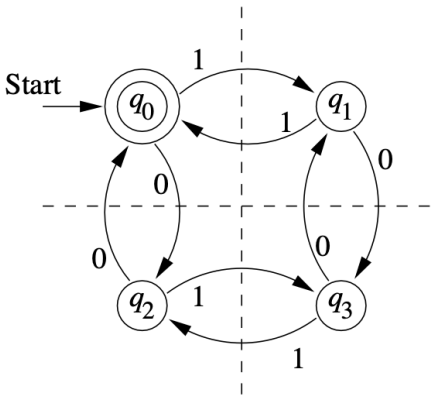


Figure 2.4: Diagram representation of the example

There are 4 states, each one with one meaning that we'll see after. For example

q_3 state means that the DFA has seen an odd number of 1 and an odd number of 0.
It's important to remember that a state is not a register where store information: in each state it's possible to store one determinate scenario.
States have the following meaning:

- q_0 means an even number of 0 and 1
- q_1 means an even number of 0 and an odd number of 1
- q_0 means an even number of 1 and and odd number of 0
- q_0 means an odd number of 0 and 1

It's possible to represent DFA by using the tabular representation shown in Figure 2.5 below:

	0	1
$\rightarrow \star q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Figure 2.5: Table of the example

To see if a string w is accepted by A we will use the $\hat{\delta}$ function as shown below in Figure 2.6:

Is string $w = 0101$ accepted by A ?

- $\hat{\delta}(q_0, \epsilon) = q_0$
- $\hat{\delta}(q_0, 0) = \delta(\hat{\delta}(q_0, \epsilon), 0) = \delta(q_0, 0) = q_2$
- $\hat{\delta}(q_0, 01) = \delta(\hat{\delta}(q_0, 0), 1) = \delta(q_2, 1) = q_3$
- $\hat{\delta}(q_0, 010) = \delta(\hat{\delta}(q_0, 01), 0) = \delta(q_3, 0) = q_1$
- $\hat{\delta}(q_0, 0101) = \delta(\hat{\delta}(q_0, 010), 1) = \delta(q_1, 1) = q_0 \in F$

Figure 2.6: Delta hat function for strings

As we can see, in the first row, there is the base case with the empty string.
For each row there is an example of one substring shorter than 0101. The real path is to invoke 0101 and in a recursive method invoke each row from the

bottom.
In this method there will be a matrioska of δ with the inner contains the base case.
If the last state is final means that string is accepted by the DFA.
It's important to introduce the concept of **language recognized** by the DFA A:

$$L(A) = \{w \mid \hat{\delta}(q,w) \in F\}$$

that in words means all the strings that endS in a final states.
We have to remember some concepts: we have said that Σ^* is a set of all strings composed by the symbols of Σ. It's an infinite set and, from this set, we could have a subset, a set of strings, strings of a certain type, that is called a **language**.
So a language L is a subset of $L \subseteq \Sigma^*$.
A set of languages is called **class** and the class of languages accepted by DFAs are called **regular languages**.
So if L is equal of $L(A)$ for some DFA A, is a regular language. It's obvious that if i've done a correct DFA A for a language L, $L = L(A)$.
In the exercises it's important to know that F is a subset of Q, in other words there will be many possible final states.
Moreover, when you make an exercise of a DFA, it's useful start from the initial state and do the shortest path to have the solution/s.
Also there is the possibility to define a failure state.

2.1.2 Nondeterministic Finite Automata

These automata accepts only regular languages (as DFAs) and it's easier to design.
There are two methods to interpret NFAs:

- **Clone Interpretation**: a nondeterministic finite automaton can **simultaneously** be in different states, with many branch, moving on in parallel states.
  One of the clone will die because in some states there will be a **stuck** point.
  The automaton accepts the string if at least one final state is reached at the end of the scan
- **Guess Interpretation**: there is an "oracle" that knows the correct path and when in the NFA there will be a branch, we'll ask to the oracle which is the correct path.
  In a given state the automaton can guess which is the next state that will lead to acceptance

In the Figure 2.7 below is shown an example of nondeterministic automaton that accepts only strings ending in 01:

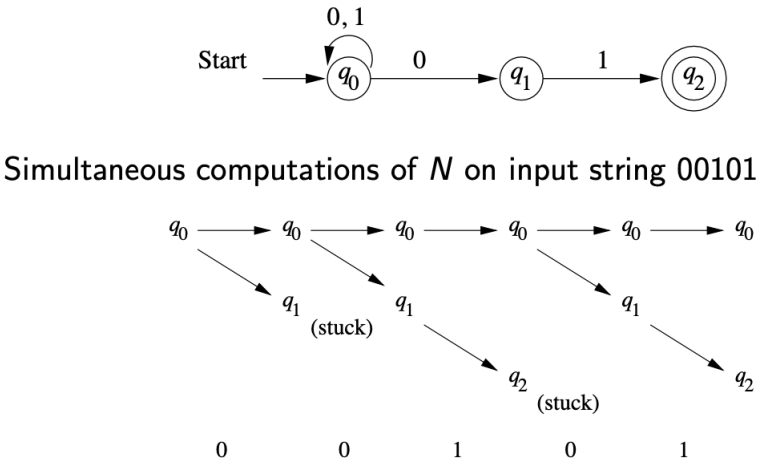


Figure 2.7: Diagram and clone interpretation of the DFA

A nondeterministic finite automata is also a 5-tuple like DFA:

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

where:

- Q is a set of finite states
- Σ is the input symbols, for example $\Sigma = \{0, 1\}$
- δ is a transition function $Q\ x\ \Sigma \hookrightarrow\ 2^Q$ where 2^Q is the set of all subsets of Q, all the possible combinations of the states
- q_0 is the initial state
- F is a set of final states, where we have to be at the end.

Also in NFA, there is the possibility to represent it with a table representation as shown in Figure 2.8 below:

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

Figure 2.8: Table of the DFA

Obviously also here there is the **extended transition** function $\hat{\delta}$. It function with recursion:

- **Base** $\hat{\delta}(q, \epsilon) = \{q\}$
  It's the zero step and if i have q and the input is ϵ, we add it in the set of possible states that reachable by q
- **Induction** $\hat{\delta}(q, x \cdot a) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a)$
  The meaning of the induction step is quite different from DFA: since for each state may be more possibility, the $\hat{\delta}$ function is the union of each possible states reachable by p (that is one of the state in which the automata could be after has consumed x as input) reading as input the symbol a.

In Figure 2.9 below the $\hat{\delta}$ function for the example shown above:

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

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

Figure 2.9: Delta hat function in an NFA example

As we can see in each row there is a set of possible transitions for a given input. When we are in q_1 and we see a 0, there is a dead end, represent by empty set. In nondeterministic FA each state hasn't a meaning like DFA.
It's important deal with the concept of **accepted language** for a NFA A:

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

In words, the language $L(A)$, is the set of strings accepted by A, that are those strings which have in the set of possible ending states at least one final state $\in F$.

2.1.3 Equivalence for DFA and NFA

When we talk about equivalence we mean that NFA and DFA can compute the same language with both. However, NFAs are more powerful and less restricted than DFAs. We can explain that using an example, described below:

$$L(A) = \{w \mid w = x1a\ ,\ x \in \{0,1\}^*,\ a \in \{0,1\}\}$$

that in words means a language that accepts only strings with penultimate symbol 1.

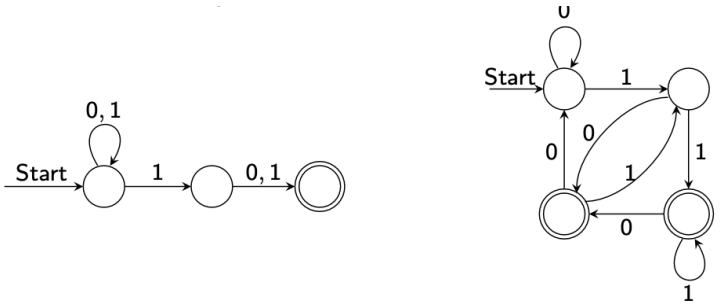


Figure 2.10: Same language recognized by different automaton

As we can see in Figure 2.10 above the DFA's is more more difficult respect NFA's.

Theorem: For every NFA N there exists some DFA D such $L(D) = L(N)$.

Proof: For the proof we use **subset construction**.

Retaking the Figure 2.11 we could use the "fence interpretation" which subdivide the different "steps" (from each state, the possible states for the symbol given) with imaginary fence.

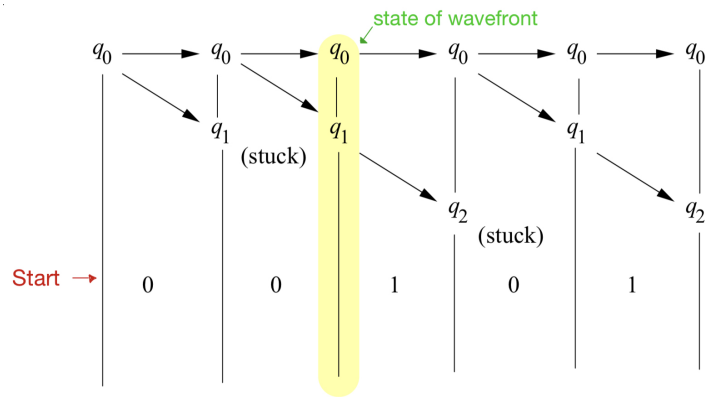


Figure 2.11: Fence interpretation of an NFA

The idea is to build a **wave front** or a **state** in D for every **state set** representing a "configuration" in a computation of N.
For example, if we are in q_0 and we read a 0, we can go in q_1 or remain in q_0. A set composed by q_0 and q_1 is a wave front or state of D.
The number of state of D is finite because the states become from a set of finite states. There will be no more than an exponential number.
Resuming, we can construct an automaton that implement an NFA using a DFA where states are wavefronts of NFA.
Now that we have this intuition we can define the 5-tuple for NFA and DFA.
Given an NFA:

$$N = (Q_N, \Sigma, \delta_N, q_0, F_N)$$

the subset construction produces a DFA:

$$D = (Q_D, \Sigma, \delta_D, q_0, F_D)$$

such that $L(A) = L(D)$.
Each element of Q_D is a subset of the states of Q_N. Every state of Q_D represent one wavefront/fence.
Subset construction:

- $Q_D = \{S \mid S \subseteq Q_N\}$: every S is a subset of the states of Q_N
- $F_D = \{S \subseteq Q_N \mid S \cap F_N \neq \emptyset\}$: the final states of the equivalent DFA is all the subset which has not empty intersection with the final states of NFA F_N. In other words, one state or one wavefront has to have a state $\in F_N$.
- $\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$: supposing to be in a wavefront/state (remembering that is a set of states) and we have to make the union of the δ_N function of all the state of Q_N in the wavefront (for example q_1 or q_2) for the symbol a. To compute all δ_D we take all the possible subsets of Q_N and we do the table of transition

It's important to remember that the number of states of Q_D is $|Q_D| = 2^{|Q_N|}$ but the majority of states in Q_D cannot be reached from the initial state.
In Figure 2.12 below is shown the δ_D construction. The subsets S that have the star mean that these states don't appear in the NFA.
Fortunately, we can often avoid exponential growth of states in Q_D using a technique called **lazy evaluation**. This technique doesn't consider the states that the automaton will never see. We build the transition table of D only for the accessible states of D.

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

Figure 2.12: Subset construction (without lazy evaluation) of an NFA

Construction of DFA D through lazy evaluation

- **Base**: the NFA'S starting state is certain a possible singleton subset accessible in D
- **Induction**: If state S is accessible in D, compute $\delta_D(S, a)$ for each symbol of Σ

For example q_1 and q_2 are never accessible starting from q_0.
In Figure 2.13 below is shown DFA D with only accessible states.
In several applications D has about as many states as N.

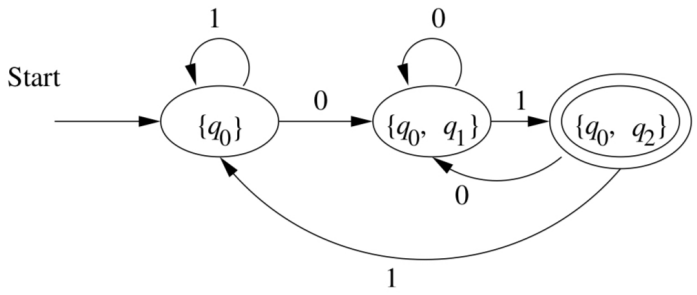


Figure 2.13: DFA constructed by an NFA

Theorem:Let D be the DFA obtained from an NFA N using the subset construction. Then $L(D) = L(N)$.
Proof:We have first to prove that, for every string $w \in \Sigma^*$ we have:

$$\hat{\delta_D}(\{q_0\}, w) = \hat{\delta_N}(\{q_0\}, w)$$

In both sides of the equation there are sets: in the left side is the set of possible states reachable starting by the wavefront composed only by q_0, in the right side is the set of states reachable starting by the state q_0.
We use induction on $|w|$.
Base: When $w = \epsilon$. The claim follows from the definition.
Induction: Suppose that is proved for string of length $n-1$. The induction steps are shown in Figure 2.14 below.

Induction

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

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

Figure 2.14: Inductive steps of the proof

From the definition of $\hat{\delta_D}$ we could expand the first row. The operation done in the second row is, for hypothesis, substitute the equation inside $\hat{\delta_D}$ function.
From the definition of δ_D we can do the substitution in the third row and the meaning of that row is: for all the states reachable by $\hat{\delta_N}$ function having x as input (recursive), do δ_N function consuming as input the symbol a.
This is the definition of $\hat{\delta_N}$ function.
Theorem: A language L is accepted by a DFA $\iff$ is accepted by an NFA.
Proof Intuitively, if we have a transition function for a DFA, we can simply create an NFA with states composed by single state of the DFA.
$\Rightarrow$: is proved by previous theorem.
$\Leftarrow$: Any DFA can be converted into an equivalent NFA by modifying δ_D to δ_N by following rule:

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

By induction on $|w|$ one can show that $\hat{\delta_D}(q_0, w) = p \iff \hat{\delta_N}(q_0, w) = \{p\}$.
It's quite common in practice for the DFA to have roughly the same number of states as the NFA from which is constructed.
However there is an example in which an NFA composed by (n+1) states means

a DFA composed by, at least, 2^n states.
Theorem:There exists an NFA N with n+1 states that has no equivalent DFA with less than 2^n states
Proof: Let N be the NFA that recognize only strings that have as n.th symbol from the last an 1. Expressed in intensive form become:

$$L(N) = \{x1c_2...c_n \mid x \in \{0,1\}^*,\ c_i \in \{0,1\}\}$$

The NFA that represent this language is shown in Figure 2.15 below.

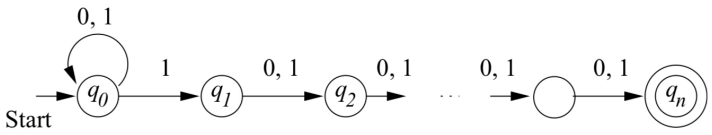


Figure 2.15: NFA for the proof

As we can see there is only one node nondeterministic and the meaning is: when you are in the $n-th$ position, go to the end of the string. To represent it using a DFA, the 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 and, since D has fewer than 2^n states there would be a state q that is the transition state of two different strings $a_1a_2a_3...a_n \neq b_1b_2b_3...b_n$:

$$\hat{\delta_D}(q_0, a_1a_2a_3...a_n) = \hat{\delta_N}(q_0, b_1b_2b_3...b_n) = q$$

This is called **pigeonhole principle**: if there are n holes and $n+1$ pigeons, there is certainly one hole with two pigeons.
Since are different strings, there exists an $a_i \neq b_i$ for some i in the string. We assume $a_i = 1$ and $b_i = 0$.
For $i = 1$ the string composed by a_i is accepted because the $n-th$ symbol is a 1 but we can't say the same for the string composed by b_i.

$$\hat{\delta_D}(q_0, 1a_2a_3...a_n) \in F$$

$$\hat{\delta_D}(q_0, 1b_2b_3...b_n) \notin F$$

This implies a contradiction.
For $i > 1$ the same thing applies: for pigeonhole principle, $\hat{\delta_D}$ function applied for the two strings gives the same state q but from the definition of L we have,

in Figure 2.16, this contradiction

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

Figure 2.16: Contradiction of the theorem

2.1.4 NFA WITH ϵ-TRANSITIONS

It's an extension of NFAs where transitions labelled with symbol ϵ are allowed. This means that the automaton **can change state without consuming any of its input**.
This new capability doesn't expand the class of languages that can be accepted by finite automata but it gives us some added "programming convenience".
It's easier to design than NFAs and accept only regular languages.
A ϵ-nondeterministic finite automata is also a 5-tuple like DFA:

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

where:

- Q, Σ, q_0 and F are defined as for NFAs
- δ is a transition function $Q \ x \ (\Sigma \cup \{\epsilon\}) \hookrightarrow \ 2^Q$ where 2^Q is the set of all subsets of Q, all the possible combination of the states

An example of a language accepted by an ϵ-NFA is the ϵ-NFA which accepts fractional numbers. In Figure 2.17 the diagram representation:
The ϵ-transitions makes operators + and - optional.
In Figure 2.18 below there is the table representation of the ϵ-NFA: The extended transition function for an ϵ-NFA can be defined only after the definition of ϵ-closure.
Let us compute the **ϵ-closure** of a state q, written $ECLOSE(q)$.
The meaning of $ECLOSE(q)$ is a set with all the possible states reachable from q itself trough a sequence of one or more ϵ.
If we talk about a set of states S, is the union of all the $ECLOSE(p)$ with $p \in S$.
Also here is possible to talk about **extended transition function**:

- **Base**: $\hat{\delta}(q, \epsilon) = ECLOSE(q)$

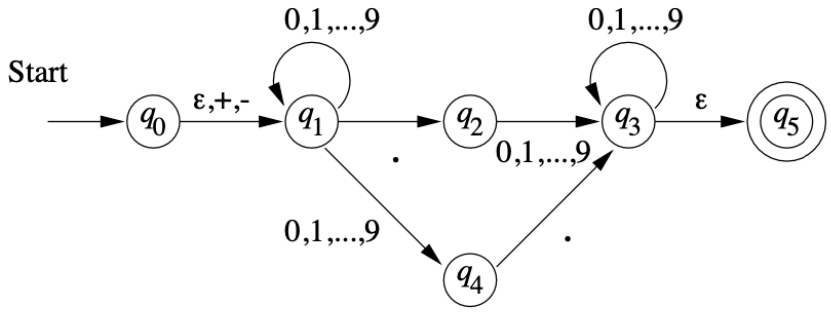


Figure 2.17: ϵ-NFA diagram representation

	ϵ	+,-	.	$0, \ldots, 9$
$\rightarrow q_0$	$\{q_1\}$	$\{q_1\}$	$\varnothing$	$\varnothing$
q_1	$\varnothing$	$\varnothing$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_3	$\{q_5\}$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_4	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$
$\star q_5$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Figure 2.18: ϵ-NFA table representation

- **Induction** $\hat{\delta}(q, xa) = ECLOSE(\{r_1, r_2, ..., r_m\})$ where r_i it's all the possible states for each string p_i (recursive delta on x) when the symbol is a.

In Figure 2.19 below one example of extended transition function applied on the previous example.
Also here there is the concept of **accepted language** by and ϵ-NFA E that is:

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

Also from an ϵ-NFA is possible construct a DFA. The steps are similar as the NFA's ones. Given an NFA:

$$E = (Q_E, \Sigma, \delta_E, q_0, F_E)$$

the subset construction produces a DFA:

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

such that $L(E) = L(D)$.
Each element of Q_D is a subset of the states of Q_E. Every state of Q_D represent

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

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

Figure 2.19: Extended transition function for an ϵ-NFA

one wavefront/fence.
Subset construction:

- $Q_D = \{S \mid S \subseteq Q_E, |S = ECLOSE(S)\}$: every S is a state or rather a subset of the states of Q_E
- $F_D = \{S \subseteq Q_E \mid S \cap F_E \neq \emptyset\}$: the final states of the equivalent DFA is all the subset which has not empty intersection with the final states of NFA F_E. In other words, one state or one wavefront has to have a state $\in F_E$.
- $q_D = ECLOSE(q_0)$, so the starting state of the DFA is a set of states given by the ϵ-closure of q_0

We start by computing $ECLOSE(q_0)$ and this is the new starting state. It may be a set of states S.
There is a method to compute $\delta_D(S, a)$ with $a \in \Sigma$ and $S \in Q_D$: the first thing is to compute all the possible states reachable by each state of the set S, using δ_E function considering a symbol a given. After this step we have a set $\{r_1 ... r_m\}$ and we do the $ECLOSE(\{r_1 ... r_m\})$ finding a set of states that is equal to $\delta_D(S, a)$. We do this for all possible reachable set of states, for all possible symbol given and we put the $\emptyset$ state where there isn't computation.

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

2.2 Regular Expressions

We introduce the concept of **regular expressions** that is a type of language-defining notation. It may be thought as a "programming language".
Regular expressions are closely related to NFA and can be thought as a **user-friendly** alternative to the NFA notation.
In few words regular expression is an algebraic description of a language. We'll define that regular expressions can define exactly the same languages that FA describe.
Regular expressions offer a declarative way to express the strings that we want to accept.
The main concept is that regular expressions denote languages.
Before describing the regular expressions notation, we need to learn operations on languages. We remember the meaning of language: a set of strings, subset of all the possible strings.

- **Union**: $L \cup M = \{w \mid w \in L \; or \; w \in M\}$
- **Concatenation**: $L.M = \{w \mid w = xy, \; x \in L \; , \; y \in M\}$ or in words all the strings of L concatenated with all the strings of M.
  Remember that a concatenation with empty set is also an empty set.
- **Powers**: $L^0 = \{\epsilon\}$, $L^k = L.L^{k-1}$
- **Kleene Closure**:$L^* = \bigcup_{i=0} L^i$
  If we have an alphabet equal to one language L, Kleene closure is equal to Σ^*

In Figure 2.20 is shown how to construct a Kleene Closure of a language L.

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

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

Figure 2.20: Kleene Closure example

A regular expression E over Σ has associated a **generated** language $L(E)$ described recursively, as follows.

We should use $L(E)$ when we want to refer to the language that E denotes.
Basis:

- The constants ϵ and $\emptyset$ are regular expressions that define respectively $L(\epsilon) = \{\epsilon\}$ and $L(\emptyset) = \emptyset$
- If a is any symbol, **a** is a regular expression. That is, $L(\mathbf{a}) = \{a\}$

Induction:

- If E and L are regular expressions, then (E+F) is a regular expression, and $L(E + F) = L(E) \cup L(F)$
- If E and L are regular expressions, then (EF) is a regular expression, and $L(EF) = L(E)L(F)$
- If E is a regular expression, then E^* is a regular expression, and $L(E^*) = (L(E))^*$
- If E is a regular expression, then (E) is a regular expression, and $L((E)) = L(E)$

Let us write an example: a regular expression for a language (set of strings) that consist of alternating 0's and 1's.
In the next line the regular expression for this language.

$$(01)^* + (10)^* + 0(10)^* + 1(01)^*$$

The $\{01\}^*$ means all the strings of form 0101..01 and the same for $\{10\}^*$. As a general rule, if we want a regular expression for the language consisting of only the string w, we use w itself as the regular expression.
While $1\{01\}^*$ means strings that begin and end with 1.
There is another form to write this language that follows:

$$(\epsilon + 1)(01)^*(\epsilon + 0)$$

The start and the end of the regular expression means the optional 1 that could be at the start/end of the string.
There are some rules of precedence: first Kleene closure, second concatenation, third union and parentheses is used to force precedence.
Each regular expression can be naturally associated with a **tree structure** representing its recursive definition, and we'll use later in proofs that need structural induction. The previous example can be associated with the following tree shown in Figure 2.21

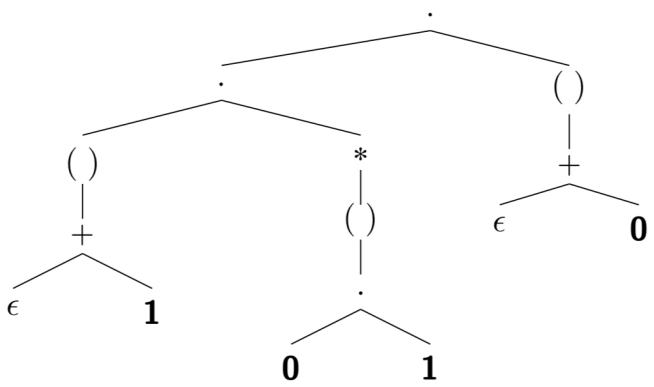


Figure 2.21: Tree associated to a regular expression

2.2.1 Equivalence of Finite Automata and Regular Expressions

These two notations turn out to represent exactly the same set of languages, which we have termed the "regular languages". In order to show that the regular expressions define the same class of languages, we must show that:

- Every language defined by one of these FA is also defined by a regular expression
- Every language defined by a regular expression is defined by one of this automata

In Figure 2.22 below is shown all the equivalences we have proved or will prove.

