Theory of Computation Notes

şafak bilici

→ Alphabet is a set of symbols.

→ String is a set of symbols

→ Language is a set of strings

→ If a language is recognized by at least one FA, then it is a regular language.

Proof Techniques

Axioms: Assumptions about the underlying mathematical structures

Hypotheses: A proposed explanation made on the basis of limited evidence as a starting point for further investigation.

Lemma: Previously proved theorems.

→ Direct proof or Constructive proof or proof by construction

→ Nonconstructive proof

→ Proof by Contradiction

→ Proof by Induction

→ Pigeon Principle

Direct Proof

Theorem → If n is an odd positive integer, then n^2 is odd as well

Proof → An odd positive integer can be written as, $n = 2K+1$, K is integer ≥ 0, then

$$n^2 = (2k+1)^2 = 4k^2 + 4k + 1 = \underbrace{2(2k^2+2k)}_{\text{even}} + 1 \rightarrow \text{odd}$$

Proof by Contradiction

Theorem → $\sqrt{2}$ is irrational

Proof → Assume that $\sqrt{2}$ is rational. It can be written as p/q and p, q relatively primes.

$$\sqrt{2} = \frac{p}{q} \rightarrow \sqrt{2}q = p \rightarrow 2q^2 = p^2 \Longleftrightarrow p \in 2\mathbb{N}+1 \rightarrow \text{common factor is 2}$$

$$\rightarrow 2q^2 = 4r^2 \rightarrow q^2 = 2r^2 \Longleftrightarrow q \in 2\mathbb{N}+1$$

Proof by Induction

Theorem → $1+2+\cdots+n = n\cdot(n+1)/2$

proof → if $n=1$, it is 1 → prove it for $n+1$

Alphabet

$\Sigma_1 = \{0, 1\}$, $\Sigma_2 = \{a, b, c, \ldots, x, y, z\}$, $\Sigma_3 = \{0, 1, x, y, z\}$

$\Sigma_4 = \{\}$ → not alphabet

$\Sigma_5 = \{\{\}\}$ → alphabet

String

over an alphabet is a finite sequence of symbols from that alphabet.

$S_1 = 001011101$

$S_2 = 01$ $\quad |S_1| = 9 < \infty$

$S_3 = 000000$

Concatenation

$S = S_1 S_2 = 001011101$

$S^1 = S$

$S^2 = (001011101)^2 = 001011101001011101$

$S = 01$

$S^0 = \varepsilon \rightarrow |\varepsilon| = 0$, $(S^0)^2 = \varepsilon\varepsilon = \varepsilon$, $\epsilon^n = \epsilon$, $\epsilon S = S\epsilon = \epsilon S \epsilon = 01$

$\epsilon^0 = \epsilon$

Kleene Star

The set of possible strings from alphabet.

$\Sigma^1 = \{a, b\}$

$\Sigma^{1*} = \{\epsilon, a, b, aa, bb, ab, ba, \ldots\}$ → is an alphabet

Kleene Plus

The set of all possible strings from alphabet excluding empty string ϵ.

$\Sigma^1 = \{a, b\}$

$\Sigma^{1+} = \Sigma^{1*} - \{\epsilon\} = \{a, b, aa, bb, ab, ba, \ldots\}$

$\therefore$

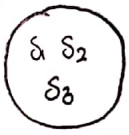
$L = \{\varepsilon\}$ is a language

$L^* = \{\epsilon, \epsilon\epsilon, \epsilon\epsilon\epsilon, \ldots\} = \{\epsilon\} \rightarrow |L^*| = 1$

$L^+ = L^* \setminus \epsilon = \{\} \rightarrow |L^+| = 0$

$L^+ = L L^*$

Language



A set of string over an alphabet.
Languages are used to describe computation problems.

Regular Languages

A subset of all languages. There is no way to define regular set. However it is possible to say whether a set is regular.

If some finite automata recognizes the language, then the language is regular.

> Union: $A \cup B = \{x \in A \text{ or } x \in B\}$

> Concatenation: $AB = \{xy \mid x \in A \text{ and } y \in B\}$

> Star: $A^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in A,\ i = 1, \ldots, k\}$

> Reverse: $w = a_1 a_2 a_3 \cdots a_n$

$w^R = a_n a_{n-1} \cdots a_1$

Prefix / Suffix

for sting $w = abba$

prefix	suffix
ϵ	abba
a	bba
ab	ba
abb	a
abba	ϵ

Language Examples

$\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$

$PRIME = \{x \mid x \in \Sigma^* \text{ and } x \in P\}$
$= \{2, 3, 5, 7, 11, \ldots\}$

$EVEN = \{x \mid x \in \Sigma^* \text{ and } x \in 2\mathbb{N}\}$
$= \{0, 2, 4 \ldots\}$

$ODD = \{x \mid x \in \Sigma^* \text{ and } x \in 2\mathbb{N}+1\}$
$= \{1, 3, 5, \ldots\}$

>> Unary Addition

$\Sigma = \{1, +, =\}$

$A = \{x+y=z : x = 1^m, y = 1^n, z = 1^k \mid k = m+n\}$

i.e., $111+11=11111 \in A$

$11+1 = 1 \notin A$

>> Squaring

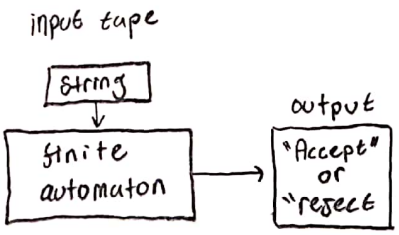
$\Sigma = \{1, *\}$

$S = \{x * y : x = 1^m, y = 1^n \mid n = m^2\}$

i.e., $111 * 111111111 \in S$

$1 * 1 \notin S$

Finite Automata



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

$Q \rightarrow$ finite set called states.

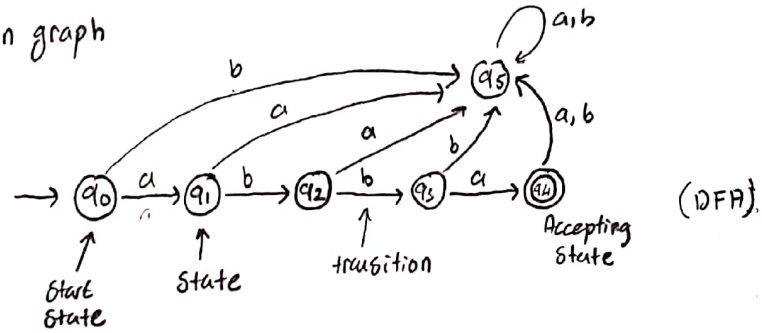
$\Sigma \rightarrow$ finite set called alphabet.

$\delta \rightarrow Q \times \Sigma \rightarrow Q$ is the transition function → you are in state r and input is a, next state is $\delta(r, a)$

$q_0 \rightarrow$ is the start state.

$F \rightarrow$ is the set of accept states.

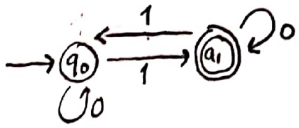
>> Transition graph



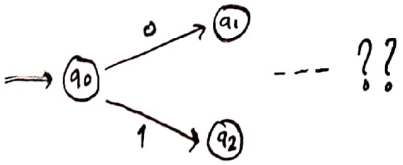
(DFA)

DFA - Deterministic Finite Automata

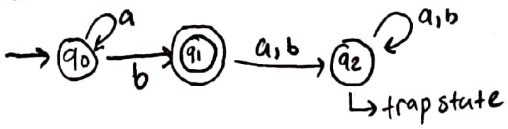
- $L = \{w \mid w \in \{0,1\}^* \text{ and } n_1(w) \text{ is odd}\}$



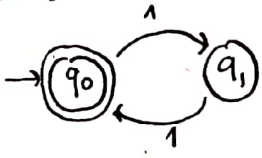
- $L = \{w \mid w \in \{0,1\}^* \text{ and } n_1(w) \text{ is odd}, n_0(w) \text{ is even}\}$



- $L = \{a^n b : n \geq 0\}$



- $L = \{X : X \in \{1\}^* \text{ and } x \in 2\mathbb{N}+1\}$

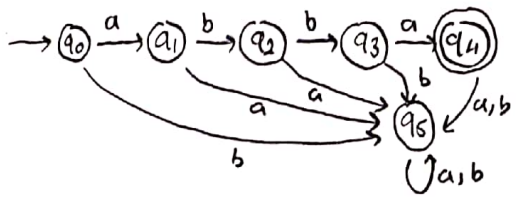


Extended Transition Function

$\delta^* : Q \times \Sigma^* \rightarrow Q$

$\delta(q, w) \rightarrow q'$

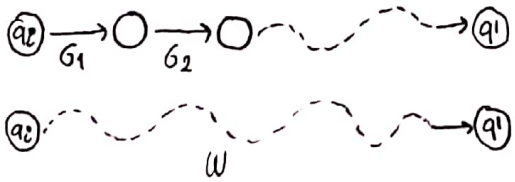
→ describes the resulting state after scanning string w to q_0.



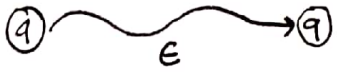
$\delta^*(q_0, ab) = q_2$

...in general, $\delta^*(q_i, w) = q'$ implies that there is a walk of transitions.