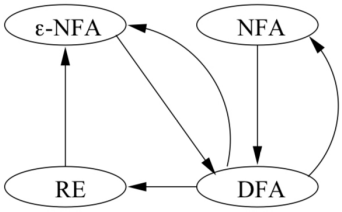


Figure 2.22: Equivalence bewteen different machine

One arc from class X to Y means that we prove every language defined by class X also defined by class Y.

Theorem:If L=L(A) for some DFA A, then there is a regular expression R such that L=L(R).

Proof: For the proof we'll use the **eliminating states method**.

This technique has three steps that are: normalization, state elimination and merge.

In the **normalization** phase the transition from a state p to a state q originally labeled with a symbol a is relabeled with a regular expression **a**. If there is a transition labeled with $(0, 1)$ the regular expression will be **(0+1)**.
If there is not transition between pair p, q, we create a new transition labeled with empty set $\emptyset$.
In the **elimination** phase we start change the automaton keeping the language usual as before. We want to eliminate all the states until we remain with only two states. If we eliminate a state s, for each transition $p \hookrightarrow q$ we need to add a label corresponding to the missing paths. Suppose, like in Figure 2.23 states $q_1, q_2, ...q_k$ are the antecedents of s and states $p_1, p_2, ...p_m$ are the successors of s (remember that could be equivalent for some i).

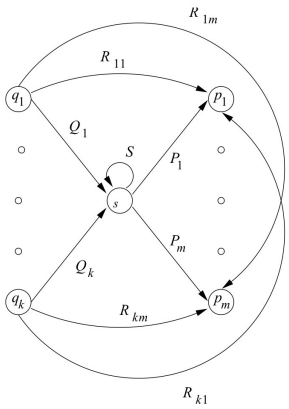


Figure 2.23: Proof

For every antecedents and successors we have to transition with a regular expression that cover the missing paths.
Removing s the DFA become like in Figure 2.24 shown below:

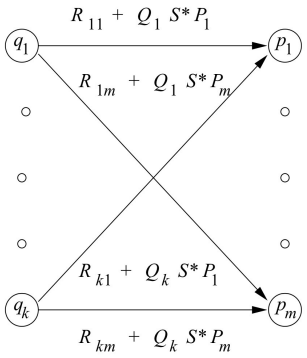
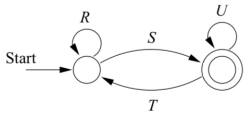


Figure 2.24: Proof

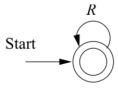
As we can see, removing a state implies the adding of an arc with the following formula:

$$R_{ij} + Q_i S^* P_j$$

After removing all the states except two, we are in the following situation:



corresponding to the regular expression $E_q = (R + SU^*T)^*SU^*$



corresponding to the regular expression $E_q = R^*$

Figure 2.25: Proof

and we get a regular expression E_q.
This operation must be done for each final state $q_i \in F$: we remove all the states except q_0 and q_i resulting an automaton A_{qi} with only two states. After we convert each automaton A_{qi} to a regular expression E_{qi} and at the end, in the **merge** phase we unite all the regular expression.
This construction method could be done for all types of FA.
Theorem: For every regular expression R we can construct an $\epsilon - NFA$ E such that $L(E) = L(R)$.
Proof: We construct E with only one final state, no arc entering the initial state e no arc exiting the final state (for convenience). The construct use the **structural induction**.
The concept of induction is that if you have a sequence of implications, if you prove the implication $P(k)$ is automatically proved also $P(k+1)$ so, if you prove $P(n)$ where n is all the natural numbers, for the previous numbers is also proved.
We have see that we could split a regular expression as a tree with at the base of the tree there are the base case. So it's possible do induction also here:

- The claim holds for trees consisting of a single node
- If the claim holds for trees A and B, it also holds for a new tree consisting of a root with A and B attached as its children
- If the claim holds for tree A, it also holds for a new tree consisting of a root with A attached as its child

The base case is the automata for regular expression $\emptyset$, ϵ, **a** and it is shown in Figure 2.26.

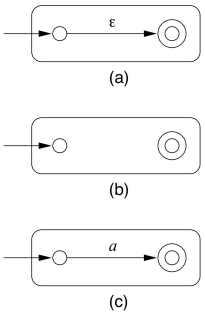


Figure 2.26: Base step of the proof

The induction step for $R + S$, RS, R^* is shown in Figure 2.27.

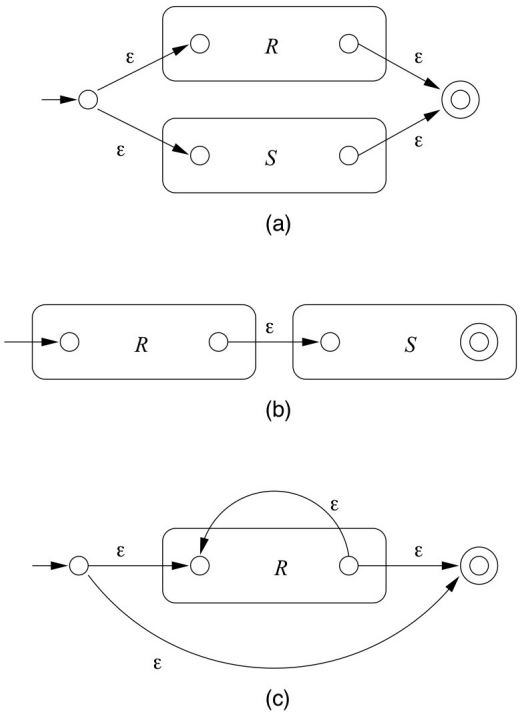


Figure 2.27: Induction step of the proof

There is not parenthesis because if you add a parenthesis you don't change the language. In Figure 2.28 there is an example of an ϵ-DFA constructed by a RE.

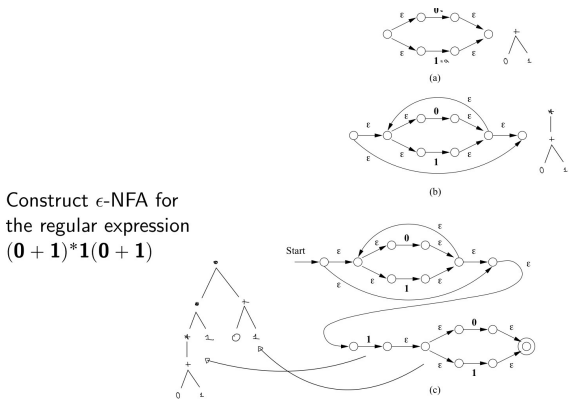


Figure 2.28: Example

2.2.2 Algebraic laws

There are some similarities between regular expressions and arithmetic expressions not only in operators but also in properties. There are also **specific** properties for regular expressions, related to Kleene closure. The following L, M, N are regular expression and not languages.

- Union is **commutative**: L+M = M+L
- Union is **associative**: (L+M)+N = L + (M+N)
- Concatenation is **associative**: (LM)N = L(MN)
- Concatenation is **not commutative**: ML ≠ LM
- $\emptyset$ is the **identity for union**
- ϵ is the **left identity and right identity for concatenation**
- $\emptyset$ is the **left annihilator and right annihilator for concatenation**
- Concatenation **is left and right distributive over union**: L(M+N)=LM + LN
- Union is **idempotent**: L+L = L
- $(L^*)^* = L^*$
- $\emptyset^* = \epsilon$
- $L^* = L^+ + \epsilon$

There is an important proof that we have to know, and it's once of the proposition $(L^*)^* = L^*$.
We have to prove that a regular expression $(R^*)^* = R^*$ and, from the moment that a regular expression generate a language, $L((R^*)^*) = (L(R^*))^* = ((L(R))^*)^*$

and $L(R^*) = (L(R))^*$.
Assuming $L(R) = L_R$ we need to show that $(L_R^*)^* = L_R^*$ and all the steps are shown in Figure 2.29.

$$\begin{aligned}
w \in (L_R^*)^* &\iff w \in \bigcup_{i=0}^{\infty} \left(\bigcup_{j=0}^{\infty} L_R^j \right)^i \\
&\iff \exists k, m \in \mathbb{N} \,:\, w \in (L_R^m)^k \\
&\iff \exists p \in \mathbb{N} \,:\, w \in L_R^p \\
&\iff w \in \bigcup_{i=0}^{\infty} L_R^i \\
&\iff w \in L_R^*
\end{aligned}$$

Figure 2.29: Example

2.3 Pumping Lemma and Properties of Regular Languages

2.3.1 Pumping Lemma for RL

The first tool that we'll see is a way to prove that certain languages are not regular. This theorem is called **pumping lemma** and is a strong technique.
Suppose to have a language (that we remember be a set of string) of this type:

$$L_{01} = \{0^n 1^n \mid n \geq 1\}$$

and for contradiction it's a regular language (that can be represented by a finite automata).
Let k be the number of states of A, a DFA that represent the language, and suppose as input one string that start with n 0's.
Considering that the DFA has states with the following meaning: the state p_0 is the start, there is not 0 in the string, the state p_1 means that the automaton has seen one zero, until p_k that means that the automaton has seen k zeros.
This thing means that for n number there will be $n + 1$ states and, by the **pigeonhole principle** there must exist a pair of index i, j such that $p_i = p_j$.
Assume that this state is called q. From q, if we have:

- if $\hat{\delta}(q, 1^i) \notin F$ assuming the state not final, the machine will foolishly reject $0^i 1^i$
- if $\hat{\delta}(q, 1^i) \in F$ assuming the state final, the machine will foolishly reject $0^j 1^i$

In other words the state would represent inconsistent information about the number of 0 in the string.
Therefore A doesn't exist, L_{01} is not a regular language.

The pumping lemma: Let L be (for sure) any regular language. Then $\exists n \in N$ (each language has is own number n, sometimes the number of states) depending on L, $\forall w \in L$ with lenght of w $|w| \geq n$, we can cut $w = xyz$ such that:

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

It's used to proof that some languages aren't regular: to do this we have to proof the opposite, that there is almost one string such that this thing doesn't value for each k.
There is an important thing to say: there could be some not regular languages that satisfy the pumping lemma and other that don't satisfy it but, if a language is regular, must satisfy the pumping lemma.
Proof: Suppose L is regular so there is some DFA such that $L = L(A)$. Suppose A has n states and we consider string $w = a_1a_2a_3...a_m$ with lenght $m \geq n$. For $i = 0, 1, 2...n$ define state p_i such that $p_i = \hat{\delta}(q_0, a_1a_2a_3..a_i)$ so p_i is the state of A after i symbols is read.
By the pigeonhole principle, there is not possible have n states distinct because there are $n + 1$ symbols and we can find two index i, j such that $p_i = p_j$.
Breaking $w = xyz$ in this way: $x = a_1a_2..a_i$, $y = a_{i+1}...a_j$, $z = a_{j+1}...a_m$ the DFA is shown in Figure 2.30 below. As we can see, x goes from p_0 to p_i, y goes from

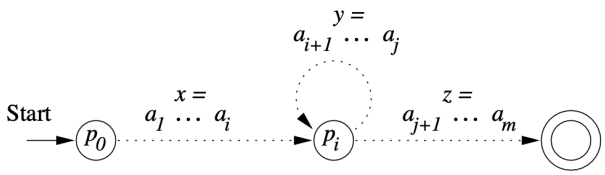


Figure 2.30: DFA of the lemma

p_i to p_i since $p_i = p_j$ and z close the DFA.
If the automaton receives as input xy^kz for any $k \geq 0$ is correct because if $k = 0$ the automaton start from q_0, goes to p_i on input x, since the equivalence between p_i and p_j, from p_i to the accepting state on input z. Thus A accepts xz.
If $k \geq 0$ A goes from q_0 to p_i on input x, circles from p_i to p_i k times on input y^k and on input z goes from p_i to the accepting state.

2.3.2 Closure Properties of Regular Languages

Let L and M regular languages over Σ. The following languages are all regular.

- **Theorem-Union**: For any regular languages L and M, $L \cup M$ is regular
  Proof: Being every language derived by a regular expression, we assume that E and F are regular expression such that L=L(E) and M=L(F). Then $L \cup M$ is generated by the property of RegEx E+F that is also a regular expression that gives a regular language

- **Theorem-Concatenation**: For any regular languages L and M, $L \cdot M$ is regular
  Proof Being every language derived by a regular expression, we assume that E and F are regular expression such that L=L(E) and M=L(F). Then $L \cdot M$ is generated by the property of RegEx EF that is also a regular expression that gives a regular language

- **Theorem-Complement**:If L is a regular language over Σ, then it's a regular language also $\bar{L} = \Sigma^* - L$ // **Proof** Being the language L regular, it can be recognized by a DFA that has a 5-tuple of this type $A = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$. Let $B = (Q,\ \Sigma,\ \delta,\ q_0,\ Q - F)$, so $L(B) = \bar{L}$. A string w is in L(B) if and only if $\hat{\delta}(q_0, w)$ is in Q-F which occurs if and only if w is not in L(A)

- **Theorem-Intersection**: If L and M are regular languages over Σ, then it's a regular language also $L \cap M$ // **Proof** By the Morgan's law $L \cap M = \overline{\bar{L} \cup \bar{M}}$ and we already know that regular languages are closed (still regular) under complement and union.
  In this case there is an alternative proof that is a direct construction of a DFA for the intersection of two regular languages which essentially runs two DFA's in parallel.
  Let L and M be the language of the automata $A_M = (Q_M,\ \Sigma,\ \delta_M,\ q_M,\ F_M)$ and $A_L = (Q_L,\ \Sigma,\ \delta_L,\ q_L,\ F_L)$ such that $L = L(A_L)$ and $M = L(A_M)$ noticing that the alphabet is the same, otherwise we'll have to take the union. The idea is that if A_L goes from state p to state s upon reading a, and A_M goes from state q to state t upon reading a, then $A_{L\cap M}$ will go from state (p,q) to state (s,t) upon reading a.
  We can merge the the two automaton into an automaton
  $A = (Q_M x Q_L,\ \Sigma,\ \delta_M x \delta_L,\ (q_{M0}, q_{L0}),\ F_M x F_L)$ where:

  $$\delta_{L\cap M}((p,q),a) = (\delta_L(p,a), \delta_M(q,a))$$

  and we can show by induction also the equivalent of $\hat{\delta}$ function and then $A_{L\cap M}$ accepts if and only if A_L, A_M accept

- **Theorem-Set Difference**: For any regular languages L and M, L / M is regular
  Proof Observing that $L/M = L \cap \bar{M}$ we already know that regular languages are closed under complement and intersection

- **Theorem-Reverse Operator**: If L is a regular language, it's also regular L^R, that is the reverse of the language.
  Proof Let L be recognized by a FA A from the moment that is regular. Turn A into an FA for L^R **reversing** all arcs, make the **old start state the new accepting state** and create a new start p_0 such that $\delta(p_0, \epsilon) = F$ with F the old set of accepting state.

Homomorphisms: Let Σ and Δ be two alphabets. A homomorphisms over Σ is a function $h : \Sigma \hookrightarrow \Delta^*$ and informally is a function which replaces each symbol with a string.
This notion and function can be extended to Σ^*: if $w = a_1a_2A_3...a_n$ then:

$$h(w) = h(a_1)h(a_2)...h(a_n)$$

and equivalently we can use a recursive definition:

$$h(w) = \epsilon \; if \; w = \epsilon, \; h(x)h(a) \; if \; w = xa$$

For a language $L \subseteq \Sigma^*$ we have that $h(L) = \{h(w) \mid w \in L\}$.
Theorem: Closure under homomorphism: Let $L \subseteq \Sigma^*$ be a regular language and let h be a homomorphism over Σ. Then $h(L)$ is a regular language.
Proof: Let E be a **regular expression** generating L. We define $h(E)$ as the regular expression obtained by substituing in E each symbol a with $a_1a_2...a_k$ under the assumption that the symbol $a \in \Sigma$ and $h(a) = a_1a_2...a_k$.
We now prove the statement:

$$L(h(E)) = h(L(E))$$

using structural induction on E. The base case in when $E = \epsilon$ or $E = \emptyset$ and being $h(E) = E$ implicate that $L(h(E)) = L(E) = h(L(E))$. There is another base case that is when $E = \mathbf{a}$ and $a \in E, h(a) = a_1a_2...a_k$. Then $L(\mathbf{a}) = \{a\}$ and thus $h(L(\mathbf{a})) = \{a_1a_2...a_k\}$. Being the regular expression $h(\mathbf{a})$ $=a_1a_2...a_k$ for hypothesis then $L(h(\mathbf{a})) = \{a_1a_2...a_k\}h(L(\mathbf{a}))$.
The induction step is shown in Figure 2.31 remember that is used structural induction.

Induction Let $E = F + G$. We can write

$$\begin{array}{lcll} L(h(E)) & = & L(h(F+G)) & \\ & = & L(h(F)+h(G)) & h \text{ defined over regex} \\ & = & L(h(F)) \cup L(h(G)) & + \text{ definition} \\ & = & h(L(F)) \cup h(L(G)) & \text{inductive hypothesis for } F, G \\ & = & h(L(F) \cup L(G)) & h \text{ defined over languages} \\ & = & h(L(F+G)) & + \text{ definition} \\ & = & h(L(E)) & \end{array}$$

Figure 2.31: Induction step of closure under homomorphism

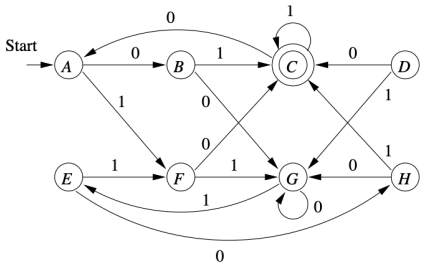
It's proved in the same way the property of concatenation and Kleene closure.
Computational complexity: What is the computational complexity of conversion among DFA, NFA? It's a function that depends on number of states in Finite Automata cases, number of operators for regular expressions and on the cardinality of the alphabet.
Suppose that an ϵ-NFA has n states. To compute ECLOSE(p) we visit at most n^2 arcs, made for n states, we obtain $O(n^3)$. The resulting DFA has 2^n states due to subset construction: for each state and each symbol we compute $\delta(S, a)$ in time $O(n^3)$ so the result it's $O(n^3) \cdot 2^n$.
If we consider lazy evaluation we compute $\delta(S, a)$ only for the s reachable states so it becomes $O(n^3) \cdot s$.
From a DFA to an NFA computation takes $O(n)$, that is linear time.

2.3.3 Decision problems

Empty language: Language $L(A) \neq \emptyset$ for FA A is not empty if and only if at least one final state is reachable from the initial state of A.
There is an algorithm to find the reachable final states that consists in moving from one state to another and if a state is reachable, we go on.
Given a regular expression E we can decide $L(E) = \emptyset$ by structural induction. The **base** case is that if the regular expression is ϵ or **a** is non empty, if $E = \emptyset$ then L(E) is empty. The **inductive** step is to apply the rule of sum, concatenation or Kleene closure.
Language membership We can test a string w for a DFA A by simulating A on w. If $|w| = n$ this takes $O(n)$ steps, if it's not a DFA but an NFA with s states it takes $O(n)$ steps. If it is a ϵ-NFA, it takes $O(n \cdot s^3)$ steps.
Alternatively we can convert an NFA or a ϵ-NFA in to a DFA and then simulate it but the time of the conversion could be exponential in the size of the input FA. This method is used when the FA has small size.
Before dealing with **equivalence of regular languages** we have to talk about **equivalent states**.
For a DFA $A = (Q, \Sigma, \delta, q_0, F)$ let p, q two states and we define:

$$p = q \iff \forall w \in \Sigma^* \quad \hat{\delta}(p, w) \in F \; if \; and \; only \; if \; \hat{\delta}(q, w) \in F$$

If $p = q$ are **equivalent** states, also are **distinguishable** states.
In Figure 2.32 below we can see one possible example of equivalent and distinguishable states.



$\hat{\delta}(C, \epsilon) \in \mathcal{F},\ \hat{\delta}(G, \epsilon) \notin \mathcal{F} \Rightarrow C \not\equiv G$ ($\mathcal{F}$ finale states)

$\hat{\delta}(A, 01) = C \in \mathcal{F},\ \hat{\delta}(G, 01) = E \notin \mathcal{F} \Rightarrow A \not\equiv G$

We prove $A \equiv E$

$\hat{\delta}(A, 1) = F = \hat{\delta}(E, 1)$. Thus $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x) = \hat{\delta}(F, x)$, $\forall x \in \{0, 1\}^*$

$\hat{\delta}(A, 00) = G = \hat{\delta}(E, 00)$. Thus $\hat{\delta}(A, 00x) = \hat{\delta}(E, 00x) = \hat{\delta}(G, x)$, $\forall x \in \{0, 1\}^*$

$\hat{\delta}(A, 01) = C = \hat{\delta}(E, 01)$. Thus $\hat{\delta}(A, 01x) = \hat{\delta}(E, 01x) = \hat{\delta}(C, x)$, $\forall x \in \{0, 1\}^*$

Figure 2.32: Example

We can compute distinguishable state pairs using this recursive relation:
Base case: if $p \in F$ and $q \notin F$, then $p \neq q$
Induction steps: if $\exists a \in \Sigma : \delta(p, a) \neq \delta(q, a)$, then $p \neq q$. We use backward propagation to find distinguishable states. To find it we use **table filling algorithm**.

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
	A	B	C	D	E	F	G

Figure 2.33: Table filled

In the table shown in Figure 2.33 we start with all squares empty.
When a square is signed with an x means that the two states are distinguishable.
We start taking all pairs of states and giving as input ϵ: if they're distinguishable we put an x in the table. After this, we focus on the empty squares and giving as input a symbol $a \in \Sigma$ we put a x only if they're distinguishable.
We iterate this process until no new pair can be distinguished.
In the example we start from final state C, and we put an x on every cross with C for the base case.
After this, we choose two states that are distinguishable and if doing backward propagation the two states before are also distinguishable, we put an x. Now we can deal with **regular language equivalence**: let L and M be regular languages specified by means of some representation and we want to test if these two languages are equivalent. We have to:

- convert L and M representations into DFAs
- construct the union DFA
- apply state equivalence algorithm
- if the two start states are distinguishable, then $L \neq M$, otherwise $L = M$

In Figure 2.34 is shown an example of how it works. We have that $A = C$

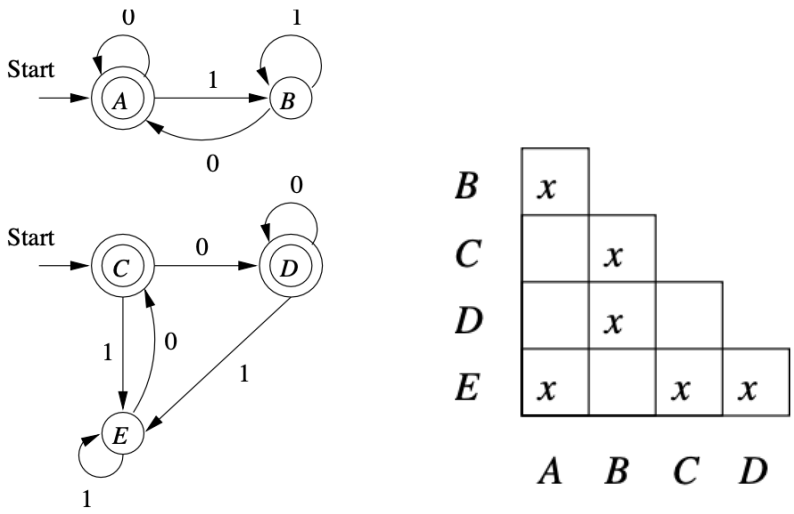


Figure 2.34: Equivalence of two regular language

thus the two DFAs are equivalent.
One important application of equivalence algorithm given a DFA as input, is to find an equivalent DFA with minimum number of states (that is unique).

The idea is to eliminate states that are unreachable from the initial and merge equivalent states into an individual one.
An important theorem is the **transitivity one**: if $p = q$ and $q = r$, then $p = r$.
The **proof** is quite simple. Suppose to the contrary that $p \neq r$: the $\exists w$ such that $\hat{\delta}(p, w) \in F$ and $\hat{\delta}(r, w) \notin F$. The first case $\hat{\delta}(q, w) \in F$ so $q = r$ and the second case $\hat{\delta}(q, w) \notin F$ so $p \neq q$ and it's not valid for hypothesis.
Theorem: To minimize DFA $A = (Q, \Sigma, \delta, q_0, F)$ we have to construct a DFA $B = (Q_=, \Sigma, \psi, q_{0,=}, F_=)$ where elements of $Q_=$ are the equivalence classes of =, elements of $F_=$ are the equivalence classes of = composed by states from F, $q_{0,=}$ is the set of states that are equivalent to q_0.
In Figure 2.35 there is an example of how it works this method.

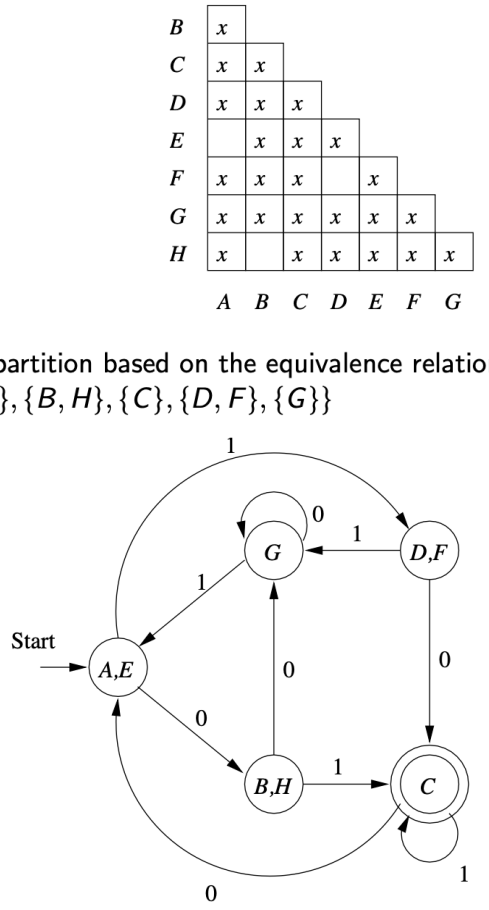


Figure 2.35: DFA minimization

We cannot **apply the algorithm to a NFA**.

3

Context Free Languages

3.1 Context-Free Grammars

We have seen that there are two types of devices, the recognition devices and the generative devices.
In the case of regular languages, the generative devices are regular expressions while the recognition devices are finite automatons.
Now we change our attention to **context-free languages** and more precisely to the recognition devices **push-down automata** and to the generative devices **context-free grammar**.
In the following part we can see an example, an informal example, that help us to understand CFL and CFG.
Let L_{pal} be the language that contain all the strings that are palindrome over an alphabet Σ. If we want to proof that is not regular it's enough apply pumping lemma choosing a string of this type: $w = 0^n 10^n$. It's easy to proof, by using pumping lemma, that is not regular.
We can inductively define L_{pal}:
Base: ϵ, 0, 1 are palindrome strings,
Induction: if w is palindrome, also $0w0$ or $1w1$ are palindrome.
CFG is a formal notation for recursively defining languages such as L_{pal} using **rewriting rules**.
A grammar consists in one or more variables, with each variable represent classes of strings, i.e languages. In this example we need for only one variable

which represent the palindrome strings, i.e this language.
In the Figure 3.1 below is shown the rules for this example.

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

Figure 3.1: Rules for a palindrome language

The first three rules form the basis, the last two the recursive call.
We can give now the definition of context-free grammar (CFG for short) that is a 4-tuple:

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

- V is a finite set of **variables** which each variable represent a language, i.e a set of strings.
- T is a finite set of **terminal symbols** representing the language alphabet
- P is a finite set of **productions** or **rules** that represent the recursive definition of a language. Each rule is in this form: $A \hookrightarrow \alpha$ where A is the head and it's a variable and α is the body and it's a string $\in (V \cup T)^*$ i.e. can be a terminal symbol or a variable. $\hookrightarrow$ is the production symbol.
- S is a variable called **initial symbol** and it's the variable that represent the language. Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol.

For the example seen before, in Figure 3.2 is shown the CFG for the palindrome strings.

$$G_{pal} = (\{P\}, \{0, 1\}, A, P)$$

with

$$A = \{P \rightarrow \epsilon, P \rightarrow 0, P \rightarrow 1, P \rightarrow 0P0, P \rightarrow 1P1\}$$

Figure 3.2: CFG for a palindrome language

There is another important example that is at the base of all compiler and it's

the CFG for **arithmetic expressions**.
Operands are **identifiers** i.e strings that start with a letter followed by numbers and letters. In this case identifiers are a small version, composed only by $\{a, b, 0, 1\}$ and are generated by the following regular expression:

$$(a + b)(a + b + 0 + 1)^*$$

There is obviously also arithmetic operators that in this case are only + and $*$.
We use this CFG:

$$G = (\{E, I\}, T, P, E)$$

where variable E represent arithmetic expressions and variable I represent identifiers.
E is also the starting variable or rather the variable which identify the language.
Variable I is an auxiliary variable used to compose expressions.
In Figure 3.3 are shown the rules.

P contains the following productions

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

Figure 3.3: Productions for arithmetic expressions

Commonly productions with a common head are grouped together: for example $A \hookrightarrow \alpha_1$, $A \hookrightarrow \alpha_2$ can be written as $A \hookrightarrow \alpha_1|\alpha_2|...$

3.1.1 Derivations

We have seen that productions is used to deduct that certain strings are in the language. Using productions is a procedure that is called **recursive inference** and it's the most conventional because we use rules from body to head, but, from a mathematical point of view, it's not accurate.
One very important thing in CFG is the concept of **derivation**: it is another approach to defining the language of a grammar, in which we use productions

from head to body, expanding the start symbol with a variable or a terminal and do it recursively.
In order to generate strings using a CFG we define a binary relation, a mathematical representation of the intuitive idea that variables are replaced with symbols, over $(V \cup T)^*$ called **rewrite or derivation**.
The process of deriving strings by applying productions requires the definition of a new relation symbol $\Rightarrow$.
Let $G = (V, T, P, S)$ be a CFG, $A \in V$, a variable, $\{\alpha, \beta\} \subseteq (V \cup T)^*$ strings. If there is a production of this type: $A \hookrightarrow \gamma$ then

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

and we say that $\alpha A \beta$ **derives in one step** $\alpha \gamma \beta$ and is said **derivation in one step**.
Notice that one derivation step replaces any variable anywhere in the string by the body of one of its productions. Moreover, the string can grows to the right and also to the left.
We can define $\Rightarrow^*$ notation that is the representation of **zero or more derivation steps** as follows:
Base: if $\alpha \in (V \cup T)^*$, then $\alpha \Rightarrow^* \alpha$
Induction: if $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \Rightarrow \gamma$ and this symbol is called **derivation**
We often write derivations by indicating all of the intermediate steps, as shown in Figure 3.4 below.

A possible derivation of $a * (a + b00)$ from E in the CFG for arithmetic expressions :

$$\begin{aligned}
E &\Rightarrow E * E & &\Rightarrow a * (E + I0)\\
&\Rightarrow E * (E) & &\Rightarrow a * (E + I00)\\
&\Rightarrow I * (E) & &\Rightarrow a * (E + b00)\\
&\Rightarrow a * (E) & &\Rightarrow a * (I + b00)\\
&\Rightarrow a * (E + E) & &\Rightarrow a * (a + b00)\\
&\Rightarrow a * (E + I)
\end{aligned}$$

Figure 3.4: Derivations of the example

In the example we can see that at first step, variable E is replaced by using a production of Figure 3.3.
In poor things we can say that recursive inference and derivation are the same:

the first one uses productions in a recursive way to produce a final string that is in the language, derivation, at each step, uses a production to produce a final string that is in the language.
As we can see in the example when we compute an expression we can apply derivation (single) relation in a different way, and not all choices lead to the desired string as shown in Figure 3.5.

$$I * E \Rightarrow a * E \Rightarrow a * (E)$$
$$I * E \Rightarrow I * (E) \Rightarrow a * (E)$$

Not all choices lead to a derivation of the desired string :

$$I * E \Rightarrow a * E \Rightarrow a * E + E$$

does not lead to a derivation of $a * (a + b00)$

Figure 3.5: Different derivations for the same string

In derivations, we can avoid having to choose the variables to rewrite if we give a rule that bring to a **canonical and unique** derivation form.
We can talk about **leftmost and rightmost derivation** represent respectively with the symbols $\Rightarrow_{lm}$, $\Rightarrow_{rm}$.
The relation $\Rightarrow_{lm}$ indicate that we always rewrite the leftmost variable with some productions. We can also use the concept of derivations applied here.
Obviously it's analogue of rightmost derivation. In Figure 3.6 there is an example of leftmost and right most derivation.

Leftmost derivation of $a * (a + b00)$:

$$\begin{aligned} E &\underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E \underset{lm}{\Rightarrow} a * (E) \underset{lm}{\Rightarrow} a * (E + E) \\ &\underset{lm}{\Rightarrow} a * (I + E) \underset{lm}{\Rightarrow} a * (a + E) \underset{lm}{\Rightarrow} a * (a + I) \underset{lm}{\Rightarrow} a * (a + I0) \\ &\underset{lm}{\Rightarrow} a * (a + I00) \underset{lm}{\Rightarrow} a * (a + b00) \end{aligned}$$

Rightmost derivation :

$$\begin{aligned} E &\underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E) \underset{rm}{\Rightarrow} E * (E + I) \\ &\underset{rm}{\Rightarrow} E * (E + I0) \underset{rm}{\Rightarrow} E * (E + I00) \underset{rm}{\Rightarrow} E * (E + b00) \\ &\underset{rm}{\Rightarrow} E * (I + b00) \underset{rm}{\Rightarrow} E * (a + b00) \underset{rm}{\Rightarrow} I * (a + b00) \\ &\underset{rm}{\Rightarrow} a * (a + b00) \end{aligned}$$

Figure 3.6: Example

From the moment that a CFG generates strings, it also generates a language:

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$$

that is the set of all strings in T^* that can be derived from the starting symbol.
L(G) is a **context-free language** of CFL for short.
The class of context free language are CFL and in Figure 3.7 there is diagram that shows that regular languages are always context free languages but there are some CFL that are not regular.

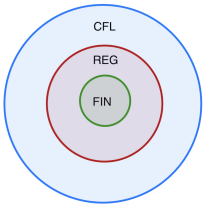


Figure 3.7: Classes of languages

Let $G = (V, T, P, S)$ be a CFG and α a string $\in (V \cup T)^*$. We can talk about **sentential form** that is when we associate after zero or more steps of derivation string α to variable S, that is our starting symbol.
In Figure 3.8 are shown examples of sentential form.

Then $E * (I + E)$ is a sentential form, since

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

This derivation is neither leftmost nor rightmost

$a * E$ is a leftmost sentential form, since

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$$

$E * (E + E)$ is a rightmost sentential form, since

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E)$$

Figure 3.8: Example of sentential form

Note that $L(G)$ is those sentential form that are in T^* i.e. they consists only of terminals.
There are two important property regarding CFG that are:

Derivation composition: we can also compose two derivations $A \Rightarrow^* \alpha B \beta$ and $B \Rightarrow^* \gamma$ into a single derivation

$$A \Rightarrow^* \alpha B \beta \Rightarrow^* \alpha \gamma \beta$$

In Figure 3.9 below there is an example of derivation composition.

$$E \Rightarrow E + E \Rightarrow E + (E)$$
$$E \Rightarrow I \Rightarrow Ib \Rightarrow ab$$

we can compose at the rightmost occurrence of E, obtaining

$$E \Rightarrow E + E \Rightarrow E + (E) \Rightarrow E + (I) \Rightarrow E + (Ib) \Rightarrow E + (ab)$$

Figure 3.9: Example of derivation composition

The second important property for CFG is **derivation factorization**.
Assume A, a variable, be $A \Rightarrow X_1 X_2 ... X_k \Rightarrow^* w$ where various X_i may be variable or symbol from the alphabet. We can factorize w as $w_1 w_2 ... w_k$ such that

$$X_i \Rightarrow^* w_i \; \forall 1 \neq i \neq k$$

In other word, all piece w_i derived by a variable or a special case, a symbol when $X_i = w_i$, in some steps.
Substring w_i can be identified by considering only derivation steps that rewrite X_i.
In the Figure 3.10 below there is an example.

$$\underbrace{a}_{E} \; \underbrace{*}_{*} \; \underbrace{b + a}_{E}$$

and we can write

$$E \overset{*}{\Rightarrow} a$$
$$* \overset{*}{\Rightarrow} *$$
$$E \overset{*}{\Rightarrow} b + a$$

Figure 3.10: Example of derivation factorization

Notice that could be a confusing example since $* \Rightarrow^* *$. Leftmost * represent the mathematical operator, that is derived in zero or more step (second *) in itself.

3.1.2 Parse Tree

Parse trees are graphical representation alternative to derivations and represent also the syntactic structure of a sentence.
Let $G = (V, T, P, S)$ be a CFG. An ordered tree is a **parse tree** of G that:

- each internal node is labeled with a variable in V
- each leaf is labeled with a symbol in $(T \cup \epsilon)$ and each leaf labeled with ϵ is the only child of its parents
- if an internal node is labeled with A and its childre, from left to right, are labeled with $X_1,\ X_2,\ ...,\ X_k$ then $A \hookrightarrow X_1X_2...X_k \in P$

It's useful a recap for the concept of tree:

- Trees are collections of nodes with a parent-child relationship. A node has at most one parent drawn above the node and zero or more children drawn below. Lines connect parents to their children
- There is one node, the root, that has no parent; this node appears at the top of the tree. Nodes with no children are called leaves. Nodes that are not leaves are interior nodes.
- The children of an internal node are ordered from left to the right

In Figure 3.11 below there is parse tree associated with the derivation $P \Rightarrow^* 0110$.

CFG for palindrome strings and parse tree associated with the derivation $P \Rightarrow 0P0 \Rightarrow 01P10 \Rightarrow 0110$

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

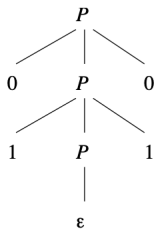


Figure 3.11: Parse tree of 0110

There is another important concept that is **yield of a parse tree** that is the string obtained by reading the leaves from the left to the right, and it's a string $(\in V \cup T)$ always derived by the root.
Of special importance are the **complete** parse trees which the yield is a string of terminal symbols and the root is labeled by initial symbol.
The set of yields of all complete parse trees is the language generated by the

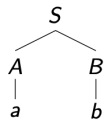
CFG.

Derivations and Parse Trees: Let $G = (V, T, P, S)$ be a CFG with $A \in V$ a variable and $w \in T^*$ a string. The following statements are equivalent:

- $A \Rightarrow^* w$
- $A \Rightarrow^*_{lm} w$
- $A \Rightarrow^*_{rm} w$
- there exist a parse tree with root $A$and yield w

But is so important say that relation between derivations and parse trees **is not one to one**. We can explain this thing well with some examples.
A parse tree can be associated with **many** derivations as the example in Figure 3.12 shows.

Example : Consider the CFG with productions $S \to AB$, $A \to a$, $B \to b$. The parse tree



is associated with two derivations

$$S \Rightarrow AB \Rightarrow aB \Rightarrow ab$$
$$S \Rightarrow AB \Rightarrow Ab \Rightarrow ab$$

Figure 3.12: Single tree implies two derivations

Also, for the same reason, a derivation can be associated with **many** parse trees, as the example in Figure 3.13 shows.

Example : Consider the CFG with productions $S \to SS \mid a$ The derivation

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow aSS \Rightarrow aaS \Rightarrow aaa$$

is associated with two parse trees

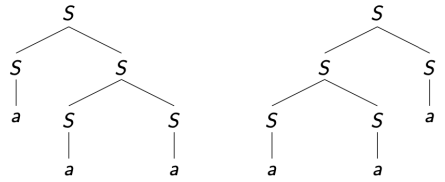


Figure 3.13: Single derivation implies two trees

3.1.3 Ambiguos CFGs

Ambiguous CFGs means that for a string there are different ways to derive it / rewrite it.
In Figure 3.14 there is an example for arithmetic expressions language that generates same string with different derivations, with different result in numerical terms.

the sentential form $E + E * E$ has two derivations

$$E \Rightarrow E + E \Rightarrow E + E * E$$
$$E \Rightarrow E * E \Rightarrow E + E * E$$

Figure 3.14: Same string with different derivations

So there are two different parse trees for the derivations of $E + E * E$ as we can see in Figure 3.15.

Associated parse trees for the derivations of $E + E * E$

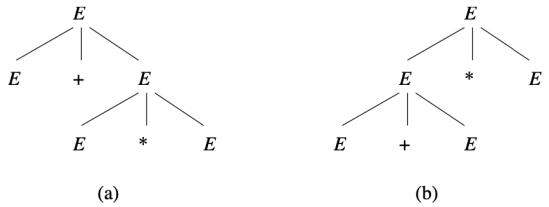


Figure 3.15: Same string with different trees

One is expanding to the right and the other to the left. The two different derivations correspond to different **precedences** for operator: in the first we have to compute first $*$ operator, in the second we have to comopute first + operator.
The existence of different derivations for a string is not problematic, if these correspond to a single parse as we can see in the example in Figure 3.16.

Example : In our CFG for arithmetic expressions, the string $a + b$ has at least two derivations

$$E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$$
$$E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$$

Figure 3.16: Different derivations with same tree

In the example we generate the same string in two different ways, so the string

$a + b$ is not ambiguous, according to the equivalence of derivation and trees seen before (is equivalent to say derivation and parse tree with yield that is a string w and root E).
Here the definition of **ambiguous**: let $G = (V, T, P, S)$ be a CFG.
G is ambiguous if there exists a string in $L(G)$ such with one or more different trees. If there isn't string of this type, G is said to be **unambiguous**.
The ambiguity is problematic in many applications where the syntactic structure (parse tree) of a sentence is used to interpret its meaning so different trees implies different meaning.
In Figure 3.17 the terminal string $a + a * a$ has two parse trees so it's ambiguous.

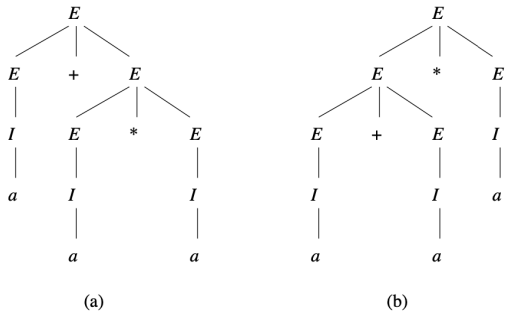


Figure 3.17: Ambiguous string

In precedence we have treated the meaning of leftmost derivation, that identifies a string with fixed derivation step: a parse tree is associated with a **unique** leftmost derivation and a leftmost derivation is associated with a **unique** parse tree.
Two different derivations can have the same trees but two different leftmost derivation must have different trees.
This concept can be obviously extended to rightmost derivation.

When a grammar fails to provide unique structures, it is sometimes possible to redesign the grammar to make the structure unique for each string in the language. Unfortunately, sometimes we cannot do so.
A CFL is **inherently ambiguous** when every CFG such that $L(G) = L$ is ambiguous, so there is not a way to make the grammar that has not some ambiguous strings.

As we can see in the example in Figure 3.18, we have the union of two languages $L_1 \cup L_2$, with the following productions that generate respectively L_1 and L_2.

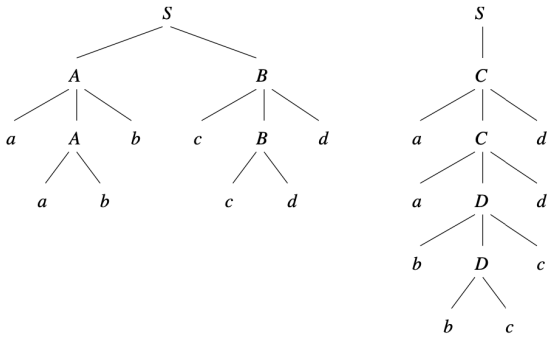
Example : Let us consider the language

$$L = \{a^n b^n c^m d^m \mid n \geqslant 1,\ m \geqslant 1\} \cup \{a^n b^m c^m d^n \mid n \geqslant 1,\ m \geqslant 1\}$$

L can be generated by a CFG with the following productions

$$\begin{aligned} S &\to AB \mid C \\ A &\to aAb \mid ab \\ B &\to cBd \mid cd \\ C &\to aCd \mid aDd \\ D &\to bDc \mid bc \end{aligned}$$

There are two parse trees for the string $aabbccdd$



(a) (b)

Associated leftmost derivations

$$S \underset{lm}{\Rightarrow} AB \underset{lm}{\Rightarrow} aAbB \underset{lm}{\Rightarrow} aabbB \underset{lm}{\Rightarrow} aabbcBd \underset{lm}{\Rightarrow} aabbccdd$$

$$S \underset{lm}{\Rightarrow} C \underset{lm}{\Rightarrow} aCd \underset{lm}{\Rightarrow} aaDdd \underset{lm}{\Rightarrow} aabDcdd \underset{lm}{\Rightarrow} aabbccdd$$

Figure 3.18: Example of inherently ambiguous language

However the intersection of this two languages is not empty because $a^n b^n c^n d^n$ is in both languages.
So it's possible to show that every CFG generating L provides similar ambiguity for the string $aabbccdd$ so language L is inherently ambiguous.

3.1.4 From RL to CFG

A **regular language** is **always a CFL** as we can see in Figure 3.7.
From a regular expression of from an FA we can always construct a CFG generating same language.
From regular expression to CFG: let E be any regular expression. We use a

variable for E (starting symbol) and a variable for each subexpression of E.
After we use **structural induction** on the regular expression to build the productions of our CFG as follows:

- if E = **a**, then add production $E \hookrightarrow a$ (base)
- if E = ϵ, then add production $E \hookrightarrow \epsilon$ (base)
- if E = $\emptyset$, then production set is empty
- if E = F + G, then add production $E \hookrightarrow F|G$ (induction)
- if E = FG, then add production $E \hookrightarrow FG$ (induction)
- if E = F^*, then add production $E \hookrightarrow FE|\epsilon$ (induction)

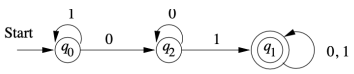
As we can see in the example of Figure 3.19, it's simple convert a regular expression $0^*1(0+1)^*$ to a CFG.

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

Figure 3.19: Example of conversion of RegEx in to a CFG

From FA to CFG: It's also possible convert a finite automata in a CFG. We use a variable Q for each state q of the FA with the initial symbol is Q_0.
For each transition from state p to state q under symbol a, add a production $P \hookrightarrow aQ$ and at the end, if q is a final state, add production $Q \hookrightarrow \epsilon$. We can see an example in Figure 3.20 below.

Automaton :



CFG :

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

String 1101 is accepted by the automaton. In the equivalent CFG, 1101 has the following derivation :

$$Q_0 \Rightarrow 1Q_0 \Rightarrow 11Q_0 \Rightarrow 110Q_2 \Rightarrow 1101Q_1 \Rightarrow 1101$$

Figure 3.20: Example of conversion of DFA in to a CFG

3.2 Push-Down Automata

We start with an intuitive idea of what is push-down automata.
As we anticipate before PDAs are recognition devices that say if a string is in the CFL.
We will see later that PDA and CFG are equivalent.
CFGs are high level, difficult to implement, PDAs are low level, easy to implement but difficult to manipulate.
A push-down is composed by an $\epsilon - NFA$ and a **stack**, representing the auxilary memory. The stack record an arbitrary number of symbols and release symbols with **LIFO** policy: Last In, First Out.
In a single transition, a PDA:

- Read and consumes a single symbol from the input or else stays as in the ϵ-transition
- Updates the current state
- Updates the top-most symbol in the stack and replaces it with a **string**, that could also be a the empty string ϵ

The first two steps are the usual of an ϵ-NFA, the third is the new part.
As we can see in Figure 3.21 below, the first part is a simple ϵ-NFA with the add of a second part, a stack.