$w = G_1 G_2 \cdots G_k$



in addition...



Language Accepted by DFA

The language accepted by an automaton M, is denoted as $L(M)$ and contains all the strings accepted by M.

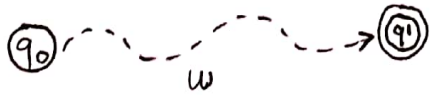
Language L' is recognized by the DFA M if

$L(M) = L'$

>> For DFA $M = (Q, \Sigma, \delta, q_0, F)$

Language accepted by M:

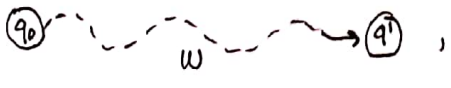
$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$



, $q' \in F$

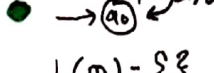
>> Language rejected by M:

$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}$

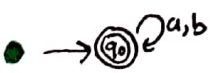


, $q' \notin F$

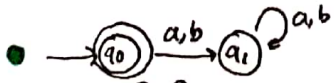
>> In addition, $L(M) \cup \overline{L(M)} = \Sigma^*$

- 

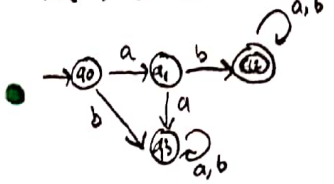
$L(M) = \{\}$

- 

$L(M) = \Sigma^*$

- 

$L(M) = \{\epsilon\}$

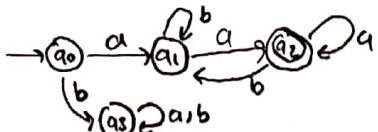
- 

$L(M) = \{w \in \Sigma^* \mid w = abV, V \in \Sigma^*\}$

- $L(M) = \{w \in \Sigma^* \mid w = abVba, V \in \Sigma^*\}$

No DFA accept that

- $L(M) = \{w \in \Sigma^* \mid w = aVa, V \in \Sigma^*\}$



- $L = \{a^n b^n \mid n > 0\}$

No DFA accept that

NFA - Non-Deterministic Finite Automata

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

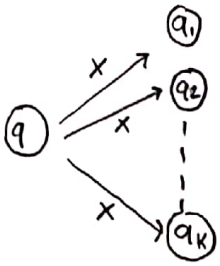
$Q \rightarrow$ Set of states

$\Sigma \rightarrow$ alphabet

$\delta \rightarrow$ Transition function: $Q \times \Sigma \rightarrow 2^{|Q|}$

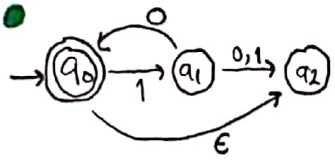
$q_0 \rightarrow$ start state

$F \rightarrow$ accepting states



$\delta(q, x) = \{q_1, q_2, \ldots, q_k\}$

resulting states with following one transition with symbol x

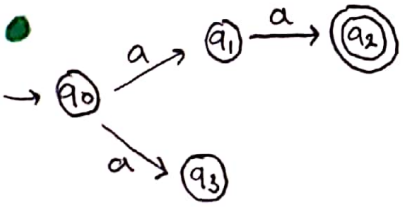
- 

$\delta(q_0, 1) = \{q_1\}$

$\delta(q_1, 0) = \{q_0, q_2\}$

$\delta(q_0, \epsilon) = \{q_2\}$

$\delta(q_2, 1) = \emptyset$

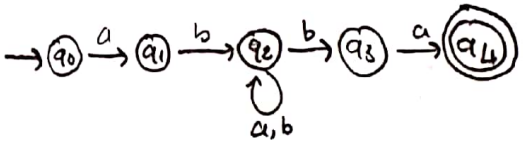
- 

$w = aa \in L(M)$

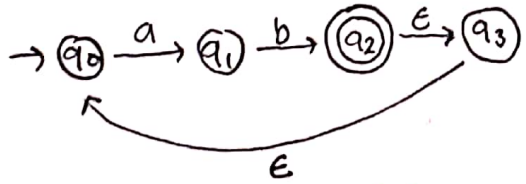
$w = a \notin L(M)$

$\Sigma = \{a\}$

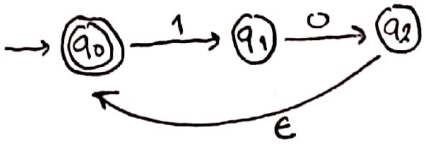
- $L(M) = \{w \in \Sigma^* \mid w = abVba, V \in \Sigma^*\}$



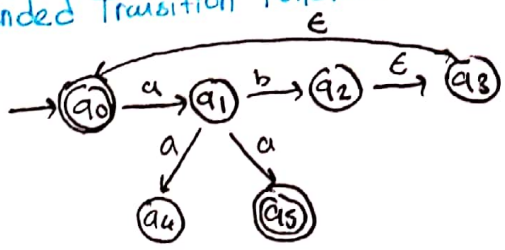
- $L(M) = \{w \in \Sigma^* \mid w = \{ab\}^+\}$



- $L(M) = \{w \in \Sigma^* \mid w = \{10\}^*\}$

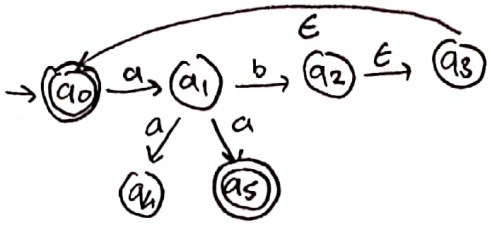


Extended Transition Function



$\delta^*(q_0, aa) = \{q_4, q_5\}$

$\delta^*(q_0, ab) = \{q_2, q_3, q_0\}$

- 

$L(M) = \{w \in \Sigma^* \mid w = \{ab\}^* \cup \{a,b\}^* aa\}$

Conversion NFA to DFA

>> inputs an NFA M

>> Output: an equivalent DFA M'

with $L(M) = L(M')$

>> The NFA has states

$q_0, q_1, q_2, \cdots$

>> The DFA has states from the power set

$\emptyset, \{q_0\}, \{q_1\}, \{q_0, q_1\}, \{q_1, q_2, q_8\}, \cdots$

Step 1:

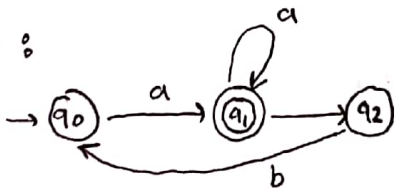
>> initial step of NFA: q_0

↓

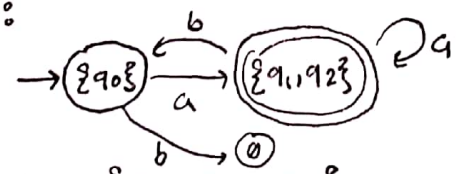
>> initial step of DFA: $\{q_0\}$

Example:

NFA M:



DFA M':



Step 2:

For every DFA's state $\{q_i, q_j, \cdots, q_m\}$

compute in the NFA

$$\left.\begin{array}{l} \delta^*(q_i, a) \\ \cup\, \delta^*(q_j, a) \\ \quad\vdots \\ \cup\, \delta^*(q_m, a) \end{array}\right\} = \{q_k', q_i', \cdots, q_n'\}$$

Add transition to DFA

$$\delta(\{q_i, q_j, \cdots, q_m\}, a) = \{q_k', q_i', \cdots q_n'\}$$

The Pumping Lemma

Regular languages are used to describe the language of the DFA/NFA machines. Each regular language has it own regular expressions.

$L(m) = \{ w \in \{a,b\}^* \mid w = a^n b,\ n \geq 0 \} \longrightarrow a^* b$

$L(m) = \{ w \in \{a,b,c\}^* \mid w = b^n c \text{ or } w = a \} \longrightarrow b^* c + a$

$L(m) = \{ w \in \{0,1\}^* \mid w \text{ starts with 1 and ends with 1},\ |w| > 2 \} \longrightarrow 1(0+1)^* 1$

$L(m) = \{ w \in \{0,1\}^* \mid w = 1^m 0 1^n 0 1^k;\ m \geq 0, n \geq 0, k \geq 0 \} \longrightarrow 1^* 0 1^* 0 1^*$

$L = \{ a^n b^n : n \geq 0 \} \longrightarrow$ non-regular language, cannot be recognized by NFA or DFA

→ If the language is finite, no need to use Pumping Lemma.