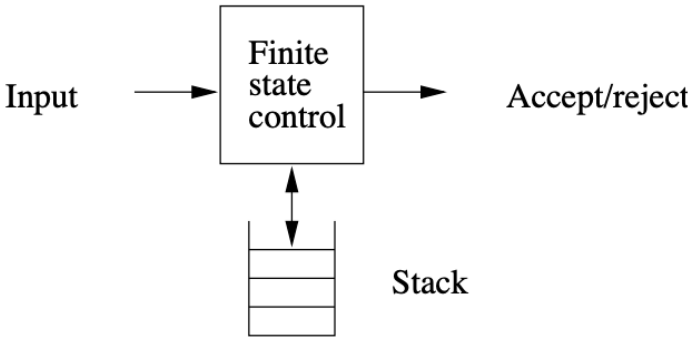


Figure 3.21: How is composed a PDA

More precisely, replacement of symbol X in the stack top-most position with string α, the stack symbols string, amounts to:

- if $\alpha = \epsilon$, then we simply remove X. This operation is called **pop**
- if $\alpha = Y$, we replace X with Y. This operation is called **switch**

- if $|\alpha| > 1$, then we push these new symbols in the stack. If $\gamma = ZX$, this operation is called **push** and correspond to simply insert on the top Z

We can give an example: let us consider the language of palindrome strings with even length:

$$L_{wwr} = \{ww^R \mid w \in \{0,1\}^*\}$$

that is generated by the following CFG productions: $P \hookrightarrow 0P0$, $P \hookrightarrow 1P1$, $P \hookrightarrow \epsilon$.

A PDA for L_{wwr} has three states, three phases in the computation and operates as follows.

In the first phase we are in q_0 and we start read w, taking as input the first symbol. We push the symbol read into the top of the stack and we remain in q_0, until we don't move in the second phase.

At some point, we move from first phase to second phase, going from q_0 to q_1, with an ϵ-transition. We do it when we are at the middle between w and w^R.

We're now reading the first symbol of w^R: we have to compare it with the top of the stack and if the two symbols match, pop the top-most symbol of the stack and remain in state q_1. If the two symbols are different, the automaton **halts** i.e has not next move.

When the stack is empty, go to state q_2 and **accept**.

A **push-down automaton** is a tuple:

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

such that

- Q defined as for NFAs
- Σ defined as for NFAs
- q_0 defined as for NFAs
- δ is the transition function $\delta : (Q\ x\ (\Sigma \cup \{\epsilon\})\ x\ \Gamma) \hookrightarrow\ 2^{Qx\Gamma^*}$ where $2^{Qx\Gamma^*}$ is the finite set of all subsets of Q, all the possible combination of the states. The union with the alphabet of the stack means that PDA has to read also a symbol of the stack alphabet. It means that as input there is a triple of this form (q_j, a, X).
  The output is a set of pairs (q_i, α) with α the string that we use to replace the top of the stack.
  This set may be infinite since Γ^* but we restrict it to a finite numbers of pairs
- Γ finite **stack alphabet**, a specialized alphabet, usually use new symbols with special meaning

- $Z_0 \in \Gamma$ is the initial symbol stack, used only at the bottom of the stack, the end of the stack
- F defined as for NFAs

The PDA for L_{wwr} is shown in Figure 3.22 below.

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

Figure 3.22: Table representation of a PDA

We've seen that δ takes as input a triple: on the left there are the states, on the top there are the possible inputs that complete the triple: (a, X) i.e a symbol $\in \Sigma$ and the top most symbol of the stack.
The cross between the state in which we are and the remain part of the input is the state in which we have to move and the new stack string. In this PDA, the last three instructions of the first row mean that we are in the middle of the string.
It can be also represented in a **graphical notation** using the convention that $(p, \alpha) \in \delta(q, a, X)$ is associated with an arc from state q to state p labeled with $a, X/\alpha$, with α string of stack symbols, as we can see in Figure 3.23.

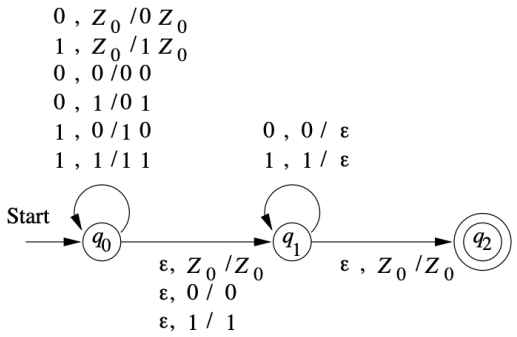


Figure 3.23: Diagram representation of a PDA

We've to deal with the concepts of **instantaneous description** and **computation**. Informally, an instantaneous description is a "configuration" of the PDA. When

we read a symbol and the top-most symbol in the stack, we move to another instantaneous description. A sequence of moves, from a configuration to another, is called computation.
To formalize the computation of a PDA we then introduce a binary relation over instantaneous description called **moves**.
Instantaneous description or ID for short is a triple:

$$(q, w, \gamma)$$

where q is the current state, w is the part of input still to be read and γ is the stack content, with top-most symbol at the left. With one moves we go from one ID to another.
Moves: is a binary relation over the set of IDs ($ID \hookrightarrow ID$) written as $\vdash$:

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

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

We try to explain the two sentences.
The first one is the case in which symbol a is read: among many pairs of possible moves that we can perform, reading as input a, being in the state q, with top-most symbol of the stack X, we choose the couple (p, α), i.e. going to the state p, replacing the top-most symbol of the stack with α.
This is the **moves** from the ID that is in state q, with input ax and stack content $X\beta$ and goes to ID with state p, string w (a is consumed) and stack content $\alpha\beta$ with α that replaces X.
The second one is the similar: no symbol is read, no input is consumed so the input that we have in the next instantaneous description is the same.
A set of moves can be united in a **computation** that is represented by this symbol $\vdash^*$.
In Figure 3.24 there are the set of moves done by the PDA described below, when the input is the string $w = 1111$.
As we can see in the first transition, there is a branch: in the left-most one, it's read a 1, in the right-most one, any symbol is consumed.
The correct computation is highlighted in red.
The only computation that ends correctly is the red highlighted.

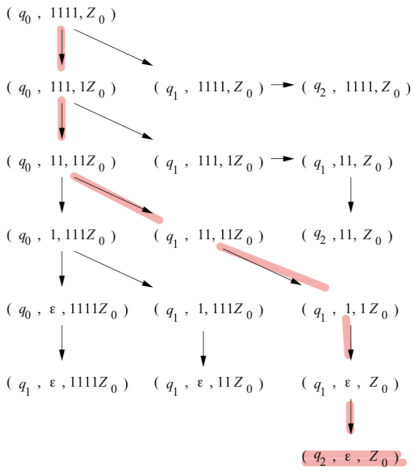


Figure 3.24: Computation of string 1111

3.2.1 Properties of computation

When a PDA reads an input string, it starts from a stack symbols string γ and it can't go under this string: it can only add symbols and remove these symbols. Moreover, intuitively, input symbols that are not read by the PDA do not affect the computation. We can enunciate three properties in an informal way and after give the mathematical proposition of these three properties:

- if an ID sequence (computation, more moves) is valid, also the sequence obtained by adding any string to the tail of the input is valid
- if an ID sequence is valid, also the sequence obtained by adding any string to the bottom of the stack is valid
- if an ID sequence is valid and some tail of the input is not consumed, also the sequence obtained by removing that tail in every ID sequence is valid

Theorem 1: $\forall w \in \Sigma^*,\ \gamma \in \Gamma^*$:

$$(q, x, \alpha) \vdash^* (p, y, \beta) \Rightarrow (q, xw, \alpha\gamma) \vdash^* (p, yw, \beta\gamma)$$

if $\gamma = \epsilon$ we get the first property, if $w = \epsilon$ we get the second property
Theorem 2: $\forall w \in \Sigma^*$:

$$(q, xw, \alpha) \vdash^* (p, yw, \beta) \Rightarrow (q, x, \alpha) \vdash^* (p, y, \beta)$$

This is the third property that we have described informally above.

3.2.2 Languages accepted by a PDA

There are two types of language accepted by a PDA:

Acceptance by final state: The language accepted by final state by P is

$$L(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \epsilon, \alpha),\ q \in F\}$$

that means that the stack doesn't necessarily need to be empty at the end of the computation but has to be in a final state.

Acceptance by empty stack: The language accepted by empty stack by P is

$$N(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon)$$

for any state q, with q that doesn't necessarily need to be final.

There are two important theorems which proves the equivalence of a language accepted by empty stack or by final states.

Theorem 1: from empty stack to final state: If $L = N(P_N)$ for some PDA $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0)$ that has acceptance by empty stack, then exists a PDA $P_F = (Q \cup \{p_0, p_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$ such that $L = L(P_F)$ that has acceptance by final state

Proof: Before going throw the mathematical details, in the Figure 3.25 there is the main idea of the structure.

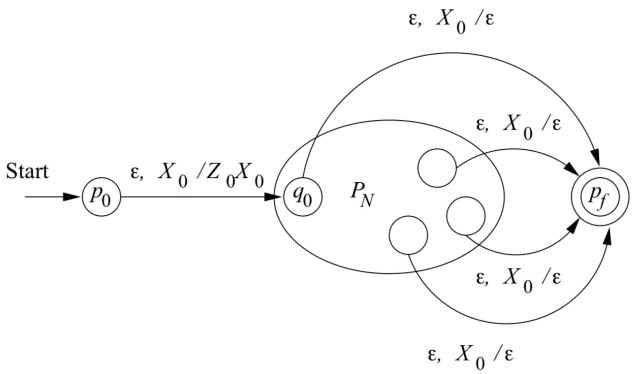


Figure 3.25: Construction of the idea for the proof

The input automaton is P_N that accepts empty stack. It has an input state and no final. This automaton is also part of the new automaton.

We add a new initial state p_0 and and we put over the initial stack symbol X_0 the start symbol Z_0 of automaton P_N; after that, we simulate the computation of

P_N using its transition and when we reach again X_0 as last symbol of the stack means that for P_N the stack is empty so we do an ϵ-transition moving to a final accepted state p_f. Under this idea, there is the mathematical proof:

- $\delta_F(p_0, \epsilon, X_0) = \{q_0, Z_0X_0\}$ and it's the first step
- each transition, with stack not empty, of δ_N is equal to δ_F
- for each $q \in Q$ we add $(p_f, \epsilon) \in \delta_F(q, \epsilon, X_0)$

$\supseteq$: Let $w \in N(P_N)$, then

$$(q_0, w, Z_0) \vdash_N^* (q, \epsilon, \epsilon)$$

and from a previous theorem

$$(q_0, w, Z_0X_0) \vdash_N^* (q, \epsilon, X_0)$$

but being $\delta_N \subset \delta_F$

$$(q_0, w, Z_0X_0) \vdash_F^* (q, \epsilon, X_0)$$

and we can conclude that

$$(p_0, w, X_0) \vdash_F (q_0, Z_0X_0) \vdash_F^* (q, \epsilon, X_0) \vdash_F^* (q, \epsilon, \epsilon)$$

$\subseteq$: By inspecting the diagram of P_F, any accepting computation for w in P_F embeds an accepting computation for w in P_N

Theorem 2: from final state to empty stack: If $L = N(P_F)$ for some PDA $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$ that has acceptance by final state then exists a PDA $P_N = (Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_N, p_0, X_0)$ such that $L = N(P_N)$ that has acceptance by empty stack

Proof:As the last theorem, we construct a diagram that we can see in Figure 3.26.

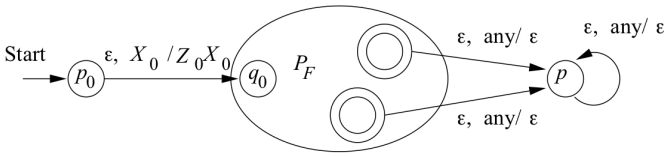


Figure 3.26: Construction of the idea for the proof

In the central part of the diagram we simulate P_F, reaching some final states.

We add two part: the first part is as before adding a symbol Z_0 on the top of the stack that is the initial stack symbol for P_F that allows us to simulate the computation of P_F. When we reach the final state, we remove any symbol on the top of the stack that bring the automaton to a state that have a loop which removes all the symbols in the stack until we reach an empty stack.
Under this idea, there is the mathematical proof:

- $\delta_N(p_0, \epsilon, X_0) = \{q_0, Z_0X_0\}$ and it's the first step
- each transition, with stack not empty, of δ_F is equal to δ_N
- for each $q \in F$ final, we add $(p, \epsilon) \in \delta_N(q, \epsilon, Y)$ for each stack symbol Y

$\supseteq$: By inspecting the diagram of P_N, any accepting computation for w in P_N embeds an accepting computation for w in P_F
$\subseteq$: Let $w \in L(P_F)$, then

$$(q_0, w, Z_0) \vdash_F^* (q, \epsilon, \alpha)$$

with q final and $\alpha \in \Gamma^*$.
Since $\delta_F \subseteq \delta_N$ and for a previous theorem that says that adding something at the bottom of the stack doesn't change anything

$$(q_0, w, Z_0X_0) \vdash_N^* (q, \epsilon, \alpha X_0)$$

and we can conclude that

$$(p_0, w, X_0) \vdash_N (q_0, Z_0X_0) \vdash_N^* (q, \epsilon, \alpha X_0) \vdash_N^* (p, \epsilon, \epsilon)$$

3.2.3 Languages between CFG and PDA

Let L be a language. The following statement are equivalent:

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

We've already seen the equivalence between empty stack and final state. Now we see the equivalence from a CFG to a PDA.
In Figure 3.27 is shown a construction for well understand this principle informally.

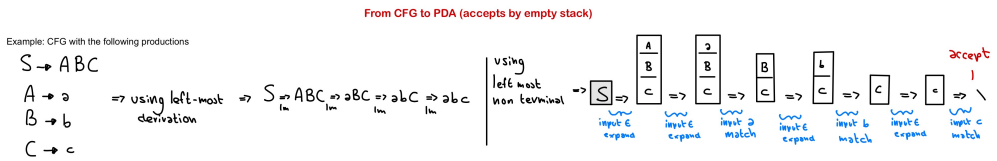


Figure 3.27: From CFG to PDA: main idea

Formally let $G = (V, T, R, S)$ be some CFG. We define

$$(\{q\}, T, V \cup T, \delta, q, S)$$

where

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

If all nondeterministic choices are **correct**, P_G completes the processing of the input with an empty stack.
In Figure 3.28 below is shown an example for arithmetic expression.

$$\begin{aligned} I &\rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ E &\rightarrow I \mid E * E \mid E + E \mid (E) \end{aligned}$$

The transition function of the PDA is

$$\begin{aligned} \delta(q, \epsilon, I) &= \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\} \\ \delta(q, \epsilon, E) &= \{(q, I), (q, E * E), (q, E + E), (q, (E))\} \\ \delta(q, X, X) &= \{(q, \epsilon)\}, \quad \forall X \in \{a, b, 0, 1, (,), +, *\} \end{aligned}$$

Figure 3.28: Example of conversion from CFG to PDA

Theorem 1: $N(P_G) = L(G)$
Theorem 2: A language generated by a PDA is equal to a language generated by a CFG

3.3 CFG SEMPLIFICATION

3.3.1 ELIMINATION OF USELESS SYMBOLS

Let G be some CFG. We can eliminate some grammatical symbols and some productions preserving the generated language.
Useless symbols: Assume a CFG $G = (V, T, P, S)$. Symbol $X \in (V \cup T)$ is called:

- **Reachable** if there exists a derivation from starting symbol $S \Rightarrow^* \alpha X \beta$ for some α, β
- **Generating** if there exists a derivation $X \Rightarrow^* w$ for some $w \in T^*$
- **Useful** if it is generating and reachable, otherwise is **useless**

We don't know the set of generating and reachable symbols. There are two **algorithm** used to find them.
The algorithm for generating symbols is the following: we compute the set of $g(G)$ generating symbols by means of the following inductive algorithm
Base: $g(G) \hookleftarrow T$
Induction : if $A \hookrightarrow X_1 X_2 ... X_n \in P$ and $X_i \in g(G)$ for each $1 \neq i \neq n$ then $g(G) \hookleftarrow g(G) \cup \{A\}$
In Figure 3.29 is shown an example.

Consider the CFG G with productions

$$S \to AB \mid a$$
$$A \to b$$

At the base step we have $g(G) = \{a, b\}$

From $S \to a$ we add S to $g(G)$; from $A \to b$ we add A a $g(G)$.
No other production can contribute to set $g(G)$

We thus have $g(G) = \{S, A, a, b\}$

Figure 3.29: Example of computation of $g(G)$

The algorithm for reachable symbols is the following: we compute the set of $r(G)$ reachable symbols by means of the following inductive algorithm
Base: $r(G) \hookleftarrow \{S\}$
Induction : if $A \hookrightarrow X_1 X_2 ... X_n \in P$ and $A \in r(G)$ for each $1 \neq i \neq n$ then $r(G) \hookleftarrow r(G) \cup \{X_i\}$

Consider the CFG G with productions

$$S \to AB \mid a$$
$$A \to b$$

At the base step we have $r(G) = \{S\}$

From $S \to AB$ we add A and B to $r(G)$.
From $S \to a$ we add a to $r(G)$.
From $A \to b$ we add b to $r(G)$

We thus obtain $r(G) = \{S, A, B, a, b\}$

Figure 3.30: Example of computation of $r(G)$

In Figure 3.30 is shown an example.
By merging these two algorithm, born the algorithm used to remove useless productions, Formalizing it, given as input a CFG G we have to:

- build G_1 by eliminating from G all non-generating symbols and all productions in which they appear
- build G_2 by eliminating from G_1 all non-reachable symbols and all productions in which they appear

In the example below in Figure 3.31 is shown the procedure to eliminate some productions.

Consider the CFG G with the following productions

$$S \to AB \mid a$$
$$A \to b$$

S, A, a e b are generating, B is not generating

In order to eliminate B we need to eliminate the production $S \to AB$, resulting in the new grammar

$$S \to a$$
$$A \to b$$

Now only S and a are reachable

After eliminating A and b, the resulting grammar has the only production $S \to a$

Figure 3.31: Example of removing of useless symbol

3.3.2 Elimination of ϵ-production

When we deal with an ϵ-productions we intent a production of this type:

$$A \hookrightarrow \epsilon$$

where A is a non terminal.
There is an important observation that we have to do: if $\epsilon \in L$ we cannot eliminate ϵ-productions to preserve the generated language.
Obviously, if there is a language without ϵ-productions, ϵ cannot be in the language.
If language L has a CFG, then $L-\{\epsilon\}$ has a CFG without ϵ-productions. If already ϵ is not in the language L, then L itself has a CFG without ϵ-productions.
An important concept is when a variable A is **nullable**: a variable A is nullable if $A \Rightarrow^* \epsilon$.
The idea is that we know that is possible in original language have this situation and we want to do a sort of preprocessing for removing ϵ-productions: if A is nullable and there exists a production $B \hookrightarrow CAD$ then we:

- directly **remove** productions with ϵ in right side
- we construct two alternative version of the above production in this way

  $$B \hookrightarrow CD$$

  for A that generates ϵ

  $$B \hookrightarrow CAD$$

  for A that generates other string.
  If also C and B are nullable, we have to remove all possible combinations of C, A and D, from production $B \hookrightarrow CAD$ as we can see in Figure 3.32

```
B -> CAD
B -> CD
B -> AD
B -> CA
B -> C
B -> A
B -> D
```

Figure 3.32: Removing of nullable symbols

The algorithm for nullable variables of $G = (V, T, P, S)$ is the following: we compute the set of $n(G)$ nullable symbols by means of the following inductive

algorithm
Base: $n(G) \hookleftarrow \{A \mid (A \hookleftarrow \epsilon) \in P\}$
Induction : if there exist in G a production $C \hookrightarrow B_1B_2...B_k$ such that $B_i \in n(G)$ for each i then

$$n(G) \hookleftarrow n(G) \cup \{C\}$$

so also C is nullable.
Note that we have that all B_i in the bodies of C are variables.
Now that we have the set $n(G)$ we can build a new CFG $G_1 = (V.T, P_1, S)$ where P_1 is computed starting from P as follows:

- each production $(A \hookrightarrow \epsilon) \in P$ is removed from P_1
- given a production $A \hookrightarrow X_1X_2...X_k$ we define a set $N = \{i_1, i_2, ..., i_m\}$ composed by the index of nullable variables X_i
- for every possible choice of set $N' \subset N$, that are 2^m, we add to P_1 a production constructed starting from the initial production by removing each X_i one for time, as shown in Figure 1.68 for the previous case

In Figure 3.33 is shown an example.

Elimination of ϵ-production from CFG G with productions

$$S \to AB$$
$$A \to aAA \mid \epsilon$$
$$B \to bBB \mid \epsilon$$

We first compute set $n(G)$

- $A, B \in n(G)$ since $A \to \epsilon$ and $B \to \epsilon$
- $S \in n(G)$ since $S \to AB$, with $A, B \in n(G)$

From $S \to AB$ we construct the new productions $S \to AB \mid A \mid B$

From $A \to aAA$ we construct the new productions
$A \to aAA \mid aA \mid a$

From $B \to bBB$ we construct the new productions
$B \to bBB \mid bB \mid b$

The resulting CFG G_1 has productions

$$S \to AB \mid A \mid B$$
$$A \to aAA \mid aA \mid a$$
$$B \to bBB \mid bB \mid b$$

and we have $L(G_1) = L(G) \setminus \{\epsilon\}$

Figure 3.33: Example of removing of ϵ-productions

Remember: do not remove all nonterminals symbols from the right-hand side, in case this leads to an epsilon-production. Therefore epsilon-productions are never produced by the construction, they are only eliminated.

3.3.3 Elimination of unary productions

Let $G = (V, T, P, S)$ be some CFG. A **unary** production has the form $A \hookrightarrow B$, where both A and B are variables $\in V$.
We can eliminate unary productions by expanding the variables in the right-hand side. This method couldn't work when there is a cycle of variable as $A \hookrightarrow B,\ B \hookrightarrow C,\ C \hookrightarrow A$.
We have a method based on the notion of unary pairs which eliminates the unary productions in the general case.
Let $G = (V, T, P, S)$ be a some CFG. (A, B) is a **unary pair** if $A \Rightarrow^* B$ using only unary productions.
The algorithm for unary pairs of $G = (V, T, P, S)$ is the following: we compute the set of $u(G)$ unary pairs of G by means of the following inductive algorithm
Base: $u(G) \hookleftarrow \{(A, A) \mid (A \in V)\}$
Induction : if $(A, B) \in u(G)$ and $(B \hookrightarrow C) \in P$

$$u(G) \hookleftarrow u(G) \cup \{(A, C)\}$$

In Figure 3.34 is shown an example.

Consider the CFG

$$\begin{aligned} I &\to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ F &\to I \mid (E) \\ T &\to F \mid T * F \\ E &\to T \mid E + T \end{aligned}$$

In the base step we derive the unary pairs (E, E), (T, T), (F, F) e (I, I)

In the inductive step

- from (E, E) and $E \to T$ we add pair (E, T)
- from (E, T) and $T \to F$ we add pair (E, F)
- from (E, F) and $F \to I$ we add pair (E, I)
- from (T, T) and $T \to F$ we add pair (T, F)
- from (T, F) and $F \to I$ we add pair (T, I)
- from (F, F) and $F \to I$ we add pair (F, I)

Figure 3.34: Example of computation of $u(G)$

Now that we have the set $u(G)$, we can build a new CFG $G_1 = (V, T, P_1, S)$ where P_1 is computed starting from P as follows:

- compute $u(G)$
- for each $(A, B) \in u(G)$ and for each $(B \hookrightarrow \alpha) \in P$ which is not a unary production, add to P_1 the production $A \hookrightarrow \alpha$

In the second step we might have $A = B$: this case means that P_1 contains all the non unary productions of P.
In Figure 3.35 we can see an example: we have already computed set $u(G)$ in the previous example.

Pair	Productions
(E, E)	$E \to E + T$
(E, T)	$E \to T * F$
(E, F)	$E \to (E)$
(E, I)	$E \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(T, T)	$T \to T * F$
(T, F)	$T \to (E)$
(T, I)	$T \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(F, F)	$F \to (E)$
(F, I)	$F \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
(I, I)	$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

we have the CFG G_1 with productions

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

Figure 3.35: Example of CFG without unary productions

3.3.4 CFG in Chomsky Normal Form (CNF)

We can now summarize the various simplifications described so far. We want to convert any CFG that has not useless symbols, ϵ-productions or unary productions.
There is an important order for applying in the correct way the transformation:

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

After this, is so important to talk about **Chomsky normal form** or CNF for short. It's a type of grammar that has productions in the following forms:

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

so the parse tree of a CNF is a binary tree.
Moreover, this grammar hasn't useless symbols.
In order to transform a CFG in a CNF we first need to eliminate in the order specified before ϵ-productions, unary productions, useless symbols.
The resulting grammar has productions of the form:

- $A \hookrightarrow a$
- $A \hookrightarrow \alpha$ where $\alpha \in (V \cup T)^*$ of length ≥ 2

To transform the previous CFG to a CNF we have to:

- arrange all bodies (right-hand side) of $|\alpha| \geq 2$ making them composed only of terminal variables, or rather, if there are productions that has in the right side hand a string α of length ≥ 2 we have to made it composed only by non-terminal variables
- break bodies of length $|\alpha| \geq 3$ into a cascade of productions, each with a body consisting of two variables

The first transformation is made by creating a new variable for every terminal $\in \Sigma$ that appears in a body of length ≥ 2, called E_i. Add a new production $E_i \hookrightarrow a$ and use it for each occurrence of a in α.
The second transformation is made by transforming each productions of this type $A \hookrightarrow B_1 B_2 ... B_k$ with $k \geq 3$ by introducing $k-2$ new variables C_1, C_2, ..., C_{k-2} and replace the production with a chain of the new productions as shown in Figure 3.36.

$$
\begin{aligned}
A &\rightarrow B_1 C_1 \\
C_1 &\rightarrow B_2 C_2 \\
&\vdots \\
C_{k-3} &\rightarrow B_{k-2} C_{k-2} \\
C_{k-2} &\rightarrow B_{k-1} B_k
\end{aligned}
$$

Figure 3.36: Breaking of the productions of length ≥ 3

3.4 Pumping Lemma and Properties of CFLs

3.4.1 Pumping Lemma for CFL

As for regular languages is a tool for showing that certain languages are not CFL.
The pumping lemma for context-free says, in an informal way: in each sufficiently long string of a CFL we can find two substrings "next to each other" that can be eliminated or can be iterated and still resulting in strings of the language.
The first step in derivation of pumping lemma is to examine the shape and size of parse trees.
To do that we can use CNF to turn parse trees into binary trees. These trees have some convenient property, one of this exploit below.
Theorem: suppose to have a CFG $G = (V, T, P, S)$ in Chomsky Normal Form and suppose that the yield of a parse tree T is a terminal string $w \in L(G)$. If the longest path in T has n arcs, then $|w| \leq 2^{n-1}$. The **proof** is a simple induction on n:
Base: previously we recall that the lenght of a path is the number of edges from the root to the node or the number of nodes minus one. If T has $n = 1$ there is one leaf labeled by a terminal and the root. String w is the terminal symbol of the leaf so $|w| = 1$. We have $|w| \leq 2^{n-1} = 2^0$ and it is verified.
Induction: for $n > 1$, being in a Chomsky normal form, and since $n > 1$, root usees a productions $S \hookrightarrow AB$. We can use factorization property to decompose $w = uv$ and from the moment that $S \Rightarrow AB \Rightarrow^* w = uv$ we can write $A \Rightarrow^* u$ and $B \Rightarrow^* v$. The paths under the subtree rooted at A and B can have lenght $\geq n - 1$ and by the inductive hypothesis $|u| \leq 2^{n-2}$ and $|v| \leq 2^{n-2}$.
So at the end we can conclude $|w| = |uv| \leq 2^{n-2} + 2^{n-2} = 2^{n-1}$
Now we can give a formal definition of pumping lemma.
Theorem: Pumping Lemma for CFL: Let L be a CFL. There exists a constant n such that if $z \in L$ and $|z| \geq n$ we can factorize it as $z = uvwxy$ under the following conditions:

- $|vwx| \leq n$
- $vx \neq \epsilon$
- $uv^iwx^iy \in L$ is still in the language for each $i \geq 0$

So, if we read this statement in reverse, to prove that a language is not CFL we need to find a string z that for all possible factorization don't hold pumping

lemma for a certain i.

Proof: Let G be some CFG in CNF what generates for $L(G)$. Being a CNF, in L there is not ϵ but it's not important because we pick a string w that has lenght > 0 so z cannot be ϵ.

So we start with a CNF grammar $G = (V, T, P, S)$ such that $L(G) = L - \{\epsilon\}$ and let G have m variables $\in V$.

We choose a string $z \in L$ such that $|z| \geq n$ and we choose $n = 2^m$.

However, for the last theorem, a parse tree with the longest path of length $\leq m$ must have a string of yield of length $\leq 2^{m-1} = n/2$.

It's no possible for z so we conclude that for $|z| \geq n$ the longest path has length $> m$.

To be a yield for some parse tree, the longest path has to be of length $\geq m + 1$.

Let $k + 1$ be the length of longest path for z, with $k \geq m$ (otherwise z can not be in the yield) so we have $k + 1 > m$.

So if we have a path of lenght $k + 1$ we have $k + 1$ variables and one terminal symbol but, since $|V| = m$ and $k + 1 > m$ there must exist two i, j such that $A_i = A_j$.

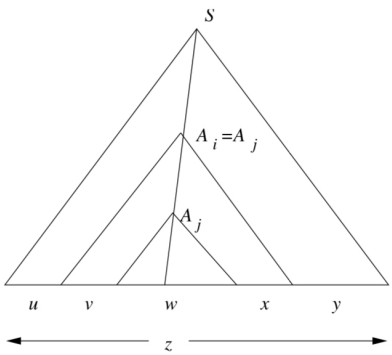


Figure 3.37: Representation of pumping lemma for CFL

As we can see in Figure 3.37 we can subdivide the tree in three part: string w is the yield of the subtree rooted by A_j. Strings v and x are the strings rooted by A_i on the left and on the right of w. Note that, since there are not unit production, v and x can not be both ϵ.

To explain the condition $vx \neq \varepsilon$ in the pumping lemma, focus on the subderivation $A \Rightarrow^* vAx$. We know that this derivation is associated with distinct nodes in the tree, so there must be at least one step involved in the derivation.

Consider for instance a derivation of the form $A \Rightarrow BA \Rightarrow^* vA$, where $B \Rightarrow^* v$. Here we have $x = \varepsilon$ and we know that $v \neq \varepsilon$, since a Chomsky normal form

grammar cannot derive the empty string.
More in general, one can have a derivation of the form $A \Rightarrow B_1C_1 \Rightarrow B_1B_2C_2 \Rightarrow \cdots \Rightarrow B_1B_2 \cdots B_{k-1}C_{k-1} \Rightarrow B_1B_2 \cdots B_kA \Rightarrow^* vA$, again with $x = \varepsilon$ and $v \neq \varepsilon$.
An entirely symmetrical argument could be made for derivations of the form $A \Rightarrow^* Ax$ with $v = \varepsilon$ and $x \neq \varepsilon$.
If $A_i = A_j = A$, then we can construct new parse trees from the original tree, as we can see in Figure 3.38(a).

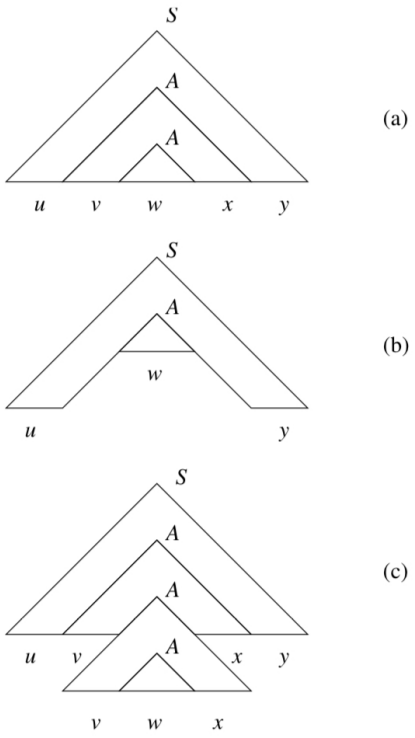


Figure 3.38: Diagram representation of a PDA

Replacing the subtree rooted at A_i by subtree rooted at A_j, we obtain the case $i = 0$ and the string uwy as shown in Figure 3.38(b). In the same way, replacing the subtree rooted at A_j with the subtree rooted at A_i, we obtain the case $i \geq 1$ and the string uv^iwx^iy as shown in Figure 3.38(c).
Existing parse trees for strings of the form uv^iwx^iy we have proved the third condition of the lemma.
The remaining condition is the first one which says that $|vwx| \leq n$. However, we picked A_i to be close to the bottom of the tree such that $k - i \leq m$. Thus, the longest path in the subtree rooted at A_i is $\leq m$. By the theorem seen before, the subtree rooted at A_i has a yield whose length is no greater than $2^m = n$.

3.4.2 Consequences of the pumping lemma

There are some important consequences of the pumping lemma:

- A CFL cannot generate **crossing pairs**.
  For example a language $L = \{0^i1^j2^i3^j \mid i, j \geq 1\}$ want the first and the third blocks equals but it's not possible. If we choose, given n, $0^n1^n2^n3^n$, vwx covers occurrences of at most two alphabet symbols, so, in all possible factorizations, the strings generated by removing it don't belong to L.
  We can deal about this concepts by considering PDA: we push i times 0, after we push j times 1, from the moment that $i \neq j$, we pop j times 1 that is not equal to i.

- A CFL cannot generate **string copies** for example $L = \{ww \mid w \in \{0,1\}^*\}$. Choosing $w = 10^n110^n1$, all the possible factorizations can't satisfy the pumping lemma

Another important property is the property of **substitution**: assume two finite alphabets Σ, Δ and a function of this type:

$$s : \Sigma \hookrightarrow 2^{\Delta^*}$$

replacing a symbol with a language composed by a subset of Δ^*. Let $w \in \Sigma^*$, with $w = a_1a_2...a_n$, $a_i \in \Sigma$ and we define:

$$s(w) = s(a_1)s(a_2)...s(a_n)$$

where each $s(a_i)$ is a language. For $s(w)$ we concatenate all the languages. For $L \subseteq \Sigma^*$, we define:

$$s(L) = \bigcup_{w \in L} s(w)$$

Function s is called **substitution**.
In Figure 3.39 there is an example of the substitution concept.

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

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

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

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

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

Figure 3.39: Example of substitution function

Theorem: Let L be a CFL defined over Σ and let s be a substitution defined on Σ such that, for each $a \in \Sigma$, $s(a)$ is a CFL. Then $s(L)$ is a CFL.
Proof: Let $G = (V, T, P, S)$ be a CFG generating L and, for each $a \in \Sigma$, let $G_a = (V_a, T_a, P_a, S_a)$ be a CFG generating $S(a)$.
We construct a CFG $G' = (V', T', P', S)$ with the same starting symbol of the starting CFG.

$$V' = (\bigcup_{a \in \Sigma} V_a) \cup V$$

$$T' = \bigcup_{a \in \Sigma} T_a$$

$$P' = (\bigcup_{a \in \Sigma} P_a) \cup P_R$$

P_R is obtained by replacing from the productions in P each occurence of a in the right side hand $S \Rightarrow^* a$ with the starting symbol of the CFG S_a.
We obtain the construction that we can see in Figure 3.40
Starting from this intuition we first prove $L(G') = s(L)$.
We first prove $\supseteq$: let $w \in s(L)$ so there exists a string $x \in L$ such that

$$x = a_1 a_2 ... a_n$$

and furthermore there exist strings $x_i \in s(a_i)$ such that $w = x_1 x_2 ... x_n$.
As we can see from the previous parse tree for G', we can generate $S_{a_1} S_{a_2} ... S_{a_n}$ in G' and then generate $x_1 x_2 ... x_n = w$. Therefore $w \in L(G')$.
Now we have to prove $\subseteq$. Let $w \in L(G')$.
Then the parse tree for w must have the form of Figure 3.40. We can remove the

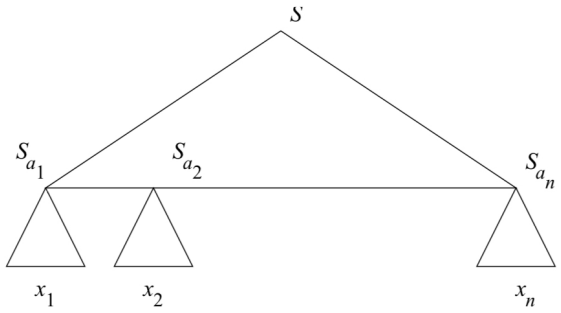


Figure 3.40: Parse tree associated to G′

subtrees at the bottom and get a parse tree with yield $S_{a_1} S_{a_2} ... S_{a_n}$ corresponding to a string $a_1 a_2 ... a_n \in L(G)$. We must also have $w \in s(a_1 a_2 ... a_n)$ and thus $w \in s(L)$.

3.4.3 Closure properties under operators

As for regular languages, also for context free languages there are closure property under some operators.
We have introduced the concept of substitution to easily prove these theorem.
Theorem: The CFLs are closed under the following operations:

- union
- concatenation
- Kleene closure and positive closure
- Homomorphism

Proof: For proving them we define a specific substitution and we apply the previous theorem.
Union: Given two CFLs L_1 and L_2, consider the CFL $L = \{1, 2\}$ and define a substitution function $s(1) = L_1$ and $s(2) = L_2$ we have that $L_1 \cup L_2 = s(L)$, which still is a CFL
Concatenation: Given two CFLs L_1 and L_2, consider the CFL $L = \{1 \cdot 2\}$ and define a substitution function $s(1) = L_1$ and $s(2) = L_2$ we have that $L_1 \cdot L_2 = s(L)$, which still is a CFL
Kleene and + closure: Given a CFL L_1, consider the CFL $L = \{1\}^*$ and define a substitution function $s(1) = L_1$ we have that $L_1^* = s(L)$, which still is a CFL. A similar argument holds for +
Homomorphism: Given a CFL L_1, and a homomorphism h, both over Σ. We define a substitution function $s(a) = \{h(a)\}$ for each $a \in \Sigma$. We then have $h(L) = s(L)$, which still is a CFL.
Theorem: If L is a CFL, then so is L^R.
Proof: Assume L is generated by a CFG $G = (V, T, P, S)$ and we build $G^R = (V, T, P^R, S)$ where

$$P^R = \{A \hookrightarrow \alpha^R \mid (A \hookrightarrow \alpha) \in P\}$$

By using induction on derivation length in G and G^R we can show that $(L(G))^R = L(G^R)$ (omitted).
We can show with an example that intersection between two context free languages is not always context free, so it means that it is not close under intersection, as we can see in the example in Figure 3.41.
Instead, the intersection between CFL and regular language is close under intersection.

$L_1 = \{0^n1^n2^i \mid n \geqslant 1,\ i \geqslant 1\}$ is a CFL, generated by the CFG

$$S \rightarrow AB$$
$$A \rightarrow 0A1 \mid 01$$
$$B \rightarrow 2B \mid 2$$

$L_2 = \{0^i1^n2^n \mid n \geqslant 1,\ i \geqslant 1\}$ is a CFL, generated by the CFG

$$S \rightarrow AB$$
$$A \rightarrow 0A \mid 0$$
$$B \rightarrow 1B2 \mid 12$$

$L_1 \cap L_2 = \{0^n1^n2^n \mid n \geqslant 1\}$ which is not a CFL

Figure 3.41: Intersection of two CFL that is not CFL

Theorem: Let L be some CFL and let R be some regular language. Then $L \cap R$ is a CFL.

Proof: We prove it using a PDA for the CFL that recognizes L, more precisely a PDA $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$ with acceptance for final state, and a DFA $A = (Q_A, \Sigma, \delta_A, q_A, F_A)$ that recognizes R.

We construct PDA for $L \cap R$ based on the idea represented in Figure 1.xx.

We construct a PDA for $L \cap R$ based on the following idea

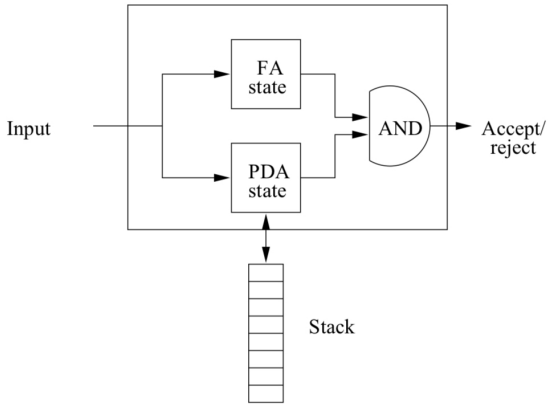


Figure 3.42: Construction of the idea for $L \cap R$

We define a PDA

$$P' = (Q_P\ x\ Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P\ x\ F_A)$$

where as input of the transition function δ there is a couple of state. For a given

symbol a we have:

$$\delta((p,q),a,X) = \{((r,s),\gamma) \mid (r,\gamma) \in \delta_P(p,a,X),\ s = \hat{\delta}\}(q,a)$$

It's possible to show by induction on the number of steps that

$$(q_P, w, Z_0) \vdash_P^* (q, \epsilon, \gamma)$$

if and only if

$$(q_P\ x\ q_A, w, Z_0) \vdash_{P'}^* ((q,p),\epsilon,\gamma),\ p = \hat{\delta}(q_A, w)$$

(q,p) is an accepting state for P' if and only if q is an accepting state for P and p is an accepting state for A (from the moment that there is the AND logic symbol). So P' accepts w if and only if P accepts w and A accepts w, that is $w \in L \cap R$.
Theorem: Let L, L_1, L_2 be CFLs and let R be a regular language. Then:

- $L - R$ is a CFL
- $\overline{L}$ may fall outside of CFL
- $L_1 - L_2$ may fall outside of CFLs

Proof: Operator - with REG -> If we have $\overline{R}$ it is also regular for the previous theorem $L \cap \overline{R}$ is a CFL so $L \cap \overline{R} = L - R$ is a CFL
Complement operator -> If $\overline{L}$ would always be a CFL, then we have that $L_1 \cap L_2 = \overline{(\overline{L_1} \cap \overline{L_2})}$ would always be a CFL that is a contradiction for previous theorem regarding intersection between CFL
Operator - with CFL -> Σ^* is a CFL and if $L_1 - L_2$ would always be a CFL, then $\Sigma^* - L = \overline{L}$ would be always a CFL that is a contradiction for the last theorem.

3.4.4 Computational properties for CFLs

We investigate the **computational complexity** for some of the previous transformation.
We denote with n the length of the entire representation of a PDA or a CFG.
Can be computed in $O(n)$ time:

- from PDA accepting by final state to PDA accepting by empty stack and vice versa

- from PDA to CFG

From PDA to CFG $O(n^3)$.
When we deal with the conversion to Chomsky normal form of a CFG we compute in time $0(n)$:

- the set of reachable symbols $r(G)$
- the set of generating symbols $g(G)$
- the elimination of useless symbols from a CFG

We compute in time $O(n)$ the set of nullable symbols $n(G)$ and the elimination of ϵ-productions, using a preliminary binarization.
We can compute in time $O(n^2)$ the set of unary symbols $u(G)$ and the elimination of unary productions from a CFG.
Given a CFG of size n, we can construct an equivalent CFG in CNF in time and space $O(n^2)$.

3.4.5 Decision problems

Emptiness test: Let G be some CFG with start symbol S. $L(G)$ is empty if and only if S is not generating. We can test the emptiness for $L(G)$ using the algorithm already mentioned for the computation of $g(G)$ and if there is the variable S, then the language is empty.

CFL membership: Given as input a string w, we want to decide whether $w \in L(G)$ where G is some **fixed** CFG.
Assume G in CNF and $|w| = n$. Since the parse trees for w are binary, the number of internal nodes for each tree, derived by a string of length n, is $2n - 1$ and we can generate all the parse trees of G with $2n - 1$ internal nodes and test whether some tree yields w.
We can use **dynamic programming** techniques that is is a computer programming technique where an algorithmic problem is first broken down into subproblems, solved, and then the two solutions are merged into a unique one.
The following algorithm is called **CKY algorithm** and it's used to find in an eficienty way if a string $w \in L$. Let be $w = a_1a_2...a_n$ and we construct a triangular **parse table** where the cell X_{ij} is a set that contains all the non-terminal variables such that $A \Rightarrow^* a_ia_{i+1}...a_j$.
The table is shown in Figure 3.43.

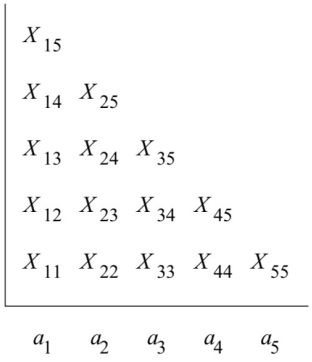


Figure 3.43: Generic table for language membership of a string

We construct the table considering one row at a time and from bottom to top.
Notice that each row correspond to one length of substrings; the bottom row is for strings of length 1, the last row is for substrings of length n.
If we focus on a generic X_{ij}, for example X_{24}, we consider a substring of length 2 constructed by goes down vertically and diagonally, reaching the bottom row, and in this case the substring is $a_2a_3a_4$.
Remembering that we are in Chomsky Normal Form, we give in Figure 3.44 the rule for the construction:

Base $X_{ii} \leftarrow \{A \mid (A \to a_i) \in P\}$

Induction We build X_{ij} for increasing values of $j - i \geqslant 1$

$X_{ij} \leftarrow X_{ij} \cup \{A\}$ if and only if there exist k, B, C such that

- $i \leqslant k < j$
- $(A \to BC) \in P$
- $B \in X_{ik}$ and $C \in X_{k+1,j}$

Figure 3.44: Rules for the population of the table

We populate the table row by row, increasing the value at each iteration.
If we break the string, we have two substring of smaller length: we have already information about string of these length from the moment that we work row by row and each row computes substring of certain lengths.
We want to find all the possible breaks such that the first substring is generated by $B \Rightarrow X_{i,k}$ and $C \Rightarrow X_{k+1,j}$ with $A \hookrightarrow BC$ added to X_{ij}.
To populate X_{ij} we need to check at most n pairs of previously built cells of the parse table. The pattern, in which we go up to the column below X_{ij} at the same tame we go down the diagonal is represented in Figure 3.45.

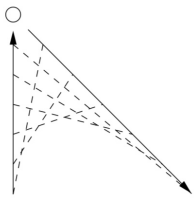


Figure 3.45: Pattern that we have to follow to populate the table

We can see an example in Figure 3.46.

Let G be a CFG with productions

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

and let $w = baaba$

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

Figure 3.46: Example

To compute this table we start by inserting in the bottom row all the productions of this type $A \hookrightarrow a_1$ with A, a_1 that are a generic non-terminal variable and a a generic terminal symbol.
After that, we have to compute the second row: for X_{12} we take the substring that goes down vertically, reaching b, and goes down diagonally, reaching a, so it's ba. The substring ba can be subdivided only in the middle and the combinations could be: BA, BC. Looking in the set of productions, we can se that these productions are generated by S and A and these are added to the set X_{12}. We compute this row in the same way.
It's more difficult when the substring is of length > 2: in this case, for example taking a substring of length 3, referring to row 3, there are different way to break in two parts this string.
For example, taking X_{24}, we have the substring obtained by going down vertically, reaching a, and go down diagonally, reaching b, obtaining the string aab. This can be break as aa and b or a and ab.
For the first case, we already know which non-terminal generates aa: in fact it's the set X_{23}. We have to do all the combination between X_{23}, in this case B, and X_{33}, in this case B. The possible combination is only BB that can't be generated

by any non-terminal.
For the second case, we already know which non-terminal generates ab: in fact it's the set X_{34}. We have to do all the combination between X_{34}, in this case $\{S, C\}$, and X_{22}, in this case $\{A, C\}$. The possible combination are $\{SA, SC, CA, CC\}$ and only CC can be generated by B, adding it to the set X_{24}.