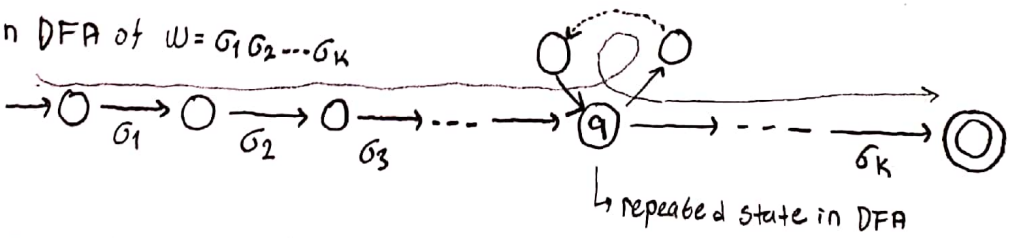
So, take an infinite regular language L,

there exists a DFA that accepts L that has m states $\longrightarrow |Q| = m$

Take a string $w \in L$ with $|w| \geq m$ (m is number of states, $|Q|$)

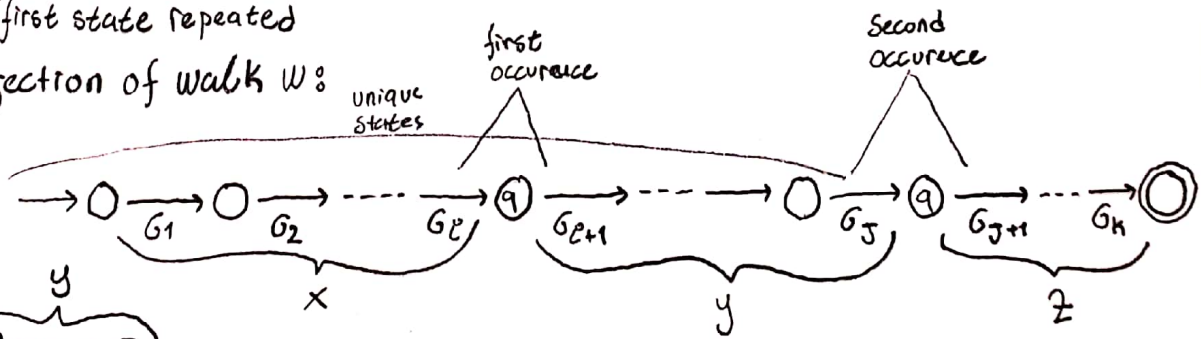
Then, at least one state is repeated in the walk of w.

→ Walk in DFA of $w = \sigma_1 \sigma_2 \cdots \sigma_k$

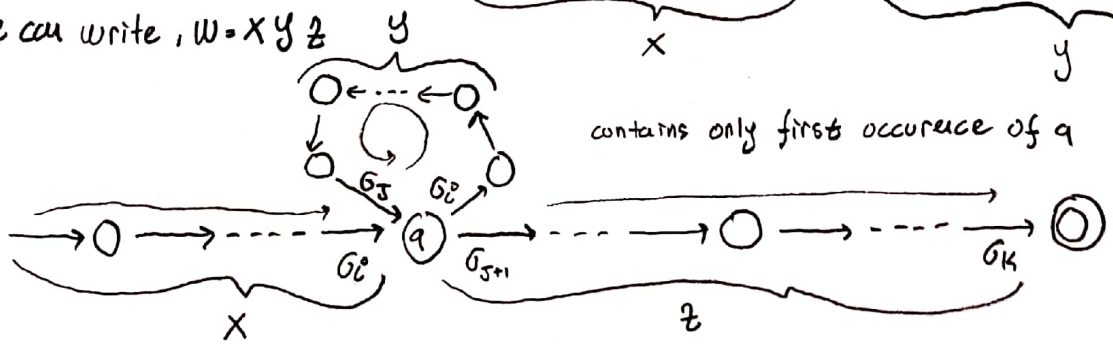


There could be many states repeated,
take q to be the first state repeated

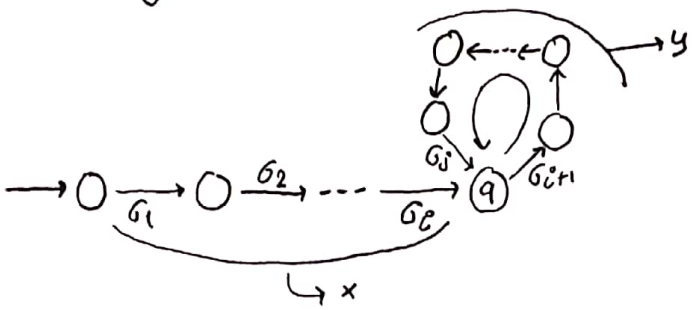
One dimensional projection of walk w:



We can write, $w = x\,y\,z$



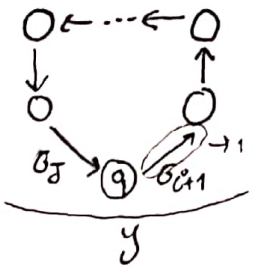
- Observation: length $|xy| \leq m$, m is number of states in DFA



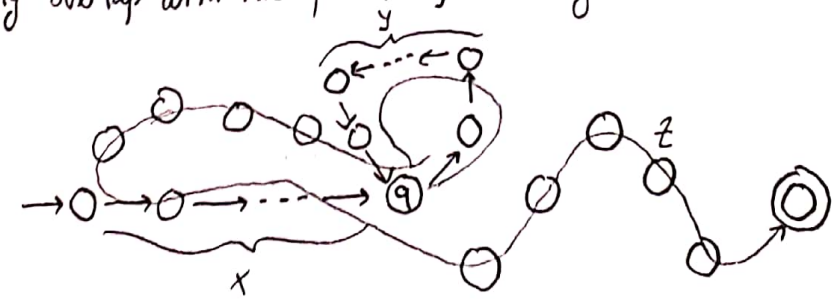
Unique states
Since, in xy, no state is repeated (except q)

- Observation: length $|y| \geq 1$

Since there is at least one transition in loop

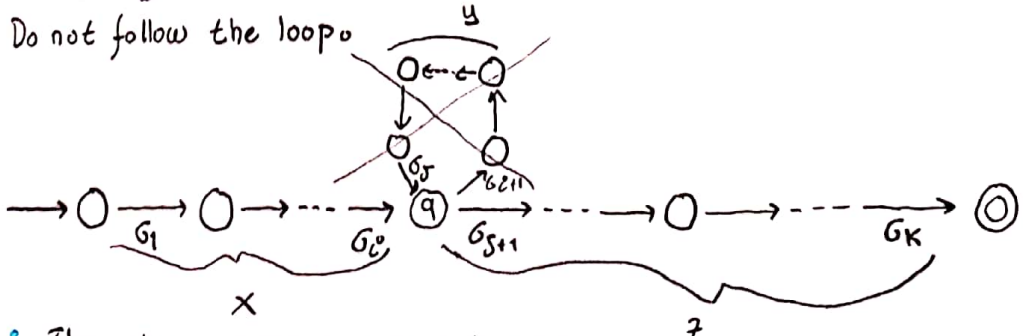


→ We dont care about the form of string z
z may actually overlap with the paths of x and y.



Additional String: The string xz is accepted

Do not follow the loop.



Additional Strings: The string xyyz is accepted → 2 loops
The string xyyyz is accepted → 3 loops
In general... The string $xy^{i^0}z$ is accepted → i^0 loops, $i = 0,1,2\ldots$
Therefore → $xy^{i^0}z \in L$
↑
Language is accepted by the DFA

The Pumping Lemma is :

- Given a infinite regular language L
- There exists an integer m (critical length)
- For any string $w \in L$ with length $|w| \geq m$
- We can write $w = xyz$
- With $|xy| \leq m$ and $|y| \geq 1$
- Such that : $xy^i z \in L$

Applications of Pumping Lemma

Theorem: The language $L = \{a^n b^n : n \geq 0\}$ is not regular

Proof: Assume that L is regular. Since L is infinite, we can apply the Pumping Lemma.

Let m is critical length for L

Pick a string w such that $w \in L$ and $|w| \geq m$

We pick $w = a^m b^m$

We can rewrite $w = a^m b^m = xyz$

With lengths $|xy| \leq m$, $|y| \geq 1$

$$w = xyz = a^m b^m = \overbrace{\underbrace{a\ldots}_{x}\underbrace{a\ldots}_{y}a}^{m}\overbrace{\underbrace{a\ldots}_{z}b\ldots b}^{m}$$

Thus : $y = a^k$, $1 \leq k \leq m$

From the Pumping Lemma : $xy^0 z \in L$

Thus : $xy^2 z \in L$

$$xy^2 z = \overbrace{\underbrace{a\ldots}_{x}\underbrace{a\ldots}_{y}\underbrace{a\ldots}_{y}a}^{m+k}\overbrace{\underbrace{a\ldots}_{z}b\ldots b}^{m} \in L$$

Thus : $a^{m+k} b^m \in L$

But $L = \{a^n b^n : n \geq 0\}$

$\Downarrow$

$a^{m+k} b^m \notin L$, due to $m + k \neq m$

Theorem: The language $L = \{VV^R : V \in \Sigma^*\}$ is not regular. $(\Sigma = \{a, b\})$

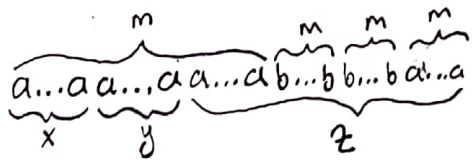
Proof: Assume that L is regular.

Let m to be critical length.