4

REC and RE Languages

4.1 Turing Machine

A Turing machine is a finite state automaton with the addiction of an infinite **memory tape**, divided into squares or cells, that contains symbols from an alphabet, with:

- sequential access
- unlimited capacity in both tape directions

Different respect the PDA model, the input is initially placed into the auxiliary memory.
The Turing machines are used to prove the property of some languages as **undecidability** and **intractability**.
In Figure 4.1 is shown the construction of a Turing machine:

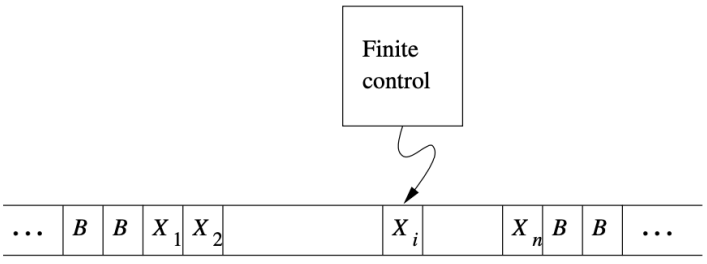


Figure 4.1: Construction idea of a TM

Initially, the finite input is placed on the tape and all the other tape cells initially

hold a special symbol called **blank**, that is a tape symbol but not an alphabet symbol.
There is the **tape head** that is always positioned at one of the tape cells and the Turing machine is said to be **scanning** that cell.
At the start, the tape head is at the leftmost cell that holds the input.

Informally, a Turing machine performs a move according to its state and with the symbol which is read by the tape head. In one move, the Turing machine will:

- Change state, the next state may be the same as the current state
- Write a new tape symbol in the cell read by the tape head (can be the same)
- Moves the tape head to the cell to the right or to the left

We can give a formal notion for a Turing Machine (TM), that's a 7-uple:

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

where:

- Q: finite set of states
- Σ: finite set of input symbols
- Γ: complete set of **tape symbols**; $\Sigma \subset \Gamma$
- δ: is the usual transition function: we give as input the couple (q, X) and the output is a triple (p, Y, D) where p is the next state, Y is the tape symbol that replace the symbol in the scanned cell and D is the direction that could be L, R to indicate left or right
- q_0: starting state
- B: blank symbol that is in Γ but not in Σ i.e. it is not an input symbol
- F: the set of final states

When TM performs a move it changes its configuration or in other words it moves from an instantaneous description to another. An instantaneous description is a string of the form:

$$X_1 X_2 ... X_{i-1} q X_i X_{i+1} ... X_n$$

ignoring the blank cells of the tape (being them infinite).
q is M's state, on the left of the symbol that TM is reading, $X_1 ... X_n$ is the portion

of M's tape between the leftmost and the rightmost symbol and X_i is the tape symbol scanned.
To represent the **computation step** of M we use the binary relation $\vdash$ defined on the set of IDs.
In Figure 4.2 is shown the computation step:

$$\text{If } \delta(q, X_i) = (p, Y, L), \text{ then}$$

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \underset{M}{\vdash} X_1 X_2 \cdots p X_{i-1} Y X_{i+1} \cdots X_n$$

$$\text{If } \delta(q, X_i) = (p, Y, R), \text{ then}$$

$$X_1 X_2 \cdots X_{i-1} q X_i X_{i+1} \cdots X_n \underset{M}{\vdash} X_1 X_2 \cdots X_{i-1} Y p X_{i+1} \cdots X_n$$

Figure 4.2: Computation step of a TM

The first case is when we have to go the the left side of the tape, the second case when we have to go to the right.
There are four special cases of computation. Suppose $\delta(q, X_i) = (p, Y, L)$ and consider this two special case:

- If $i = 1$, then M moves to the blank to the left of X_1. In that case we have
$$qX_1X_2...X_n \vdash pBYX_2...X_n$$
- If $i = n$ and $Y = B$ then the symbol B written over X_n joins the infinite tail of blanks symbol so doens't appear in the next ID
$$X_1X_2...X_{n-1}qX_n \vdash X_1X_2...X_{n-2}pX_n$$

The remaining special cases happen when $\delta(q, X_i) = (p, Y, R)$:

- If $i = n$, then M moves to the blank to the right of X_n. In that case we have
$$X_1X_2...X_{n-1}qX_n \vdash X_1X_2...X_{n-1}YpB$$
- If $i = 1$ and $Y = B$ then the symbol B written over X_1 joins the infinite tail of blanks symbol so doens't appear in the next ID
$$qX_1X_2...X_n \vdash pX_2...X_n$$

As usual, $\vdash^*$ will be used to indicate **zero, one or more moves** of the TM. An accepting computation has the following form:

$$q_0 w \vdash^* \alpha p \beta$$

where α, β are strings without condition and we say that the computation is accepted if we start from an initial ID $q_0 w$ and M reaches a final state p without

worrying about the rest of the tape.
Example: Let us specify a TM M with $L(M) = \{0^n1^n \mid n \geq 1\}$ We have an example input string: taking as input 0011, that is in the language, this string must be accepted by the TM. The computation steps are shown in Figure 4.3 below:

$$\begin{aligned} q_0 0011 &\vdash Xq_1 011 \vdash X0q_1 11 \\ &\vdash Xq_2 0Y1 \vdash q_2 X0Y1 \\ &\vdash Xq_0 0Y1 \vdash XXq_1 Y1 \\ &\vdash XXYq_1 1 \vdash XXq_2 YY \\ &\vdash Xq_2 XYY \vdash XXq_0 YY \\ &\vdash XXYq_3 Y \vdash XXYYq_3 B \\ &\vdash XXYYBq_4 B \end{aligned}$$

Figure 4.3: Computation step of 0011 for TM M

When we read a 0 and we are in q_0, we move to q_1 and we put an X for replace the scanned symbol. If we read a 0, we go on, remaning in the same state q_1 until we reach a 1: we replace the scanned symbol with an X and we go back to the previous tape symbol, moving to a state q_2. We do this until we reach the head of the tape, and if we read a symbol already scanned we go on moving to q_0 and the loop restart.
If input has the form 0^*1^* then at each ID the tape is of the form $X^*0^*Y^*1^*$ with M that implements the following strategy:

- in q_0 it replaces the leftmost 0 with X and moves to q_1
- in q_1 it proceeds from left to right, goes over 0 and Y looking for a 1. When it finds a 1, replaces it with an Y and moves to q_2
- in q_2 it proceeds from right to left, goes over Y and 0 looking for the right most X, moving back to q_0
- in q_0 if it finds one or more 0 it resumes the above cycle, otherwise it moves to q_3
- in q_3 it overrides all of the Y's and accepts if there is no 1

As we can see, input string is overwritten during the computation.

We can explain the TM by using table representation, as shown in Figure 4.4.

	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)
$\star q_4$					

Figure 4.4: table representation of a TM

We can also represent δ by using the following transition diagram, shown in Figure 4.5.

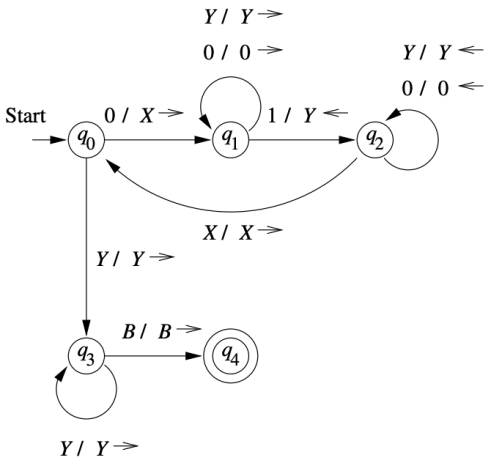


Figure 4.5: Diagram representation of a TM

In the diagram representation we are in one state, we read an input, that is the first symbol of $x|x$ put over each arc.
We go to another state, indicated by the arrow, replacing last symbol with second symbol of $x|x$ moving on the direction of the smallest arrow.
If we give as input one string that is not in the language as 0010 we compute until $XXY0q_1B$: here there is no computation from q_1 to tape symbol B so M dies and does not accept its input.
We have defined a TM as recognition device but, alternatively, we can use these devices to compute **functions** on natural number $\mathbb{N}_+^k \hookrightarrow \mathbb{N}_+$.

We can give an example of this type of TM. In this example the TM computes the function $max(m-n,0)$ so if the difference is negative, holds 0. The input is in this form 0^m10^n encoding each natural number in **unary notation**: for example (3,5) can be represented as 000100000 using 1 as separator and the output is a single number represented with the value of the function given by a sequence of 0.
In Figure 4.6 is shown the table representation for this TM.

	0	1	B
$\rightarrow q_0$	(q_1, B, R)	(q_5, B, R)	
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
$\star q_6$			

Figure 4.6: Table representation of this example

4.1.1 Language accepted by a TM

Being different types of TM, there are different types of language. A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ accepts the language:

$$L(M) = \{w \mid w \in \Sigma^*,\ q_0 w \vdash^* \alpha p \beta,\ p \in F,\ \alpha, \beta \in \Gamma^*\}$$

The class of languages accepted by this TMs is called **recursively enumerable** (RE).
As we have seen in the previous example, a TM can accept a string by **halting**. A TM halts if it enters in a state q with tape symbol X and the next move, or rather ,$\delta(q, X)$, is not defined.
We can always assume that a TM halts when a string is accepted and, without changing the language, we can make, for every final state $q \in F$, a $\delta(q, X)$ undefined.
Unfortunately, if a TM doesn't accept, we can't assume that it will halt in a non final state.
The class of languages accepted by some TM that halts for every input are called

recursive (REC), in which we have an input, and the TM halts always, even if the input is not accepted.
We can resume the differences between REC and RE:

- **Recursive** language (REC): the language is accepted by a TM that halts on each input string (in the language or not). Remembering the previous example, for each input we can define an output.
- **Recursive enumerable** language (RE): the language is accepted by a TM that halts when the strings belongs to the language. So, for the strings not in the language, the TM may compute forever

Resuming again we can say that the set of language accepted by Turing Machine can be subdivided in language which is made with TM that halts always (for strings that are in the language or not) and languages made by TM that halts only for strings in the language that may implicate, in case of string not in the language, the infinite computation.
So we can say that REC is a subset of RE ($REC \subseteq RE$) from the moment that in REC misses the class of languages for which there is a Turing machine that accepts the strings in the language but may run indefinitely or never halt on strings not in the language.
We can expand Figure 3.7 as we can see in Figure 4.7:

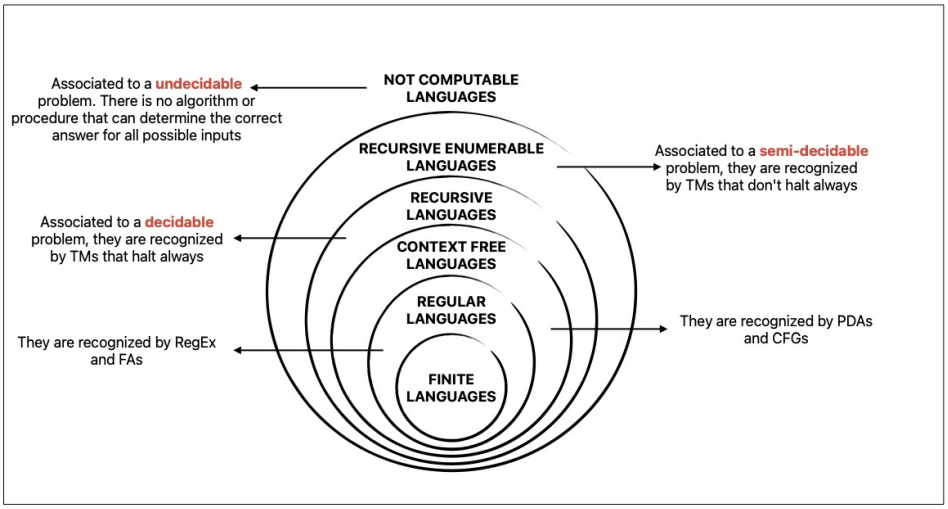


Figure 4.7: Set subdivision of languages

4.1.2 Decision problem

When we deal about problem we work with a mathematical object x, as a string, and with a property P, as the property of prime number, even number or others.

Decision problem associated to property P is, given an input x, we have to decide if $P(x)$ holds. For this reason we can associate a language L_P to the objects that satisfy this property.

The decision problem can be reformulated as the problem of finding if an object x belongs to L_P.

It turns out as we shall see that anything we more colloquially call a problem can be expressed as membership in a language. More precisely if Σ is an alphabet and L is a language over Σ then the problem L is: given a string $w \in \Sigma^*$, find if $w \in L$.

Not all decisional problems are as the one seen before: some problems require computing to give an output, as the example seen previously in which we have to compute a function.

Fortunately, we can associate to mathematical problem a decision problem, that is not more difficult than the original problem.

A decision problem is **decidable** if its encoding L_P is a recursive language or alternatively, if there is a TM M that always halts such that $L(M) = L_P$. If there isn't a TM M that always halts such that $L(M) = L_P$ the problem can be **semi-decidable**: a decision problem is **semi-decidable** if its encoding L_P is a recursive enumerable language or alternatively, if there is a TM M that halts certainly for string in the language and may compute forever for string not in the language such that $L(M) = L_P$.

If a decision problem L_P can't be recognized by one of this two types of TM, then it's **undecidable**.

4.1.3 Programming techniques for TM

Although the class of TM is very simple, this model has the same computational power of a modern computer from the moment that they can compute the same decision problems.
We're going to see that a TM can take take as input an algorithm, and gives as output another algorithm different. The algorithms of input and output are TMs.
This is the thing that computers do and is what we use to prove undecidable problems.
In the following part there are some examples of how we might think a TM and its tape in order to understand this computational power.
We reformulate the TM definition using:

- a finite number of registers with **random access**, placed inside each state, containing finite values. The state q is not only q but $[q, A, B, C]$
- a finite number of tape tracks i.e. there are more tapes

An idea of this construction is shown in Figure 4.8 below.

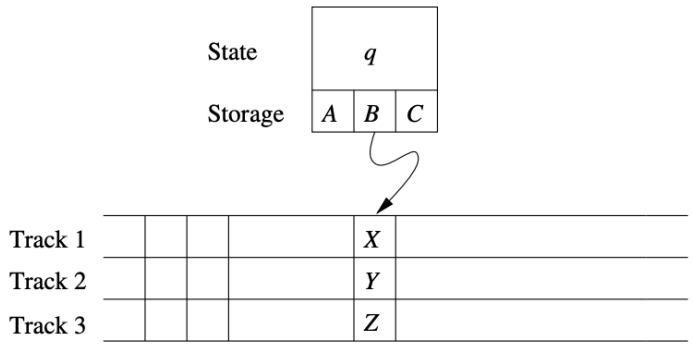


Figure 4.8: Multi-track TM

Example 1: A TM M that "memorizes" the first symbol read and verifies that this not appear again in the input

$$L(M) = L(01^* + 10^*)$$

If we do this with a simple TM we have to go up and down from the tape to find the symbol. Indeed, with this type of TM, we store our state, and in the register

$0,\ 1,\ B$.
We start in q_0 with a blank symbol and when we read as first symbol 0 or 1 we move to q_1 memorizing the symbol in the register.
If we see another symbol equal to the first symbol, there is no computation. Indeed, if the symbol is different, we remain in this state until we get a blank symbol, that means that the input is finished.
When we are in q_1, B, it means that the string are accepted.
In Figure 4.9 there are the definition of the TM.

Let $M = (Q, \{0,1\}, \{0,1,B\}, \delta, [q_0, B], B, \{[q_1, B]\})$, with
$Q = \{q_0, q_1\} \times \{0,1,B\}$

	0	1	B
$\rightarrow [q_0, B]$	$([q_1, 0], 0, R)$	$([q_1, 1], 1, R)$	
$[q_1, 0]$		$([q_1, 0], 1, R)$	$([q_1, B], B, R)$
$[q_1, 1]$	$([q_1, 1], 0, R)$		$([q_1, B], B, R)$
$\star[q_1, B]$			

Figure 4.9: TM of the example

For the second example we use the trick of composing the TM by many tracks. This is why is also called **multi-track**.
A multi-track Turing machine is a variant of the simple Turing machine. It consists of multiple tapes with a single head pointer. They are beneficial in solving complex problems and limit the amount of work, unlike a simple Turing machine.
As there is a single head, the direction of all the tapes changes together. The input is placed in the first tape and is transferred to the other tapes as per convenience.
Using this technique, we don't extend what the TM can do.
Example 2: A TM for the language $L = \{wcw \mid w \in \{0,1\}^*\}$. We use a tape track for "marking" those input symbols that we have already tested putting a $*$ and a register to remember the symbol until we see the first symbol of the second part.
Assuming a tape track $T2$ with a string as input $101c101$: if we see the first 1, we put in track $T1$ a * to represent that this symbol is already seen, we move after c and if the first symbol read is 1, we put an * and come back to first *. We move on and repeat this procedure. In Figure 4.10 there is the definition of the TM M.

$$M = (Q, \Sigma, \Gamma, \delta, [q_1, B], [B, B], \{[q_0, B]\})$$

where

- $Q = \{q_1, q_2, \ldots, q_9\} \times \{0, 1, B\}$
- $\Sigma = \{[B, 0], [B, 1], [B, c]\}$
- $\Gamma = \{B, *\} \times \{0, 1, c, B\}$

Figure 4.10: TM of the example

Use of a subroutine: We remember that there are TM with output. Giving a TM for the computation of the product function $0^m 10^n 1 \hookrightarrow 0^m n$. We use a subroutine "Copy" that we can see in Figure 4.11. When we reach q_1, we give to another Turing Machine with different task the input and this TM computes something.

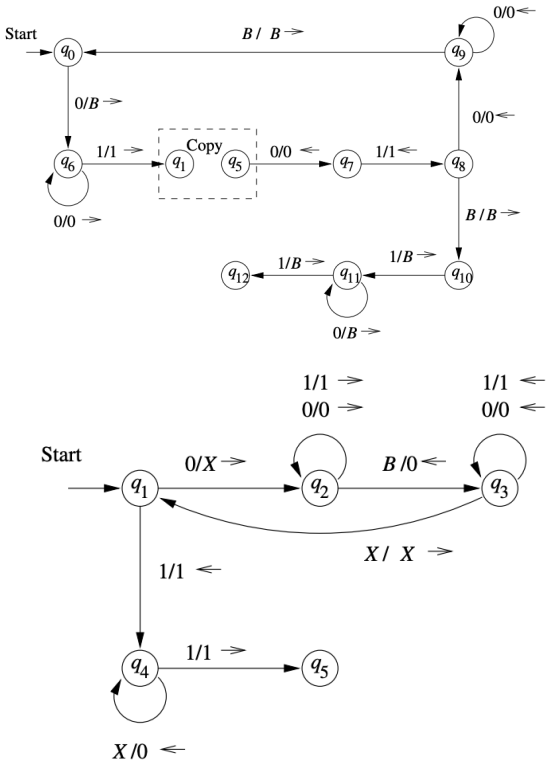


Figure 4.11: Use of subroutine for TMs

4.1.4 TM EXTENSIONS

It's important deal with the concept some **extensions** of the TM definition and after, for each extension, we prove that **computational capacity** is the same as the one off the classic definition of TM.
Multi-tape TM: We use a finite number of **independent** tapes for the computation, with the input on the first tape, as seen in Figure 4.12.

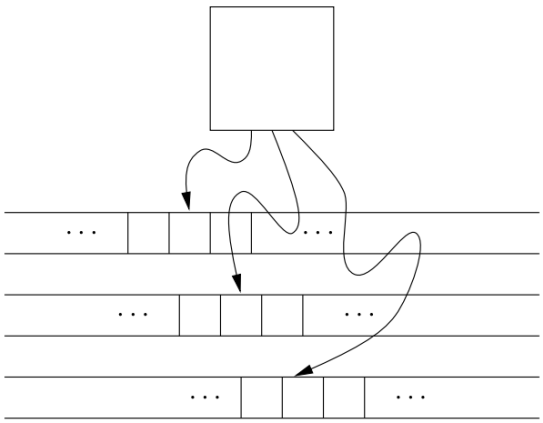


Figure 4.12: Multi-tape TM

It's different respect on multitrack from the moment that we can move indepently in multitape.
In a single move the multi-tape TM performs the following actions:

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

Theorem: A language accepted by a multi-tape TM M is RE
Proof We can simulate M using a TM N with a multi-track tape.
The main idea is to take a multi-track tape N to simulate a multi-tape TM M. To simulate a multi-tape of two tapes we need four tapes multitracks. Half tracks simulate the positions of the head and other half tracks have the content of the tapes of the multitape given as input.
N visits k heads (with k number of multitape tapes) to simulate a single move of M: after scanning k heads, from left to right pass, N knows

- what tape symbols are being scanned by each of M's heads

- the state of M stored in N's finite control

and thus N knows what move M will make.

N now comeback, changing the symbol in the track representing the corresponding tapes of M and move the head markers to the left or to the right, if necessary. Finally N changes the state of M and at this point N has simulated one move.

We select as N's accepting states all those states that record M's state as one of the accepting state of M. Thus, whenever M accepts, N also accepts, and N doesn't accept otherwise.

The idea of the construction is shown in Figure 4.13 below.

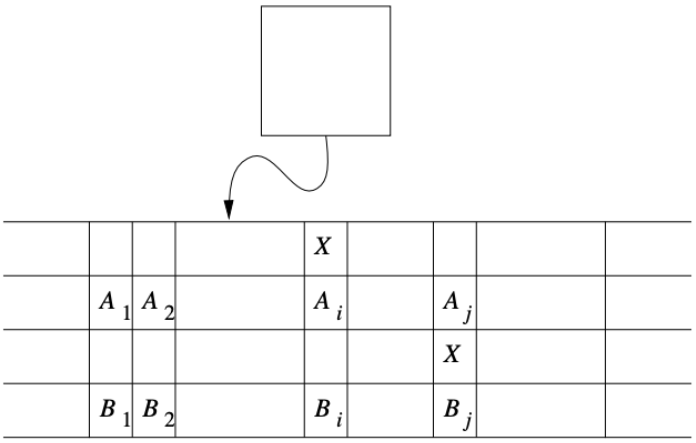


Figure 4.13: Diagram representation of a PDA

Theorem: The TM N in the proof of the previous theorem simulates the first n moves of the TM M with k tapes in time $O(n^2)$

Nondeterministic TM: In a **nondeterministic** Turing Machine, NTM for short, the transition function δ is set-valued:

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

At each step, the NTM chooses one of the triples as the next move.

The NTM 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.

Theorem: For each NTM M_N, there exists a deterministic TM M_D such that $L(M_N) = L(M_D)$

Proof: With a NTM we can construct a tree of all possible computations for an input w.

For the proof we specify M_D as a TM with two tapes, or rather a multitape, as shown in Figure 4.14

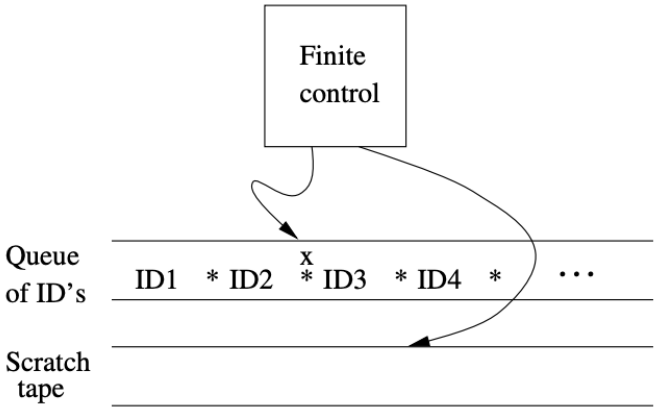


Figure 4.14: Multitape TM M_D

We want to visit the tree, for the input w, **breadth** first, seeing node at growing heights, until we see an ID with an accepting state, thus we stop.

The first tape is a **queue** of ID's, as seen in Figure 4.14, with the first ID put as first symbol of the tape.

The second tape is a scratch tape in which we copy ID1 and we compute all the other ID's that comes for the next move of δ and we copy those at the end of the queue.

We move the queue going in the next ID, iterating this process, until we see an accepting state.

Let m the maximum number of choices for M_N: after n moves M_N reaches a number of ID bounded by

$$1 + m + m^n + m^3 + ... + m^n \leq nm^n + 1$$

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

M_D reaches an accepting state of M_N in a finite amount of time, but it's so slow.

And so $L(M_D) = L(M_N)$

4.1.5 TM WITH RESTRICTIONS

We impose some restrictions on the definition of TM / multi-tape TM:

- tape is unlimited only in one direction
- two tapes used in stack mode

We prove that these models are equivalent to TM, and these models will be useful in some proofs.
Semi-infinite tape TM: In a TM with semi-infinite tape there are no cells to the left of the **initial tape head position** X_0 (that we don't have in normal TM) and a tape symbol can never be overwritten by the blank B.
There are no "holes" of blank symbols in the cells of the tape that we've already visited.
We have to show that a TM with semi-infinite tape has the same power of a normal TM and we do this by simulating a standard TM by using a TM with semi-infinite tape with two tracks, as seen in Figure 4.15

X_0	X_1	X_2	$\cdots$
*	X_{-1}	X_{-2}	$\cdots$

Figure 4.15: Semi-infinite tape TM

- the upper track contains the portion of TM that are at the right side of the initial position head X_0
- 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: when we read it, the TM know that it can't move to the left

If our tape head is in the upper portion, when we apply the δ function, we move to the right/left. Instead, if the cell is to the left of the initial position, the cell appears in the lower tape, and if the δ function move to the right/left we go to the reversal side.
* is a special symbol that prevent TM by blocking itself. When we read X_0 and we have to go the the left in the original TM, we move down and to the right in the multitape TM to simulate this move.

Theorem: Each language accepted by a TM M_2 is also accepted by a TM M_2 with semi-infinite tape.
The proof is the idea given before.

Multi-stack Machine:We apply to a multi-tape TM the restriction to use each tape in a stack mode:

- can only overwrite at the top
- can only insert at the top
- can only delete at the top

Let M be a multi-tape TM with tape used in stack mode. We also assume that:

- the input is provided in an **external**, read-only tape and with \$ end marker and it can be read from left to right.
- M can perform ϵ-moves, or rather we don't consume any input and still make one move but this can only happen if there are not conflict with other moves: if we are in a given tape head and if there are not moves to one direction, it is allowed to compute an ϵ-move

In Figure 4.16 there is a picture of how the machine looks like: as we can see each tape has the function of a stack.

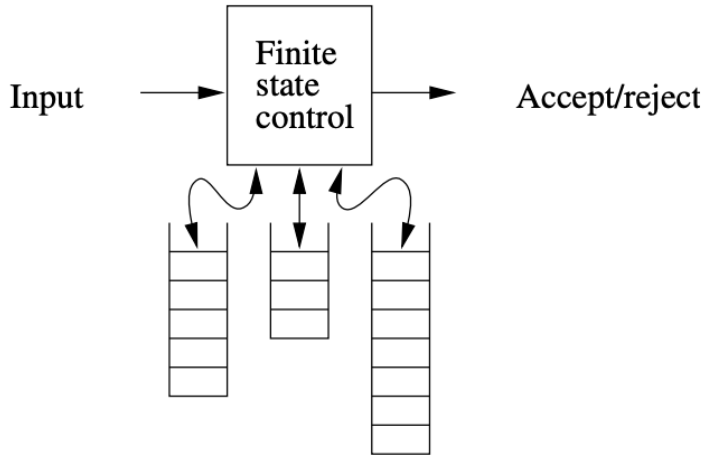


Figure 4.16: Multistack machine

Theorem: if a language L is accepted by a TM M, it is also accepted by a multistack machine with two stacks

Proof: this is an informal sketch idea of how we can simulate a TM by using a multistack machine.
The basic idea is to use only two stacks:

- simulate the tape to the left of the current position with the first stack
- simulate the tape starting from the current position and extending to the right with the second stack

Here there are the transition rules for S:

- Initially take the external input and copy it in the first stack
- Put the content of the first stack into the second stack
- If M performs a move, replacing X with Y and moving to the left, S pops Z from first stack and push ZX in the second stack
- If M performs a move, replacing X with Y and moving to the right, S pops X from second stack and push Y in the first stack

4.2 Undecidability

We've said that a modern computer has the same computational powerful of a Turing Machines. Moreover, we've said that a language L is **recursively enumerable** (RE) if there is a TM M that accepts this language $L = L(M)$. We can do a distinction between languages accepted by TMs that always halt and that halt for only strings in the language. Given as input a string w, M certainly halts if $w \in L(M)$, but M may not halt if $w \notin L(M)$. We say that these languages that are in RE-REC are called **semi-decidable** and, more generally, outside of REC, are called **undecidable**

A language L is **recursive** (REC), or equivalently, the decision problem L is **decidable**, if $L = L(M)$ for a TM M that **halts** for every input, with response that could be yes or not, so we don't have the uncertainty of the previous language.

The difference between with these two cases is the computing time: in RE we have to wait infinite time to know if a string w is in the language.

A recursive/decidable language corresponds to the definition of **algorithm**, for which we impose that computation halts both for positive and negative instances of the problem (and it's the definition of algorithm: a set of finite instructions that converges in a finite amount of time to an output).

String indexing: let us sort all strings in $\{0,1\}^*$ by length and lexicografically, for strings of the same length, as we can see in Figure 4.17, and we associate with each string a positive integer i called **index**.

i	string
1	ϵ
2	0
3	1
4	00
5	01
⋮	⋮

Figure 4.17: String indexing

We write w_i to denote the i_{th} string and we can easily verify that for each $w \in \{0,1\}^*$, we have that:

$$w = w_i \iff i = 1w$$

We now want to encode a TM with binary input alphabet $\{0,1\}$ of this type $M = (Q, \{0,1\}, \Gamma, \delta, q_1, B, F)$ that we denote with $enc(M)$, that is the binary string representing this machine.
We need to encode each element of the TM:

- we rename the states $q_1, ..., q_r$ with a convention that initial state is q_1 and the final state q_2 (unique)
- we rename the tape symbols $X_1, ..., X_s$ with the convention that $0 = X_1$, $1 = X_2$ and $B = X_3$
- directions $L = D_1$ and $R = D_2$

So each element has a number that characterises it.
The encoding that we use is the **unary encoding** i.e. the number of 0 is the value of the encoding. For example $enc(q_{17}) = 0^{17}$.
For the transiction function, if

$$\delta(q_i, X_j) = (q_k, X_f, D_m)$$

the encoding is:

$$0^i 1 0^j 1 0^k 1 0^f 1 0^m$$

Notice that we never have two consecutive occurrences of 1 since each index is never 0.
For a TM we concatenate the codes C_i (the five block seen before, all possible combinations of transitions) for all transitions, separated by 11

$$C_1 11 C_2 11 ... C_{n-1} 11 C_n$$

There are several encoding for M, obtained by indexing the symbols and/or listing the transitions in different orders.
We use $enc(M)$ to denote a generic M code. If we provide the encoding of a TM, we are able to recognizes the TM in a unique way.
There are strings that may not be valid as 001110 since there are three following zeros: these strings go to a Turing Machine called $M_\emptyset$, the TM that halts for every input.
We have a convention which will translate strings into Turing Machine and each string goes to a unique Turing Machine.
In Figure 4.18 there is an example of encoding.

Let $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$, where δ is defined as

$$\delta(q_1, 1) = (q_3, 0, R) \quad \delta(q_3, 0) = (q_1, 1, R)$$
$$\delta(q_3, 1) = (q_2, 0, R) \quad \delta(q_3, B) = (q_3, 1, L)$$

Transition encodings C_i

$$0100100010100 \quad 0001010100100$$
$$00010010010100 \quad 0001000100010010$$

TM encoding enc(M)

010010001010011000101010010011000100100101001100010001000100010

Figure 4.18: Example of encoding

Diagonalization Language: The diagonalization language is the set

$$L_d = \{w \mid w = w_i, w_i \notin L(M_i)\}$$

or, in other word, L_d consists of all encoded strings w_i such that the encoded TM M_i does not accept w_i as input.
In Figure 4.19 is shown when M_i accepts (1) or rejects (0) w_j.

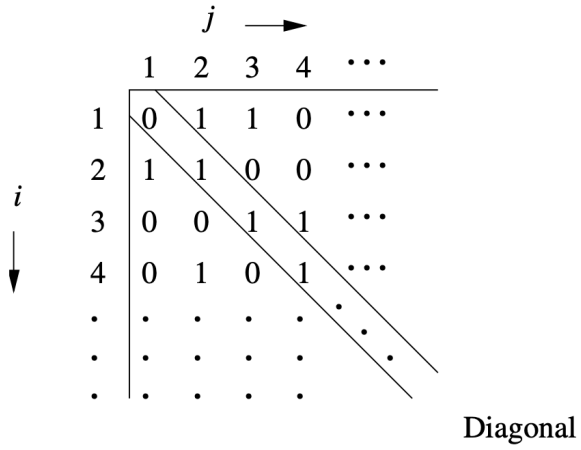


Figure 4.19: Table of accepting or rejecting

We may think of the $i-th$ row as the **characteristic vector** for the language $L(M_i)$ that indicates the strings accepted by this TM.
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.

The following statements are logically equivalent:

- the $i-th$ element of the diagonal is 0
- $w_i \notin L(M_i)$
- $w_i \in L_d$

If we complement the diagonal, we obtain the characteristic vector of L_d. This vector is not in none row of the table because the L_d vector, that is the complement of the diagonal, mismatch at least with the element of the diagonal of each row (from the moment that it's the complement of it).
It follows that L_d **is not in RE**.
A language L is **recursive** (REC) if $L = L(M)$ for some TM M such that if $w \in L$, then M halts in a final state, if $w \notin L$, then M halts in a non final state.
If we think of L as decision problem P_L, then we say that P_L is **decidable** whenever L is **recursive** and P_L is **undecidable** otherwise.
Decidability corresponds to the notion of **algorithm**: we have a sequence of steps that always ends and produces some answer. In Figure 4.20 there is a diagram representation of languages classes.

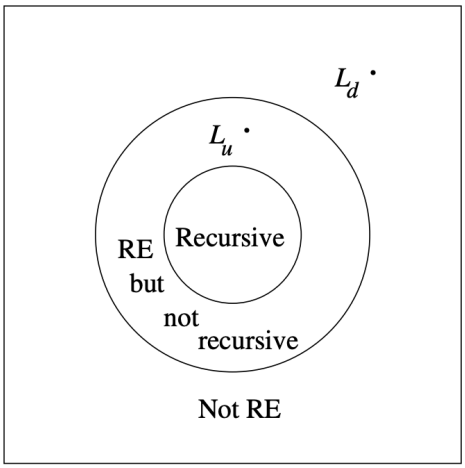


Figure 4.20: Belonging to set of languages

We can say that recursive is equal to decidable and M always halts, RE is when M halts upon acceptance and non-RE when we cannot compute, as in the case of L_d.

4.2.1 Properties of recursive languages

Theorem: If L is recursive, then $\overline{L}$ is recursive

Proof: If L is recursive, there is a TM M that always halts, such that $L = L(M)$. We construct a TM M' such that M' accepts when M does not and vice versa. M' always halts and $L(M') = \overline{L}$. This construction is shown in Figure 4.21

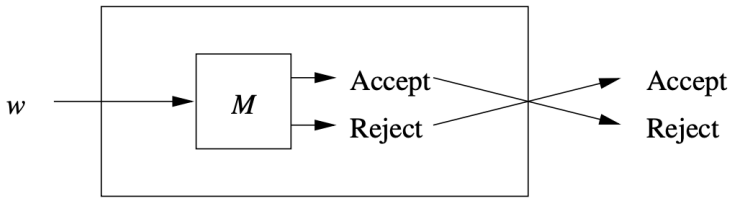


Figure 4.21: TM closed under complement

From this statement follows a **corollary**: if L is in RE and $\overline{L}$ is not in RE, then L cannot be a recursive language. Taking the Figure 4.20, we have that L is in the second larger circle, and we don't know if is also in the smallest one. Instead $\overline{L}$ is outside of the set of RE: the proof is that if L will be in REC, for the previous theorem, also $\overline{L}$ will be REC, but it's a contradiction.

Theorem: If L and $\overline{L}$ are in RE, then L is recursive

Proof. Let $L = L(M_1)$ and $\overline{L} = L(M_2)$, and these Turing Machine halts surely only in strings that are in the languages. We have to show that L is recursive by constructing of a TM M that always halts. We construct a multi-tape TM M that simulates M_1, M_2 in parallel, as shown in Figure 4.22

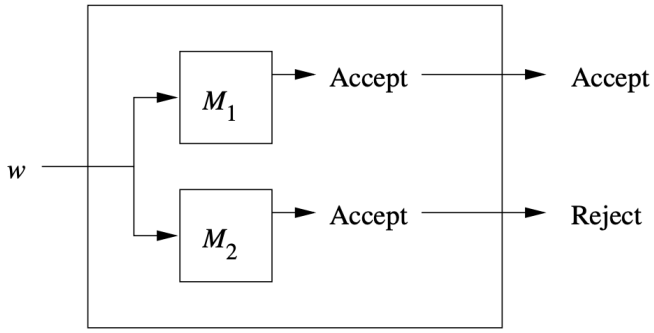


Figure 4.22: TM closed under complement

If the input is in L, M_1 accepts and halts, then also M accepts and halts. If the input is not in L, then TM M_2 accepts and halts, so M rejects and halts, implicat-

ing that the TM M always halts.
Given a language L and its complement $\overline{L}$, they can't be placed in every combination of the diagram of Figure 4.20, the theory allows only 4 combinations:

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

Universal language: We want to encode pairs (M, w) consisting of a TM M with binary input alphabet and one binary string w. We use $enc(M)$ followed by 111 and w to represent $enc(M, w)$.
The language L_u, called **universal language**, is the set

$$L_u = \{enc(M, w) \mid w \in L(M)\}$$

or, in words, the set of binary strings that giving a pair (M, w) as input, $w \in L(M)$.
This language is the first one that is in RE-REC. First we show that $L_u \in RE$ by showing that a Turing Machine M that doesn't halt always, generates the same language of L_u.
To show that there is a TM U, called **universal Turing machine**, such that $L_u = L(U)$.
The construction of this TM is shown in Figure 4.23.

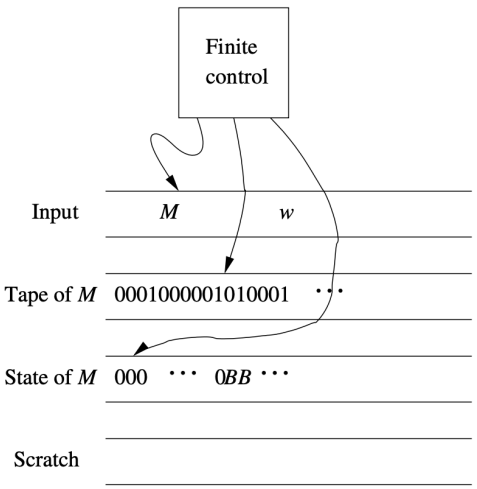


Figure 4.23: Universal TM U

It's a TM with many tapes, and it functions as an interpreter, taking as input a TM and a string and it executes the program on that string.
In the first tape is taken as input the couple (M, w), in the second tape we simulate M over string w and is the tape of M. Also the tape of M is encoded, representing each symbol X_i with a sequence of 0^i, separated by a 1. We use the encoded notion because M is written encoded and will be easier apply the rules that are encoded in the same way of the tape.
The third tape represent the state of M. Also this tape is encoded, with state q_i represented by a sequence of 0^i.
At the end there is the scratch tape, and we use it for doing marginal computation.
The strategy is to: in the middle of computation, after several steps, we have the tape of M encoded and the state of M encoded as in Figure 4.23. We look into the tape head (second arrow), supposing a symbol 0^p and into the state of M (third arrow), supposing a state 0^k. We collect this two symbols and we search 0^p10^k inside $enc(M, w)$ and this means a transition function of the type $\delta(q_p, X_k)$.
When we find it, we get the answer, encoded as $0^j10^s10^h$, that gives us the new state p_j, the symbol X_s of which we replace the tape symbol X_k, using the scratch tape if $s > k$, splitting the tape of M in two parts and putting one half in the scratch tape and inserting the new symbol, restoring all after, and the direction.
If there is no transition of the type 0^j... the TM M halts and U halts as well, instead, if M reach a final state, then U halts and accepts.
For this reason, existing a TM U that halts certainly for strings $\in L_u$, it's almost RE.
Now we have to show that is outside of REC. To do this we assume that L_u is recursive (REC) and, for the previous theorem, $\overline{L_u}$ is also recursive.
Let M be a TM that always halts such that $L(M) = L_u$ and we build a new TM M' for L_d showing that is in REC (that is not possible from the moment that we have proved that is not recursive).
The complement of L_u is that, if we give as input $enc(M, w)$, belongs to the language only the string that are not accepted by the machine M. We have a subroutine that says us if a string is not accepted by a machine, is accepted by $\overline{L_u}$.
As we can see, in Figure 4.24 there is the construction for the TM M' that accepts L_d and we have that if an $enc(w_i, M_i)$ is not in $L(M_i)$, the TM M for $\overline{L_u}$ accepts and also the TM M' for L_d, creating a TM for L_d, but is not possible from the

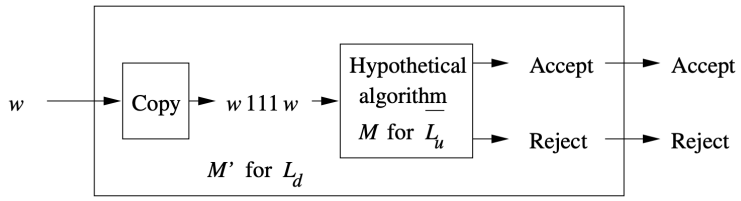


Figure 4.24: Ambiguous TM for L_d

moment that L_d is outside of RE.
Theorem: The language L_u is in RE - REC
Halting Problem: Given a TM M, we define $H(M)$ as the set of strings w such that M **halts** with input w.
Let us consider the language L_h, called the **halting problem**

$$L_h = \{enc(M, w) \mid w \in H(M)\}$$

L_h is a RE language.

4.2.2 Reduction

To prove that a language is REC we can find a machine that always halts, to prove that a language is RE we can find a machine that halts certainly only for strings in the language.
However, if we want to find that a language, associated to a decision problem P_2 is **not in a class**, we can pick up a problem P_1 for a language that is not in class X, find a reduction i.e mapping the instances of the first problem P_1 to the instances of the second problem P_2 that we want to find if is in class X and at this point we've shown that P_2 is more difficult than P_1, and if P_1 is not in class X, also P_2 doesn't belong to it.
For example, we know that L_d is not in RE: if we show that there is a map for all instances of L_d to the instances of a second language L_i, then also L_i is not in RE.
Given a problem P_1, a decision problem, an "yes or not" question for some strings, which define a language, known to be difficult, we want to know whether a second problem P_2, under investigation, is as hard, or even harder than P_1.
To this end we show that, if we could solve P_2, then we could also solve P_1, written as

$$P_1 \leq_m P_2$$

This technique is called **reduction** of P_1 to P_2.
A **reduction** from P_1 to P_2 is an algorithm (usually a TM) that converts an **instance** x (usually a string) of P_1 into an instance y of P_2, such that:

- if x has positive answer then y has positive answer (if x is in the language, also y is in the language)
- if x has negative answer then y has negative answer

If we are able to construct an algorithm that do this, we have proved that P_2 is more or equal difficult to P_1.
The idea is shown in Figure 4.25

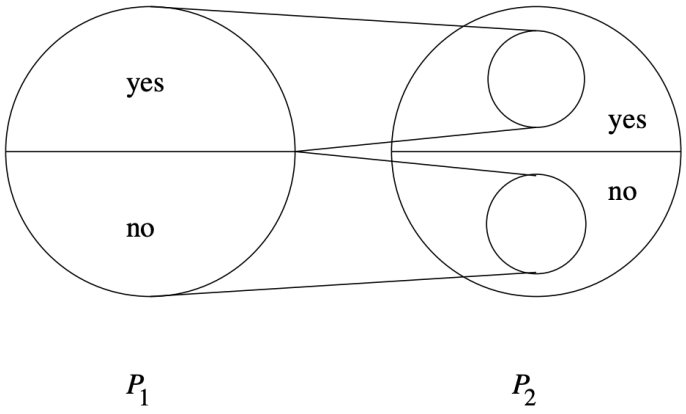


Figure 4.25: Reduction idea

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

Whatever the answer is, it's also valid for x in P_1.
We've built an algorithm that solves P_1. Thus solving P_2 is **at least as difficult** as solving P_1
Theorem: If $P_1 \leq_m P_2$, then

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

Proof: (First part) Let us assume that P_2 is decidable (REC, so we have a TM that always halts and solves P_2) and the theorem holds, so P_1 is not REC

- we apply the reduction to transform instance x of P_1 into instance y of P_2
- we appy on y the algorithm to decide P_2

We found an algorithm to decide P_1, which is a **contradiction**
Second part Let us assume that P_2 is RE, so there is a TM that halts for the strings in the language and the theorem holds, so P_1 is not RE

- we apply the reduction to transform instance x of P_1 into instance y of P_2
- we apply on y the algorithm to accept P_2 (it doesn't halt if y is a negative instance)

We have found a TM to accept P_1 (which doesn't halt if x is a negative instance) which is a **contradiction**
TM accepting non empty language: We consider two languages formed by TM encodings

$$L_e = \{enc(M) \mid L(M) = \emptyset\}$$

$$L_{ne} = \{enc(M) \mid L(M) \neq \emptyset\}$$

remembering that $enc(M)$ is a TM so, given as input a TM, so a language, in L_e there are the TMs with an associated empty language and in L_{ne} the vice-versa. Note that $\overline{L_e} = L_{ne}$
We want to find out whether these languages are recursive, or RE but non recursive, or else non-RE.
Theorem: L_{ne} is RE
Proof: We construct a nondeterministic TM M with $L(M) = L_{ne}$ as we can see in Figure 4.26

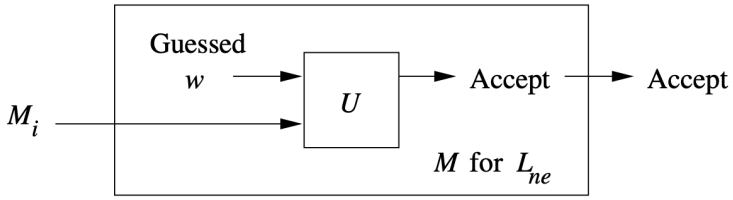


Figure 4.26: TM for L_{ne}

Given M_i as input, M implements the following strategy:

- using nondeterminism, **guess** a string w
- simulate U, the universal TM, on M_i and w

M accepts M_i, the input, if and only if there exists at least w such that $w \in L(M_i)$. The theorem then follows from the equivalence between nondeterministic TM and TM.
From the moment that there is a TM that halts only for if $L = L(M_i)$ is not empty, the language is in RE.
Theorem: L_{ne} is not in REC
Proof: In order to show that we can do a reduction from L_u to L_{ne}: we can find a map that maps every instances of L_u to an instance of L_{ne} maintaining the yes/yes and no/no property.
The reduction uses as target only two instances of L_{ne}

- the language Σ^* (positive instance, yes response)
- the empty language $\emptyset$ (negative instance, no response)

Let us transform any instance $enc(M, w)$ of L_u into an instance $enc(M)'$ of $L(M') = L_{ne}$ defined as follows, shown in Figure 4.27

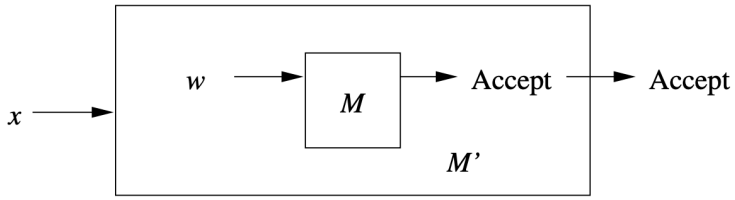


Figure 4.27: TM for reduction

M' ignores its input and uses its finite control to simulate a computation of M on w