Pick a string $w \in L$

we pick $w = a^m b^m b^m a^m$

$w = xyz$, $|xy| \leq m$, $|y| \geq 1$

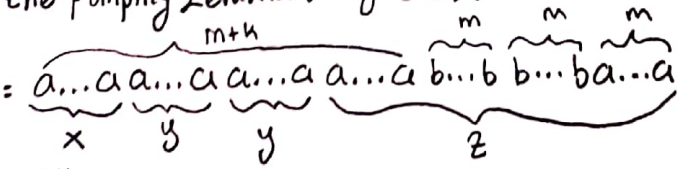
$w = xyz =$ 

Thus: $y = a^k$, $1 \leq k \leq m$

From the Pumping Lemma: $xy^0z \in L$

Thus: $xy^2z \in L$

From the Pumping Lemma: $xy^2z \in L$

$xy^2z =$ 

Thus: $a^{m+k} b^m b^m a^m \in L$

Since, $L = \{VV^R : V \in \Sigma^*\}$

$a^{m+k} b^m b^m a^m \notin L$

Theorem: $L = \{a^n b^l c^{n+l} : n, l \geq 0\}$ is not regular.

Proof: Assume that L is regular.

$m \rightarrow$ critical length

pick $\rightarrow w = a^m b^m c^{2m} \in L$

$w = a^m b^m c^{2m} = xyz \longrightarrow |xy| \leq m,\ |y| \geq 1$

$w = xyz = \overbrace{a\ldots a}^{x}\overbrace{a\ldots a}^{y}\overbrace{a\ldots a\, b\ldots b\, c\ldots c\, c\ldots c}^{z}$ (m, m, 2m)

$y = a^k$

$xy^0z \in L$

Thus: $xy^0z = xz \in L$

$xz = \overbrace{a\ldots a}^{x}\overbrace{a\ldots a\, b\ldots b\, c\ldots c\, c\ldots c}^{z}$ (m−k, m, 2m)

Thus: $a^{m-k} b^m c^{2m} \in L$

Since, $L = \{a^n b^l c^{n+l} : n, l \geq 0\}$

$a^{m-k} b^m c^{2m} \notin L$ due to $m-k+m \neq 2m$

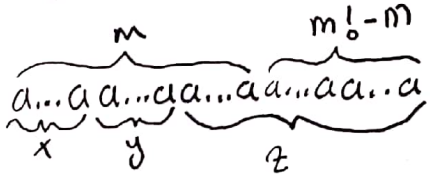
Theorem: $L = \{a^{n!} : n \geq 0\}$ is not regular.

Proof: Assume that L is regular.

$m \rightarrow$ critical length

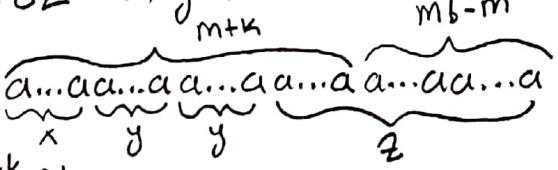
$w \rightarrow a^{m!}$, $|w| \geq m$

$w = xyz \rightarrow |xy| \leq m,\ |y| \geq 1$

$w = xyz = a^{m!} =$ 

Thus: $y = a^k$

$\rightarrow xy^0z \in L \rightarrow xy^2z \in L$

$w = xy^2z =$ 

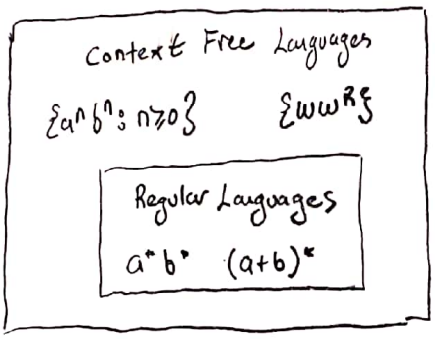
Thus: $a^{m!+k} \in L$

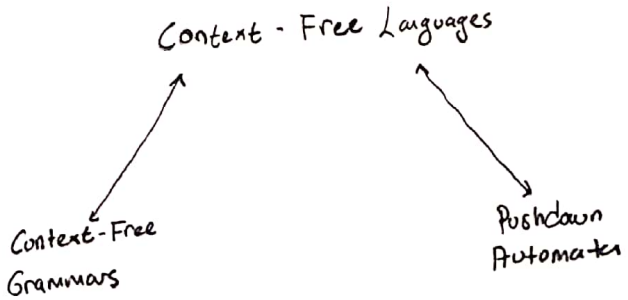
For $L = \{a^{n!} : n \geq 0\}$ there must exist p such that: $m! + k = p!$

$m! + k = p! \leq m! + m! < m!m + m! = m(m!+1) = (m+1)!$

$m! + k \leq (m+1)! \rightarrow m! + k \neq p! \rightarrow a^{m!+k} \notin L$

Context-Free Languages





Context-Free Grammars

Grammars

>> Grammars express languages.

>> Example: the English language grammar.

$\langle sentence \rangle \rightarrow \langle noun\text{-}phrase \rangle \langle predicate \rangle$

$\langle noun\_phrase \rangle \rightarrow \langle article \rangle \langle noun \rangle$

$\langle predicate \rangle \rightarrow \langle verb \rangle$

$\langle article \rangle \rightarrow a$

$\langle article \rangle \rightarrow the$

$\langle noun \rangle \rightarrow dog$

$\langle noun \rangle \rightarrow cat$

$\langle verb \rangle \rightarrow runs$

$\langle verb \rangle \rightarrow sleeps$

>> Derivation of string "the dog walks"

$\langle sentence \rangle \rightarrow \langle noun\_phase \rangle \langle predicate \rangle$

$\rightarrow \langle noun\_phase \rangle \langle verb \rangle$

$\rightarrow \langle article \rangle \langle noun \rangle \langle verb \rangle$

$\rightarrow$ the $\langle noun \rangle \langle verb \rangle$

$\rightarrow$ the dog $\langle verb \rangle$

$\rightarrow$ the dog walks

>> Language of the grammar:

$L = \{$ "a cat runs",
"a cat sleeps",
"the cat runs",
"the cat sleeps",
"a dog runs",
"a dog sleeps",
"the dog runs",
"the dog sleeps", $\}$

Productions

Sequence of terminals (symbols)

⟨noun⟩ → cat

⟨sentence⟩ → ⟨noun_phrase⟩⟨predicate⟩

Variables

↳ Sequence of variables

Another example:

↳ Sequence of terminals and variables

Grammar: $S \to aSb$

$S \to \epsilon$

↳ the right may be ϵ

Variable

>> Grammar:

$S \to aSb$

$S \to \epsilon$

>> Derivation of string: ab

$S \to aSb \to ab$

$S \to aSb$ $\quad$ $S \to \epsilon$

>> Grammar:

$S \to aSb$

$S \to \epsilon$

>> Derivation of string: aabb

$S \to aSb \to aaSbb \to aabb$

$S \to aSb$ $\quad$ $S \to \epsilon$

Grammar:

$S \to aSb$

$S \to \epsilon$

Language of The Grammar:

$L = \{a^n b^n : n \geq 0\}$

A Convenient Notation

>> we write

$S \xrightarrow{*} aaabbb$

for one or more derivation steps

>> instead of

$S \to aSb \to aaSbb \to aaaSbbb \to aaabbb$

In general we write

$w_1 \xrightarrow{*} w_n$

if: $w_1 \to w_2 \to \cdots \to w_n$

in zero or more derivation steps

Example Grammar	Possible Derivations
$S \to aSb$	$S \to \epsilon$
$S \to \epsilon$	$S \xrightarrow{*} ab$
	$S \xrightarrow{*} aaabbb$

Another Convenient Notation

$S \to aSb$
$S \to \epsilon$
$\Longrightarrow S \to aSb \mid \epsilon$

$\langle article \rangle \to a$
$\langle article \rangle \to the$
$\Longrightarrow \langle article \rangle \to a \mid the$

Formal Definition

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

V → set of variables
T → set of terminal symbols
S → start variable
P → set of productions

Example

Grammar $S \to aSb \mid \epsilon$

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

$\{S\}$ $\{a, b\}$ $\{S\}$ $\{S \to aSb, S \to \epsilon\}$

Language of Grammar:

» For a grammar G with start variable S

$L(G) = \{w : S \overset{*}{\to} w, w \in T^*\}$

↳ string of terminals or ϵ

Example

$G: S \to aSb \mid \epsilon$

$L(G) = \{a^n b^n : n \geq 0\}$

since there is derivation $S \overset{*}{\to} a^n b^n$ for any $n \geq 0$

Example

$G: S \to aSa \mid bSb \mid \epsilon$

$L(G) = \{ww^R : w \in \{a, b\}^*\}$

Example

$G: S \to aSb \mid SS \mid \epsilon$ → Example derivations: $S \to SS \to aSbS \to abS \to ab$

$S \to SS \to aSbS \to abS \to abaSb \to abab$

$L(G) = \{w : n_a(w) = n_b(w)$, and $n_a(v) \geq n_b(v)$ in any prefix $v\}$

Derivation Trees

Derivation Order

Consider the following example grammar with 5 productions.

1. $S \rightarrow AB$ 2. $A \rightarrow aaA$ 4. $B \rightarrow Bb$
3. $A \rightarrow \epsilon$ 5. $B \rightarrow \epsilon$

>> Leftmost derivation order of string aab:

$$S \rightarrow AB \rightarrow aaAB \rightarrow aaB \rightarrow aaBb \rightarrow aab$$

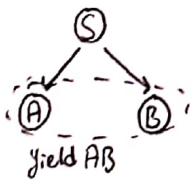
>> Rightmost derivation order of string aab:

$$S \rightarrow AB \rightarrow ABb \rightarrow Ab \rightarrow aaAb \rightarrow aab$$

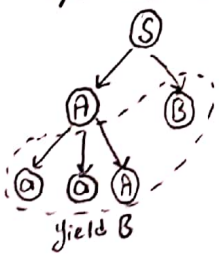
Derivation Tree

$S \rightarrow AB$, $A \rightarrow aaA \mid \epsilon$, $B \rightarrow Bb \mid \epsilon$

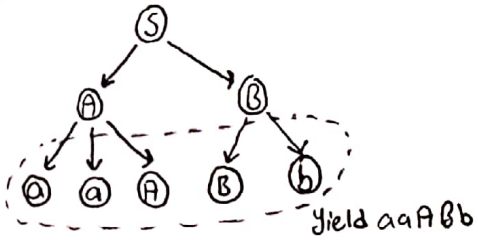
1) $S \rightarrow AB$



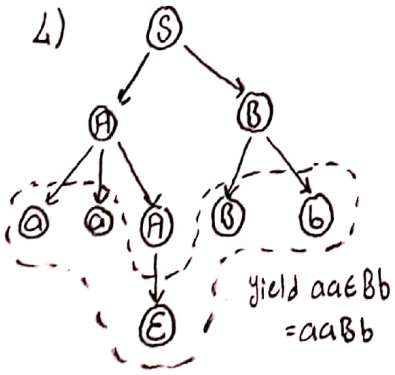
2) $S \rightarrow AB \rightarrow aaAB$



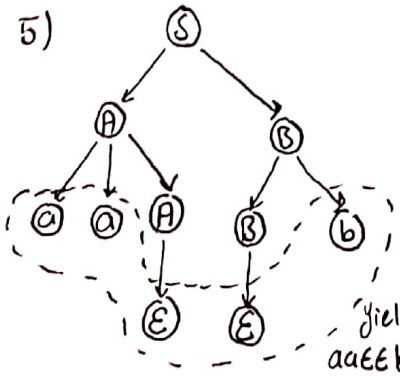
3) $S \rightarrow AB \rightarrow aaAB \rightarrow aaABb$



4)



5)



Ambiguity

Grammar For Mathematical Expressions

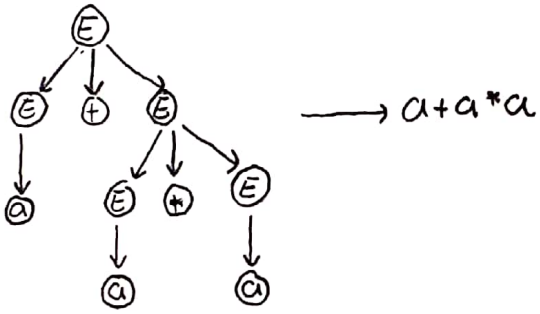
$E \rightarrow E+E \mid E*E \mid (E) \mid a$

>> Example string

$(a+a)*a+(a+a*(a+a))$

>> Leftmost derivation ①

$E \rightarrow E+E \rightarrow a+E \rightarrow a+E*E \rightarrow a+a*E \rightarrow a+a*a$

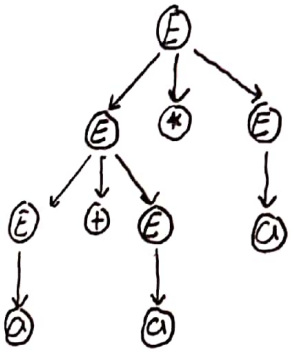


$\longrightarrow a+a*a$

Good Tree

>> Another leftmost derivation ②

$E \rightarrow E*E \rightarrow E+E*E \rightarrow a+E*E \rightarrow a+a*E \rightarrow a+a*a$



Bad Tree

>> Two derivation trees for $a+a*a$

Take $a=2$

- Tree one's result is 6.
- Tree two's result is 8.

Two different derivation trees may cause problems in applications which use derivation trees:

- Evaulating Expressions
- In general, in compilers for programming languages.

Ambiguous Grammar

A context-free grammar G is ambiguous if there is string $w \in L(G)$ which has two different derivation trees or two leftmost derivations.

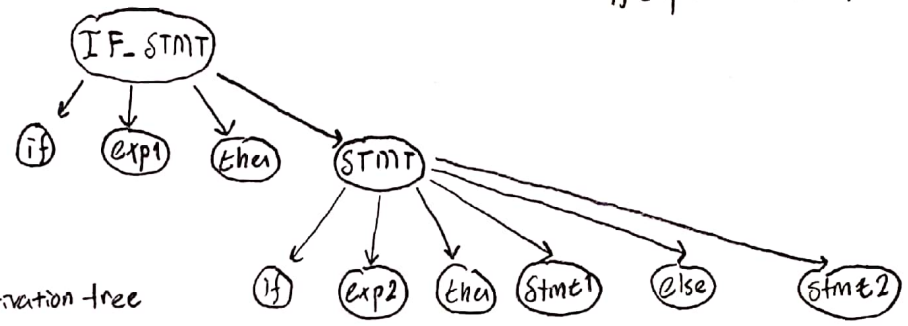
>> Example to ambiguous grammar:

IF_STMT → if EXPR then STMT | if EXPR then STMT else STMT

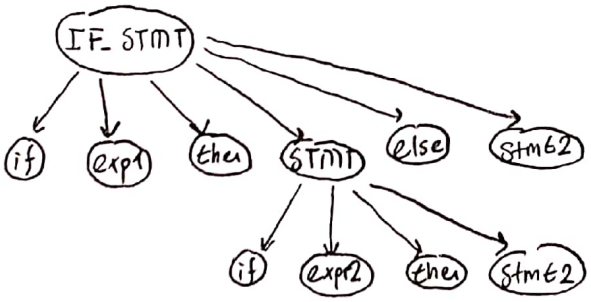
Variables

Terminals

if exp1 then if exp2 then stmt1 else stmt2



two different derivation tree



>> In general, ambiguity is bad and we want to remove it.
Sometimes it is possible to find a non-ambiguous grammar for a language.
But in general, we cannot do so.

>> A successful example:

Ambiguous Grammar:

$E \to E+E$

$E \to E*E$

$E \to (E)$

$E \to a$

Equivalent Non-Ambiguous Grammar:

$E \to E+T \mid T$

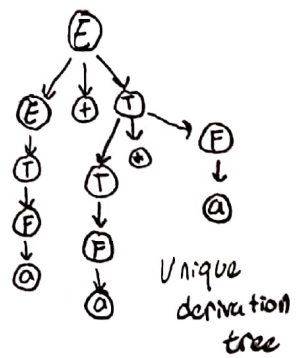
$T \to T*F \mid F$

$F \to (E) \mid a$

$E \to E+T \to T+T \to F+T \to a+T$

$\to a+T*F \to a+F*F \to a+a*F$

$\to a+a*a$



Unique derivation tree

>> An un-successful example:

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\} \; ; \; n, m \geq 0$$

L is inherently ambiguous:

every grammar that generates this language is ambiguous.

>> Example (ambiguous) grammar for L:

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$$

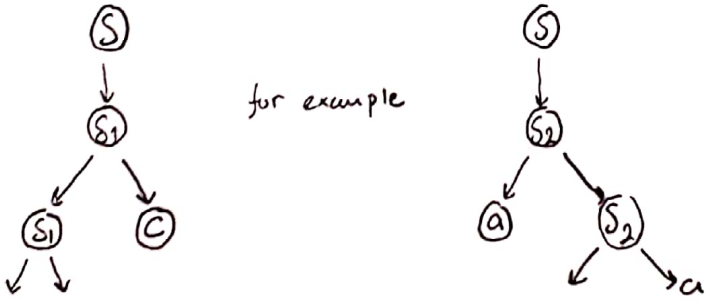
$S \rightarrow S_1 | S_2$

$S_1 \rightarrow S_1 c | A$
$A \rightarrow aAb | \epsilon$

$S_2 \rightarrow aS_2 | B$
$B \rightarrow bBc | \epsilon$

The string $a^n b^n c^n \in L$

has always two different derivation trees (for any grammar)



The Pumping Lemma For CFLs

» For every CFL L, there is an integer n, such that for every string z in L of length $>n$, there exists $z = uvwxy$ such that:

1- $|vwx| < n$

2- $|vx| > 0$

3- $uv^i wx^i y$ is in L, $i \geq 0$

Proof of the Pumping Lemma

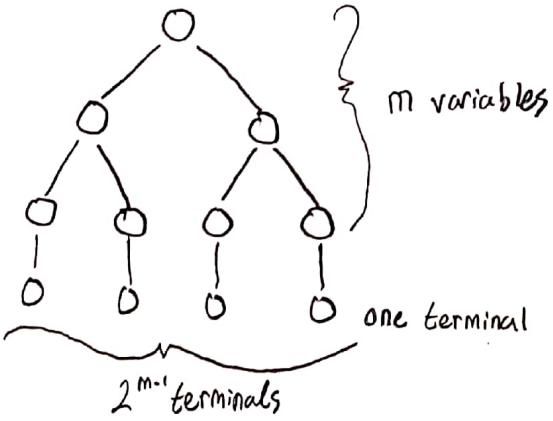
» start with a Chomsky Normal Form grammar for $L - \{\epsilon\}$

» Let the grammar have m variables.

- pick $n = 2^m$
- let $|z| > n$

» we claim ("Lemma 1") that a parse tree with yield z must have a path of length $m+2$ or more.