- if M accepts w, so $w \in L(M)$, meaning that $L(M)$ is not empty, then M' accepts any input x, that is, $L(M') = \Sigma^*$; thus $L(M') \neq \emptyset$
- if M doesn't accept w, so $w \notin L(M)$, then M' does not accept any input that is $L(M') = \emptyset$ $\forall x$ in input, M' will not stop

And so we have constructed an algorithm M' such that the mapping is valid and also the reduction.
Theorem: L_e is not RE
Proof: We have to observe that $\overline{L_e} = L_{ne}$ and since L_{ne} is RE but is not recursive, L_e cannot be in RE (if it were, both languages would be recursive)

4.2.3 Properties of the languages generated by TMs

Languages L_e and L_{ne} are associated with decision problems related to **properties** of RE languages (languages generated by TMs): given a property P, remembering that a language is a set of string, we can define the language L_P that is the set of strings (encoding of TM), which the related TM recognizes a language with property P.
If we are not able to produce a TM that produces a language (so languages that are in RE), we can't verify if a language has the property P!
For this reason, these decision problems are related with the belonging to RE.
In what follows, we will concern with more general properties of RE languages, and the associated decision problems.
The fact that L_e and L_{ne} are undecidable is a special case of a more general theorem, known as Rice's theorem.
If we focus on RE as set of languages and a property P, we can subdivide the set in two subset: the languages which has the property P and languages that hasn't this property.
A problem of the RE languages is **trivial** if it is satisfied by all or none of the RE languages, so there is no subdivision of the set of RE.
Rice's theorem states that all properties P of the RE languages that are non trivial (in RE there are languages which have the property and other don't have the property) are undecidable.
This means that, for any nontrivial property P, there is no TM that

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

As example, we want to show that the property of be context free for a language is non-trivial such that, in CFL, there are languages that are RE and languages that are not CFL.
So, from Rice's theorem follows that it's undecidable decision problem.
Theorem-Rice's theorem: Any nontrivial property of RE languages is undecidable
Proof: Let P be a a non trivial property and assume that $\emptyset \notin P$.
Remember that L_P is

$$L_P = \{enc(M) \mid L(M) \in P\}$$

so the encoding of M that recognizes a language which has the property P.
We have to remember that a language is a set of string and, L_P is the set of strings

which encoding are languages that have the property P.
Suppose that L has the property P and M_L is the tm such that $L = L(M_L)$
We prove that $L_u \leq_m L_P$ using as target instances only two languages:

- $L(M_L)$ as positive instance from the moment that P is not empty (for assumption), picking a language $L \in P$ correspond to take a TM M_L such that $L(M_L) = L$
- $\emptyset$ as negative instance

So we have to create a map such that taking as input an instance of L_u $enc(M, w)$, if M recognizes w, then we have to map it to a TM M_L such that $L = L(M_L)$.
Given an instance $enc(M, w)$ for L_u, we produce an instance $enc(M')$ of L_P as suggests the following construction, shown in Figure 4.28

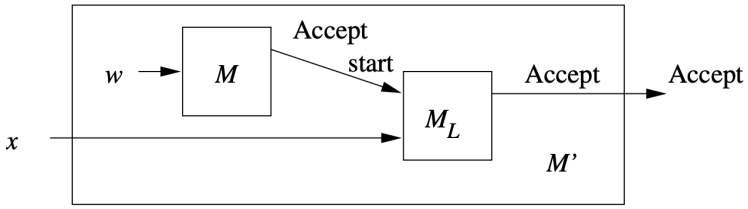


Figure 4.28: TM for reduction

- if M doesn't accept w, M' doesn't accept any input string and thus $L(M') = \emptyset \notin P$ so hasn't the property P
- if M accepts w, M' simulates M_L on instance x, and thus, $L(M') = L \in P$

From the moment that there is a reduction, L_P is in RE-REC.
Let us now assume that $\emptyset \in P$ and we consider $\overline{P}$, also non trivial, the set of RE languages that don't satisfy the property P.
Since $\emptyset \notin \overline{P}$, the previous proof proves that $L_u \leq_m L_{\overline{P}}$.
so $L_{\overline{P}}$ is not recursive. From the moment that

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

if L_P were recursive, also $\overline{L_P}$ will be and so also $L_{\overline{P}}$ would be recursive and this is a **contradiction** with respect to what we have previously asserted.
So we have shown that for both cases L_P with P nontrivial is in RE-REC.
This theorem is very important from the moment that it can be used to show that certain language are in RE-REC instead of using a reduction that is a more complex technique.

4.2.4 Post's correspondence problem

We now investigate to real problems, i.e. problems that don't concern TM. We show that Post's correspondence problem, which refers to string, is undecidable, using the following reductions, shown in Figure 4.29

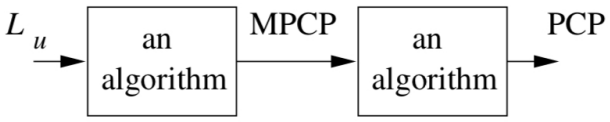


Figure 4.29: Reduction for PCP

We use this result to show that other real-word problems are undecidable.
An instance of Post's correspondence problem, or PCP for short, is formed by two equal length lists of strings

$$A = w_1, w_2, w_3, .., w_k$$

$$B = x_1, x_2, x_3, .., x_k$$

where $w_j, x_j \in \Sigma^*$.
Instance (A, B) has a solution if there are $m \geq 1$ indices $i_1, i_2, ..., i_m$ such that

$$w_{i_1} w_{i_2} w_{i_3} = x_{i_1} x_{i_2} x_{i_3}$$

In Figure 4.30 there is an example of this method

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

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

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

Figure 4.30: TM for reduction

Possible solutions are also all repetitions of $2, 1, 1, 3$
In Figure 4.31 another example that hasn't solution.

	A	B
i	w_i	x_i
1	10	101
2	011	11
3	101	011

This instance has no solution. To prove this, let us assume $i_1, i_2, \ldots, i_m$ is a solution

If $i_1 = 2$ or $i_1 = 3$ we have a **mismatch** at the first position. Then we must have $i_1 = 1$

Figure 4.31: TM for reduction

An instance of the **modified** PCP, MPCP, is an instance (A,B) of PCP and has a solution if there are $m \geq 0$ indices $i_1, i_2, ..., i_m$ such that

$$w_1 w_{i_1} w_{i_2} w_{i_3} = x_1 x_{i_1} x_{i_2} x_{i_3}$$

Noting that (w_1, x_1) must be the starting choice and m can be 0.
Theorem: $L_u \leq_m MPCP$
Theorem: $MPCP \leq_m PCP$ so PCP is **undecidable**

4.2.5 CFG Ambiguity

We assume a binary encoding for CFGs, similar to the one used for TM We write $enc(G)$ for the encoding of CFG G The **ambiguity problem** for a CFG is defined as follows

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

We define the corresponding language

$$L_{AMB} = \{enc(G) \mid G \; is \; ambiguous\}$$

We can find a reduction, by transforming instances of PCP to instances of L_{AMB}. Let (A, B) be an instance of PCP over the alphabet Σ, where $A = w_1, w_2, w_3, .., w_k$ and $B = x_1, w_2, w_3, .., x_k$ and let G_A be a CFG defined as:

- non terminal set $\{A\}$
- an alphabet $\Sigma \cup \{a_i \mid 1 \leq i \leq k\}$ with a_i an alias of the couple w_i, x_i
- a production set of this type

$$A \hookrightarrow w_1 A a_1 \mid w_2 A a_2 \mid ... \mid w_k A a_k$$
$$\hookrightarrow w_1 a_1 \mid w_2 a_2 \mid ... \mid w_k a_k$$

The strings generated by G_A are in the following form

$$w_{i_1} w_{i_2} w_{i_3} = a_{i_3} a_{i_2} a_{i_1}$$

as suggested by Figure 4.32

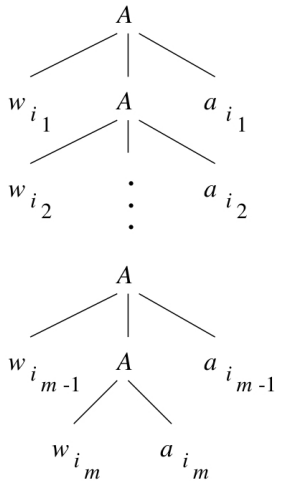


Figure 4.32: Parse tree for CFG G_A

It's defined symmetrically also G_B.
We observe that G_A and G_B are unambiguous. We define $L_A = L(G_A)$ and $L_B = L(G_B)$ and we also define G_{AB} that is the CFG that generates the language $L_A \cup L_B$ such that

- non terminal set $\{S, A, B\}$
- an alphabet $\Sigma \cup \{a_i \mid 1 \leq i \leq k\}$
- production set $S \hookrightarrow A|B$ and in addition all production of G_A and G_B

Theorem: $PCP \leq_m L_{AMB}$
Proof: We need to show that, for the given reduction, $enc(G_{AB}) \in L_{AMB}$ if and only if (A, B) has a solution.

If part: Let $i_1, i_2, ..., i_k$ be a solution for (A, B), then G_{AB} has two derivations for the same string

$$S \Rightarrow A \Rightarrow w_{i_1} A a_{i_1} \Rightarrow w_{i_1} w_{i_2} A a_{i_2} a_{i_1} \Rightarrow^* w_{i_1} w_{i_2} ... w_{i_k} a_{i_k} ... a_{i_2} a_{i_1}$$

$$S \Rightarrow B \Rightarrow x_{i_1} B a_{i_1} \Rightarrow x_{i_1} x_{i_2} B a_{i_2} a_{i_1} \Rightarrow^* w x_{i_1} x_{i_2} ... x_{i_k} a_{i_k} ... a_{i_2} a_{i_1}$$

Only if part: Assume G_{AB} ambiguous. Consider an ambiguous string in $L(G_{AB})$ having the form

$$z a_{i_m} ... a_{i_2} a_{i_1}$$

with $z \in \Sigma^+$. Since G_A and G_B are not ambiguous, the ambiguous string has to have two leftmost derivations starting with $S \Rightarrow A$ and $S \Rightarrow B$.
Then $i_1, i_2, ..., i_m$ is a solution for (A, B)

4.3 Intractability

In the previous section we have investigated about what can be decided or not, what can be compute or not.
In this chapter we will focus on what can be computed efficiently or rather in **polynomial time**. All these results are based on the statement that $P \neq NP$, which has not yet been proven or falsified. If, in the next future, we will prove that $P = NP$, all this chapter will have not sense.
A TM M has **time complexity** $T(n)$ if, for any input string w, with $|w| = n$, M halts after at most $T(n)$ computational steps.
A language L belongs to the class P if there exist a function $T(n)$ **polynomial**, such that $L = L(M)$ for some deterministic TM M with time complexity $T(n)$.
If the time complexity is not polynomial, we usually say that the time is **exponential**, even if the function is not exponential.
Analyzing the computational complexity of a TM presents two difficulties, as compared to the analysis of a computer algorithm:

- an algorithm can output a structure, while a TM just a response
- Algorithms have input alphabets of unlimited size, while TMs have finite input alphabet

A language L belongs to the class NP if there exist a function $T(n)$ **polynomial**, such that $L = L(M)$ for some **nondeterministic** TM, **NTM**, M with time complexity $T(n)$.
We know that every TM is also a NTM so $P \subseteq NP$ but we are not able to prove that $P \neq NP$: intuitively, a polynomial NTM can perform **exponential** number of computations "simultaneously", therefore is commonly assumed it.
To show that a problem P_2 cannot be solved in polynomial time, we reduce a problem $P_1 \notin P$ to P_2, as shown in Figure 4.33

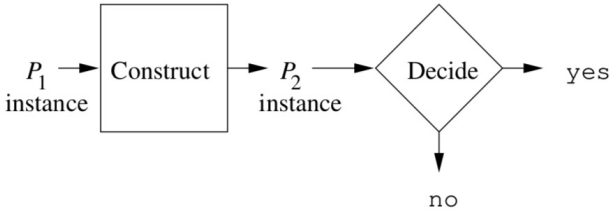


Figure 4.33: Reduction for NP problems

We impose the additional constraint that the reduction operates in **polynomial time**, so we can write $P_1 \leq_p P_2$.
Theorem: If $P_1 \leq_p P_2$, and $P_1 \notin P$, then $P_2 \notin P$
Proof: If $P_2 \in P$, we would have a polynomial time algorithm to solve P_1, from the moment that the reduction operates in polynomial time, so it's a contradiction
We can say that a language L is **NP-complete** if:

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

We focus on the most difficult NP problems among those in NP. If we take the Chomsky hierarchy, we can say that all the language $\in P$, are surely also CFL from the moment that there is an algorithm, CKY algorithm, that allows us to prove, in a polynomial time, that a string is in the language.
Some P problems are also in REC, but in REC there are also the NP problem (separated from P problem).
NP-complete problems are the most difficult in the class of NP.
Theorem: If P_1 IS NP-complete, $P_2 \in NP$ and $P_1 \leq_p P_2$, then P_2 is NP-complete.
Proof: The polynomial time reduction has the transitivity property so, for any language $L \in NP$ we have that $L \leq_p P_1$ and $P_1 \leq_p P_2$, and therefore $L \leq_p P_2$
Theorem: If a NP-complete problem is in P, then $P = NP$
Proof: Assume that a problem P is NP-complete and P$\in P$. For any language $L \in NP$ we have that $L \leq_p P$, and therefore we can solve L in polynomial time.
If we assume that $P \neq NP$, then we consider this proof an evidence that P $\notin P$.
A language L is NP-hard if, for each language $L' \in NP$, we have $L' \leq_p L$. In other words, L could be much more difficult than the problems in NP.

4.3.1 Satisfiability problem

Decide if a Boolean expression is satisfiable is an NP-complete problem. To show that we have to explicitly reduce each problem in NP to it.
Boolean expressions are composed by logical symbol:

- an infinite set of variables defined on Boolean values 0 or 1
- binary operators $\wedge$ as logical AND and $\vee$ logical OR
- unary operator $\neg$ as logical NOT

- brackets to force precedence

A Boolean expressions E can be defined recursively as union of these logical operators.
A **truth assignment** T for a Boolean expression E assigns a Boolean value $T(x)$, true or false, ad each variable.
After the assignment to each variable, we have a Boolean value $E(T)$ and T satisfies E if $E(T) = 1$.
So E is **satisfiable** if there exist such T that satisfies E.
In Figure 4.34 an example of satisfiability:

$x \wedge \neg(y \vee z)$ is satisfiable : $T(x) = 1,\ T(y) = 0,\ T(z) = 0$

$x \wedge (\neg x \vee y) \wedge \neg y$ cannot be satisfied

- we must have $T(x) = 1$ and $T(y) = 0$
- therefore $(\neg x \vee y)$ must be false

Figure 4.34: Satisfiability

The **satisfiability problem**, or SAT, is defined as given as input a Boolean expression, find if E is satisfiable.

We can encode a Boolean expressions in order to create the SAT language, formed by all Boolean expressions well-formed, correctly coded and satisfiable. We encode each symbol of the alphabet $\{x, y, z..\}$ as $\{x1.y10, z01...\}$. In Figure 4.35 an example of coded Boolean expression.

$$x \wedge \neg(y \vee z)$$

is encoded as

$$x1 \wedge \neg(x10 \vee x11)$$

Figure 4.35: Encoding of a Boolean expression

Theorem: The SAT is an NP-complete language
We've seen only the first part of the proof, such that $SAT \in NP$, from the moment that there is a NTM thar solves SAT.

We introduce a **simplified** version of SAT, called 3SAT, that is also an NP-complete problem and it is convenient to reduction that we will study after.
We define:

- the **literal** that is a variable x or the negation of a variable $x \hookrightarrow \bar{x}$,
- the **clause** that is the disjunction or logical OR with literals: $x \vee \bar{y} \vee z$

A Boolean expression in **conjuctive normal form** or CNF, is a conjuction or logical AND of clauses : $(x \vee \bar{y}) \wedge (x \vee z)$
We use use the + for the OR and the x for the AND, as shown in Figure 4.36

- $(x \vee \overline{y}) \wedge (\overline{x} \vee z)$ is written as $(x + \overline{y})(\overline{x} + z)$
- $(x + y\overline{z})(\overline{x} + y + z)$ is not in CNF
- xyz is in CNF

Figure 4.36: Example of CNF

A Boolean expression is in **k-conjunctive normal form** if it is in CNF and every clauses has exactly k literals: for example $(x \vee \bar{y}) \wedge (x \vee z)$ is in 2-CNF
We introduce the CSAT and kSAT problem or rather if a CNF or a k-CNF is satisfiable.
Theorem: CSAT is NP-complete
Proof: CSAT $\in NP$;CSAT $\leq_p$ CSAT
Theorem: 3SAT is NP-complete
Proof: 3SAT $\in NP$; CSAT $\leq_p$ 3SAT
Find out that a decision problem is NP-complete indicates that there are very few chances to discover an efficient algorithm for its solution. Typically decsion problems that are NP-complete are described according to the following scheme:

- problem name and abbreviation
- problem input and its encoding
- specification of positive instances of the problem
- problem used in the reduction for the NP-completeness result

In Figure 4.37 there is an example.

PROBLEM : satisfiability of Boolean expressions in 3-CNF (3SAT)
INPUT : Boolean expressions in 3-CNF
OUTPUT : "yes" if and only if the Boolean expressions is satisfiable
REDUCTION : from CSAT

Figure 4.37: Definition of a problem

4.3.2 INDEPENDENT SET PROBLEM

An important problem is the one related to the **independent set**: in a graph G, a subset I of the nodes is an indipendent set if no pair o nodes in I is connected by some arc of G.
An independent set is maximal if any other independent set of G has a number of nodes smaller or equal than the former.
In Figure 4.38 there is an example

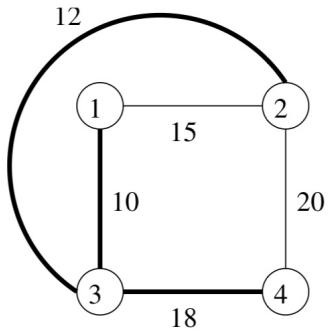


Figure 4.38: Independent set example

$I = \{1, 4\}$ is an independent set and is maximal because any set of three nodes from the graph has some arc connection
We consider the decision version of this problem, shown in the following Figure 4.39

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

Figure 4.39: Independent set problem

Theorem: IS is NP-complete
Proof: For proving that a problem is NP-complete we have to show that, first, it's NP, and then we find a reduction from 3SAT to show that is NP-complete
First part: $IS \in NP \hookrightarrow$ we use an NTM that:

- arbitrarily chooses k nodes using **nondeterminism**
- verifies that the chosen se is indipendent and accepts

These two steps can be performed in polynomial time in the size of the input data. From the moment that exist an NTM that solve IS in polynomial, we can say that is NP.
Second part: Now we can show that $3SAT \leq_p IS$
Let $E = e_1 \wedge e_2 \wedge .. \wedge e_m$ a Boolean expression in 3-CNF where each e_i is a clausen with 3 literals.
We build G with $3m$ nodes, with each node identified by a pair $[i, j]$ with $1 \leq i \leq m$ and $j \in \{1, 2, 3\}$. Then, the $[i, j]$ node represents the j literal in the i clause.
In Figure 4.40 an example:

$$E = (x_1 + x_2 + x_3)(\overline{x_1} + x_2 + x_4)(\overline{x_2} + x_3 + x_5)(\overline{x_3} + \overline{x_4} + \overline{x_5})$$

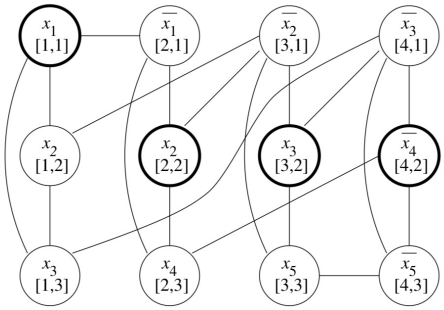


Figure 4.40: Independent set problem

We can construct the graph G as:

- one arc for each pair of nodes in the same column; then one can choose no more than one node per clause
- one arc for each pair of nodes $[i_1, j_1], [i_2, j_2]$ if these represent the literals x and $\hat{x}$;
  then one cannot choose two literals in an indipendent set if they are the negation of the other

Let $k = m$: we can build G and k in polynomial time in the length of the representation of E. We now prove that E is satisfiable $\iff$ G has an indipendent set with m elements.
$\Rightarrow$ part: let I an inidpendent set with m elements. We define:

- $T(x) = 1$ if the node for x is in I
- $T(x) = 0$ if the node for $\bar{x}$ is in I
- $T(x)$ can be arbitrarily defined if the nodes for x and $\bar{x}$ are not in I

Since nodes x and $\bar{x}$ cannot **simultaneously** belong to I, the definition of T is consistent.
An indipendent set I contain exactly one node per clause. It follows that the definition of T **satisfies** E.
$\Leftarrow$ part: let T an assignment that satisfies E. We arbitrarily choose a true literal for each clause, and we add to I the node associated with that literal.
I has a size m, and is an independent set:

- if an arc connects two nodes from the same clause, the two nodes are not both in I by construction
- if the remaining arcs connect two nodes corresponding to a literal and its negation, then the two nodes are not both in I because we have chosen only literals that are true in T

Another important problem is, given a graph G, a subset C is a **node cover** if each arc of G contains at least one node in C. A node cover is **minimal** if its size is smaller or equal than the size of any other node cover G.
In Figure 4.40 the definition of this problem:

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

Figure 4.41: Set covering

Theorem: NC is NP-complete
For the proof we use a reduction from IS.
Given G oriented graph, a **directed Hamiltonian circuit** in G is an oriented cycle that passes through each node of G **exaclty one**.

PROBLEM : directed Hamiltonian circuit (DHC)
INPUT : directed graph G
OUTPUT : "yes" if and only if G has a directed Hamiltonian circuit
REDUCTION : from 3SAT

Figure 4.42: DHC problem

In Figure 4.41 the definition of this problem:
Theorem: DHC is NP-complete
For the proof we use a reduction from 3SAT.
In Figure 4.42 we can see the reduction hierarchy for the NP-hard problem:

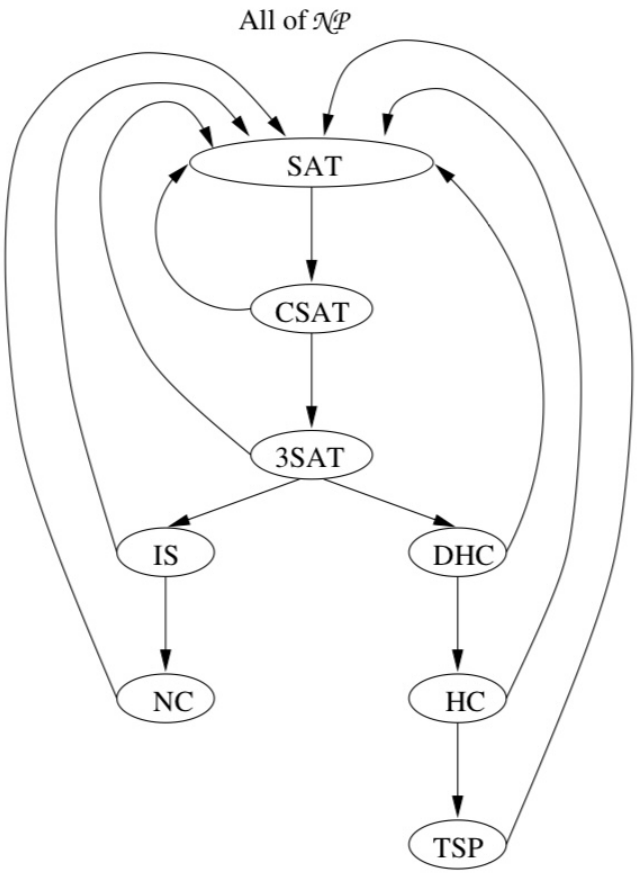


Figure 4.43: NP problems

where HC and TSP are respectively the problem of finding an undirected Hamiltonian circuit and the Traveling Salesman Problem