» if all paths in the parse tree of a CNF grammar are of length $< m+1$, then the longest yield has length 2^{m-1}:



Using the Pumping Lemma

>> Non-CFL's typically involve trying to match two pairs of counts or match two strings.

>> Example: pumping lemma can be used to show that $L = \{ww \mid w \in \{0+1\}^*\}$ is not a CFL.

>> $L = \{0^i 1 0^i 1 0^i \mid i > 1\}$ is not a CFL.

- We cant match two pair, or three counts as a group
- proof using the pumping lemma
- Suppose L were a CFL.
- Let n be L's pumping lemma constant.
- consider $z = 0^n 1 0^n 1 0^n$
- we can write $z = uvwxy$, $|vwx| < n$ and $|vx| > 1$
- case 1: vx has no 0's.
- Then at least one of them is a 1, and uwy has at most one 1, which no string in L does.
- Case 2: vx has at least one.
- vx is too short (length $< n$) to extend to all three blocks of 0's in $0^n 1 0^n 1 0^n$.
- Thus, uwy has at least one block of n 0's, and at least one block with fewer than n 0's.
- Thus, uwy is not in L.

Simplifications of Context-Free Grammars

A Substitution Rule

Example

$S \to aB$

$A \to aaA$

$A \to abBc$

$B \to aA$

$B \to b$

$B \to b$ — Substitute

$S \to aB \mid ab$ → Equivalent Grammar

$A \to aaA$

$A \to abBc \mid abbc$

$B \to aA$

Example

$S \to aB \mid ab$

$A \to aaA$

$A \to abBc \mid abbc$

$B \to aA$

Substitute $B \to aA$

$S \to aB \mid ab \mid aaA$ → Equivalent Grammar

$A \to aaA$

$A \to abBc \mid abbc \mid abaAc$

>> In general:

$A \to xBz$

$B \to y_1$

Substitute $B \to y_1$

$A \to xBz \mid xy_1z$

Nullable Variables

$S \to aMb$

$M \to aMb$

$M \to \epsilon$ → ϵ-production

M: nullable variable

Removing ϵ-productions:

$S \to aMb$

$M \to aMb$

$M \to \epsilon$

Substitute $M \to \epsilon$

$S \to aMb \mid ab$

$M \to aMb \mid ab$

Unit Productions:

$X \rightarrow Y$ (a single variable in both sides)

Example

$S \rightarrow aA$
$A \rightarrow a$
$A \rightarrow B$ — Unit productions
$B \rightarrow A$
$B \rightarrow bb$

- Removal of Unit productions:

$S \rightarrow aA$
$A \rightarrow a$
$A \rightarrow B$
$B \rightarrow A$
$B \rightarrow bb$

Substitute $A \rightarrow B$ →

$S \rightarrow aA \mid aB$
$A \rightarrow a$
$B \rightarrow A \mid B$
$B \rightarrow bb$

- Unit productions of form $X \rightarrow Y$ can be removed

$S \rightarrow aA \mid aB$
$A \rightarrow a$
$B \rightarrow A \mid B$
$B \rightarrow bb$

$B \rightarrow B$ remove →

$S \rightarrow aA \mid aB$
$A \rightarrow a$
$B \rightarrow A$
$B \rightarrow bb$

$S \rightarrow aA \mid aB$
$A \rightarrow a$
$B \rightarrow A$
$B \rightarrow bb$

Substitute $B \rightarrow A$ →

$S \rightarrow aA \mid aB \mid \cancel{aA}$
$A \rightarrow a$
$B \rightarrow bb$

→ final grammar

Useless Productions

Example

$S \to aSb$

$S \to \lambda$

$S \to A$

$A \to aA$ ⟶ useless production

Some derivations never terminate:

$S \to A \to aA \to aaA \to \dots \to aa\dots aA \dots$

Example

$S \to A$

$A \to aA$

$A \to \lambda$

$B \to bA$ ⟶ useless production

not reachable from S

» A production $A \to x$ is useless if any of its variables is useless

$S \to aSb$

$S \to \epsilon$

$S \to A$ useless

$A \to aA$ useless

$B \to C$ useless

$C \to D$ useless

» Removing useless variables and productions.

Example

$S \to aS \mid A \mid C$

$A \to a$

$B \to aa$ → useless variable

$C \to aCb$ → nullable

Step1 → remove nullable variables

Step2 → remove unit productions

Step3 → remove useless variables

↳ $S \to aS \mid A$

$A \to a$

$B \to aa$

↳ $S \to aS \mid a$

$A \to a$

Normal Forms for Context-Free Grammars

Chomsky Normal Form

$A \to BC$ or $A \to a$ > each production has a form

BC: Variables; a: terminal

Example

$S \to AS$	$S \to AS$
$S \to a$	$S \to AAS$
$A \to SA$	$A \to SA$
$A \to b$	$A \to aa$
Chomsky normal form	Not chomsky normal form

Conversion to Chomsky Normal Form

Example

$S \to ABa$
$A \to aab$
$B \to Ac$

} Not in Chomsk Normal Form

» Converting to CNF:

- Introduce new variables for the terminals: T_a, T_b, T_c

$S \to ABa$ → $S \to ABT_a$
$A \to aab$ → $A \to T_aT_aT_b$
$B \to Ac$ → $B \to AT_c$

$T_a \to a$
$T_b \to b$
$T_c \to c$

- Introduce new intermediate variable V_1 to break first production.

$S \to ABT_a$
$A \to T_aT_aT_b$
$B \to AT_c$
$T_a \to a$
$T_b \to b$
$T_c \to c$

→

$S \to AV_1$
$V_1 \to BT_a$
$A \to T_aT_aT_b$
$B \to AT_c$
$T_a \to a$
$T_b \to b$
$T_c \to c$

• Introduce intermediate variable V_2

$S \to AV_1$
$V_1 \to BT_a$
$A \to T_aT_aT_a$
$B \to AT_c$
$T_a \to a$
$T_b \to b$
$T_c \to c$

$\longrightarrow$

$S \to AV_1$
$V_1 \to BT_a$
$A \to T_aV_2$
$V_2 \to T_aT_b$
$B \to AT_c$
$T_a \to a$
$T_b \to b$
$T_c \to c$

Final Grammar in Chomsky Normal Form

>> In general, from any CFG (which doesnt produce ϵ) not in Chomsky Normal Form. We can obtain an equivalent grammar in Chomsky Normal form.

The Procedure

>> First remove:

Nullable Variables

Unit Productions

>> Then, for every symbol a:

New variable: T_a

Add production: $T_a \to a$

>> In productions with length at least 2 replace a with T_a

Productions of form $A \to a$ do not need to change!

>> Replace any production $A \to C_1C_2 \cdots C_n$ with $A \to C_1V_1, V_1 \to C_2V_2, \cdots, V_{n-2} \to C_{n-1}C_n$

New intermediate variables $V_1, V_2, \cdots, V_{n-2}$

Observations

>> CNFs are good for parsing and prooving theorems.

>> It is easy to find CNF for any CFG.

Greinbach Normal Form

$S \to abSb$
$S \to aa$

$\longrightarrow$

$S \to aT_bST_b$
$S \to aT_a$
$T_a \to a$
$T_b \to b$

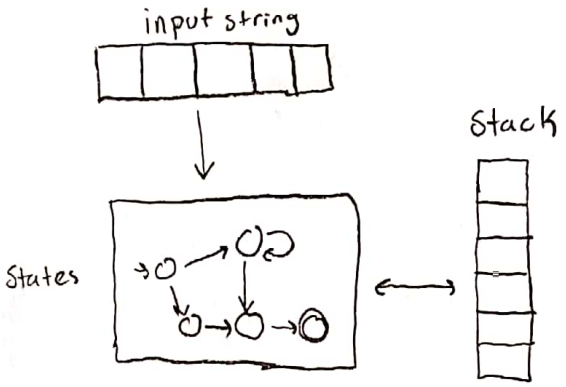
Greinbach Normal Form

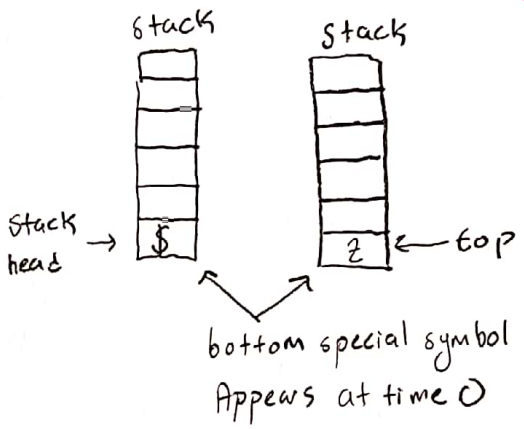
>> Greinbach Normal Forms are very good for parsing strings, better than CNF

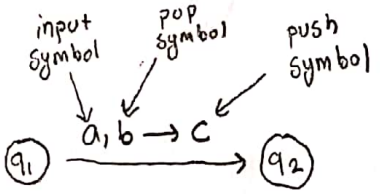
>> However, it is difficult to find the Greinbach Normal Form of a grammar.

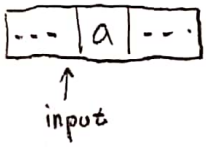
Pushdown Automaton

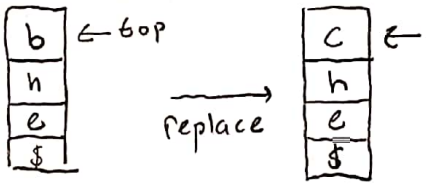
PDA

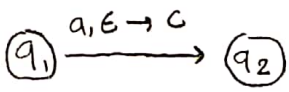


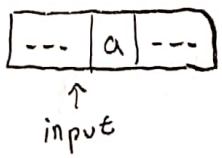


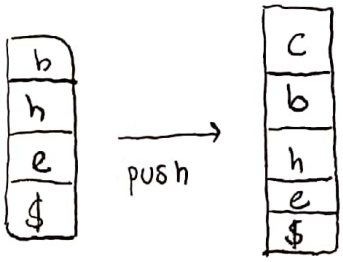


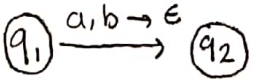


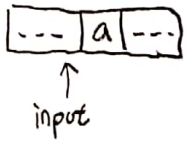


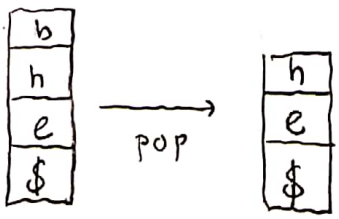


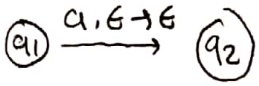




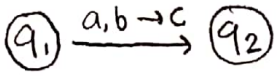








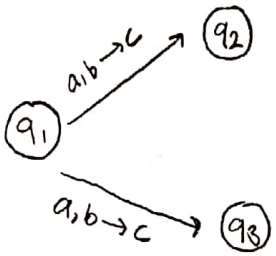
, no change

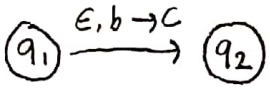


; if stack is empty, automaton halts and rejects input.

Non-Determinism

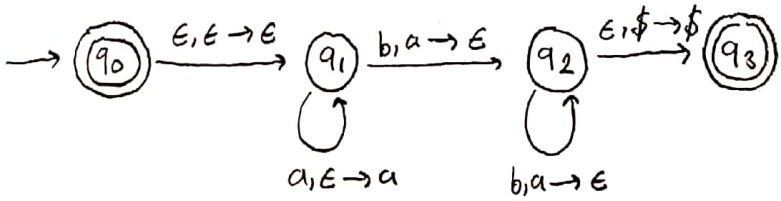
PDAs are non-deterministic.





Example

PDA M: $L(M) = \{a^n b^n : n \geq 0\}$



» A string is accepted if there is a computation such that:

All the input is consumed

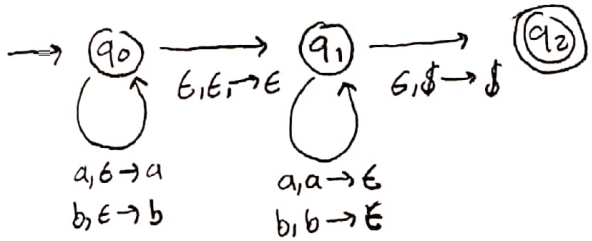
and

The last state is an accepting state.

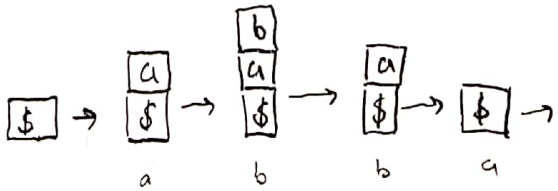
We don't care about the stack contents at the end of the accepting computation.

Example

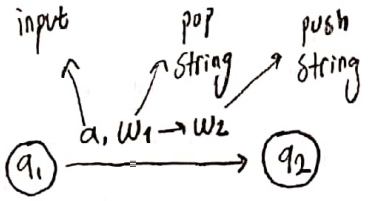
PDA M: $L(M) = \{VV^R : V \in \{a, b\}\}$



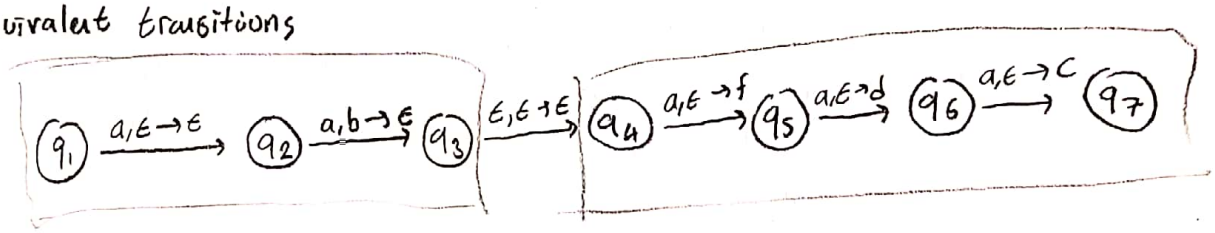
abba



Pushing & Popping Strings



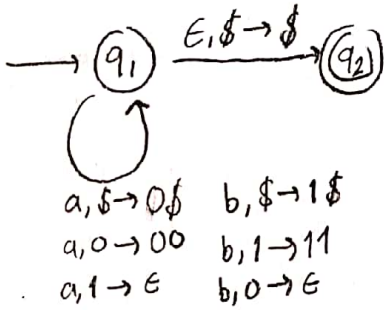
» Equivalent transitions

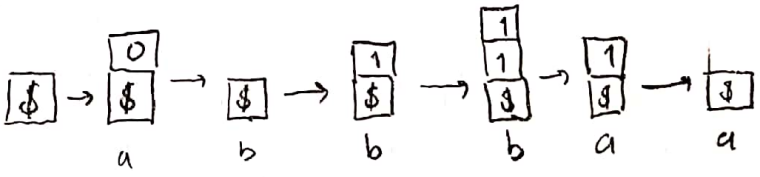


Example

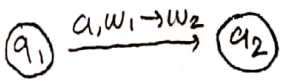
$$L(m) = \{ w \in \{a,b\}^* : n_a(w) = n_b(w) \}$$

input → abbbaa





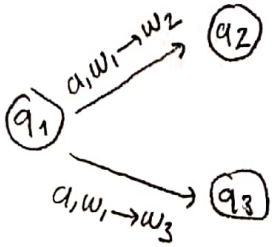
Formalities For PDAs

- 

» Transition function :

$$\delta(q_1, a, w_1) = \{(q_2, w_2)\}$$

(pop: w_1, push: w_2)

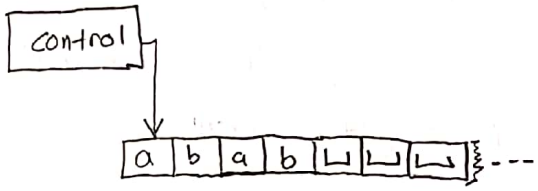
- 

» Transition function

$$\delta(q_1, a, w_1) = \{(q_2, w_2), (q_3, w_3)\}$$

Turing Machines

Turing machine has a tape head that can read and write symbols and move around on the tape. Initially the tape contains only the input string and blank everywhere else



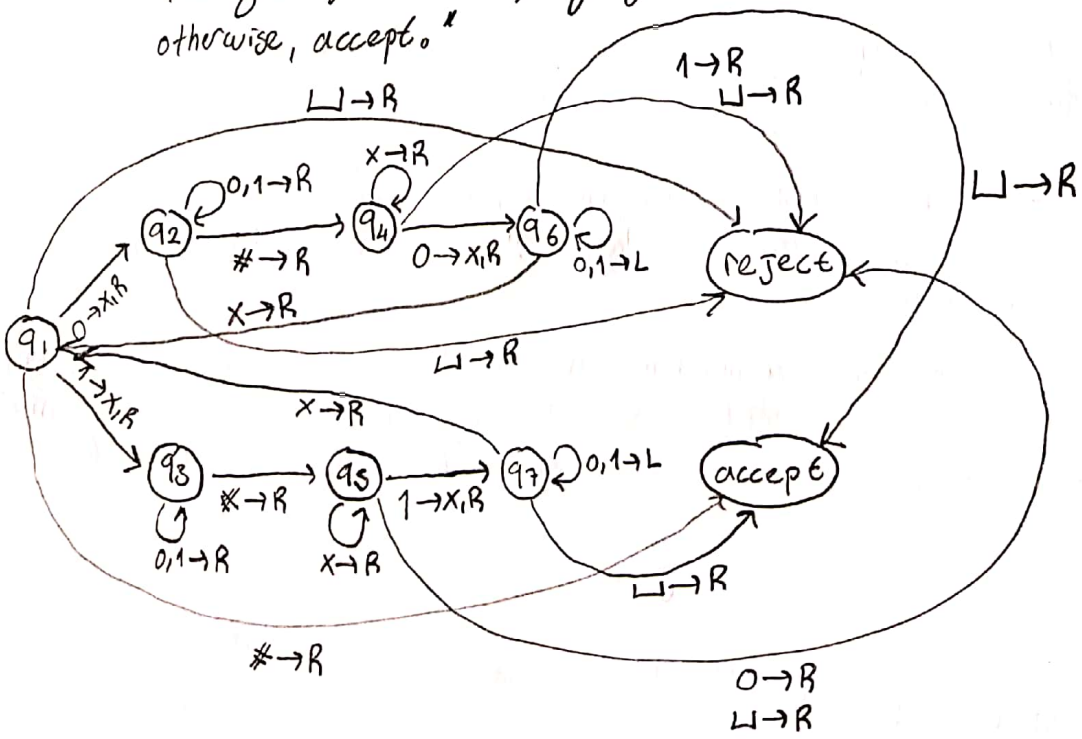
Example

$B = \{w\#w \mid w \in \{0,1\}^*\}$ is recognized by Turing machine M_1.

M_1 = " On the input string w:

1- Zig zag across tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.

2- When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbol remains, reject; otherwise, accept. "



Formal Definition of Turing Machine

$\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$

That is, when the machine is in a certain state q and the head is over a tape square containing a symbol a, and if $\delta(q,a) = (r,b,L)$, the machine writes b replacing a, and goes state r. The third component is either L or R that indicates whether the head moves to the left or right after writing.

- A Turing Machine is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$-
  - Q is the set of states.
  - Σ is the alphabet not containing the blank symbol $\sqcup$,
  - Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
  - $\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function.
  - $q_0 \in Q$ is the start state.
  - $q_{accept} \in Q$ is the accept state.
  - $q_{reject} \in Q$ is the reject state.

Definition

Call a language Turing-Recognizable if some turing machine recognizes it.

When we start a Turing Machine on an input, three outcomes are possible. The machine may accept, reject or loop. By loop, we mean that the machine simply does not halt.

Sometimes distinguishing a machine that is looping from one that is merely taking a long time is difficult. For this reason we prefer turing machines that halt on all inputs; such machines never loop. These machines are called deciders.

Definition

Call a language Turing-decidable if some Turing Machine decides it.

Note that

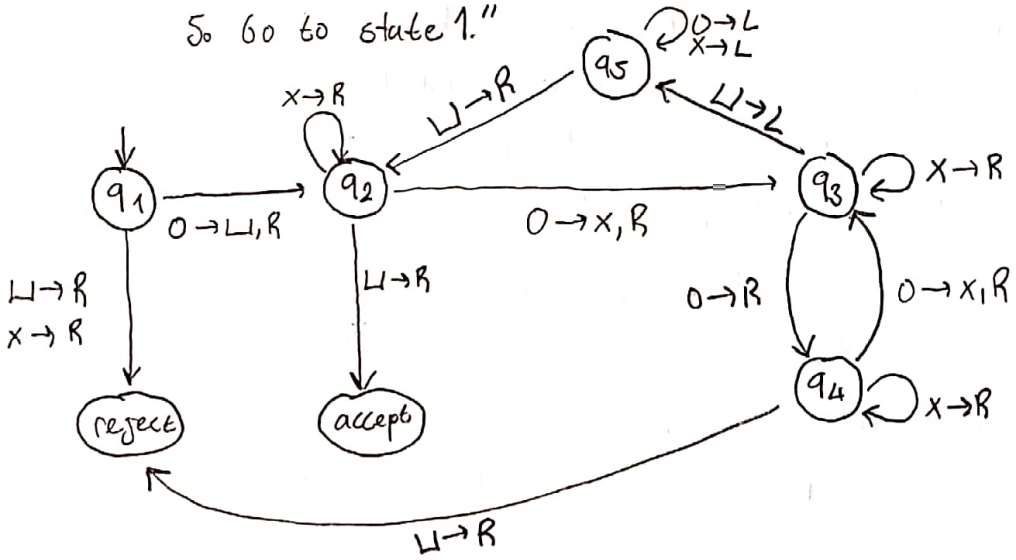
decidable $\subseteq$ recognizable

Example

M_2 decides $A = \{0^{2^n} \mid n \geq 0\}$

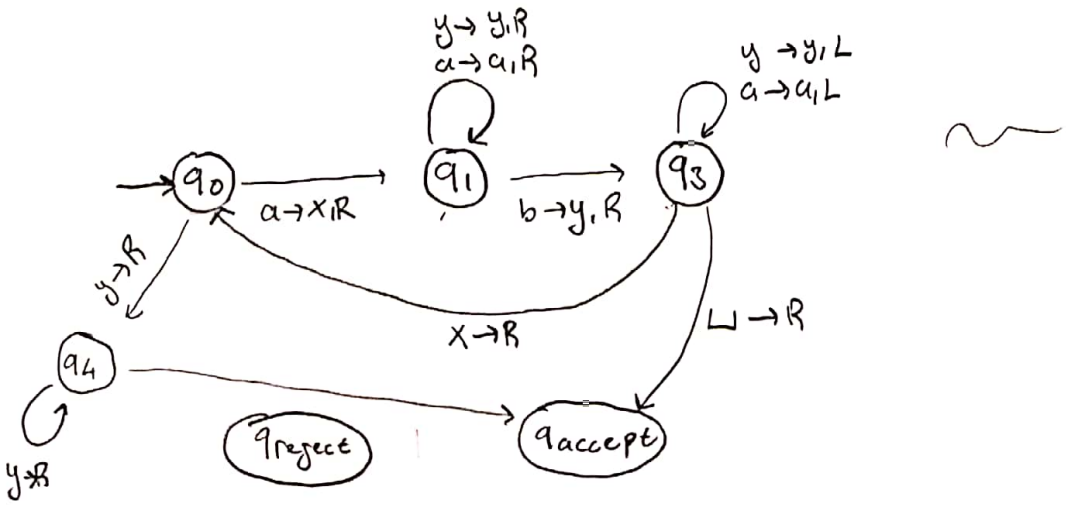
M_2 = " on the input string w:

1. Sweep left to right across tape, crossing off every other 0.
2. If in stage 1 the tape contained a single 0, accept.
3. If in stage 1 the tape contained more than a single 0 and the number of (0) is odd, reject.
4. Return the head to the left-hand end of the tape.
5. Go to stage 1."



Example

M_3 decides $A = \{a^n b^n \mid a, b \in \Sigma \text{ and } n \geq 1\}$



Example

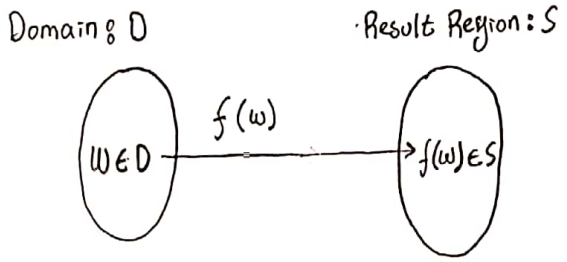
M_4 decides $A = \{a^i b^j c^k \mid i \times j = k \text{ and } i, j, k \geq 1\}$

M_4 = "On input string w:

1. Scan w from leftmost to rightmost to determine that whether $w \in a^+ b^+ c^+$ or not. Reject, if is not.
2. Return to the head, left most.
3. Cross of a 'a' and scan to the right until a 'b' occurs. Shuttle between the b's and the c's, crossing off one of each until all b's are gone. If all c's have been crossed off and some b's remain, reject.
4. Restore the crossed off b's and repeat stage 3 if there is another 'a' to cross off. If all a's have been crossed off, determine whether all c's also have been crossed off. If yes, accept otherwise reject."

Computing Functions with Turing Machines

A function $f(w)$ has



A function may have parameters: $f(x,y) = x+y$

integer domain:

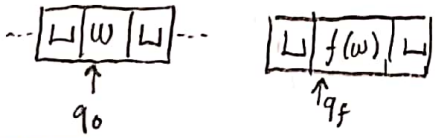
Decimal: 5

Binary: 101

Unary: 11111 , we prefer unary representation.

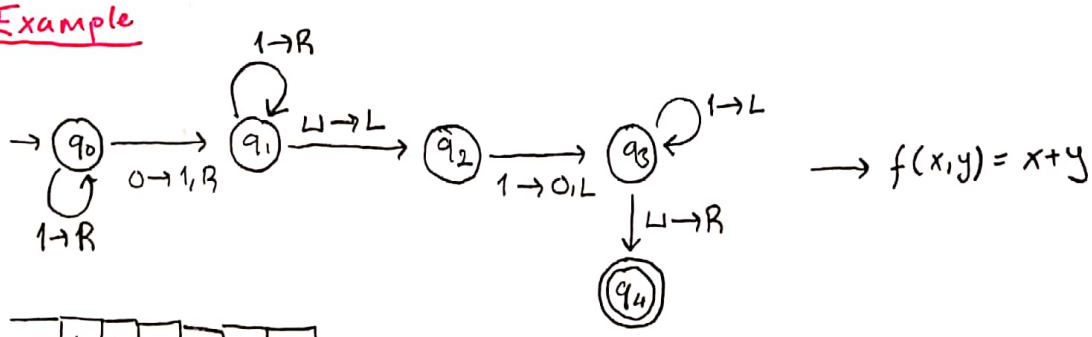
Definition

A function f is computable if there is a Turing Machine M such that:

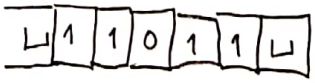


$\longrightarrow \quad q_0 w \overset{*}{\succ} q_f f(w)$

Example



$\longrightarrow f(x,y) = x+y$

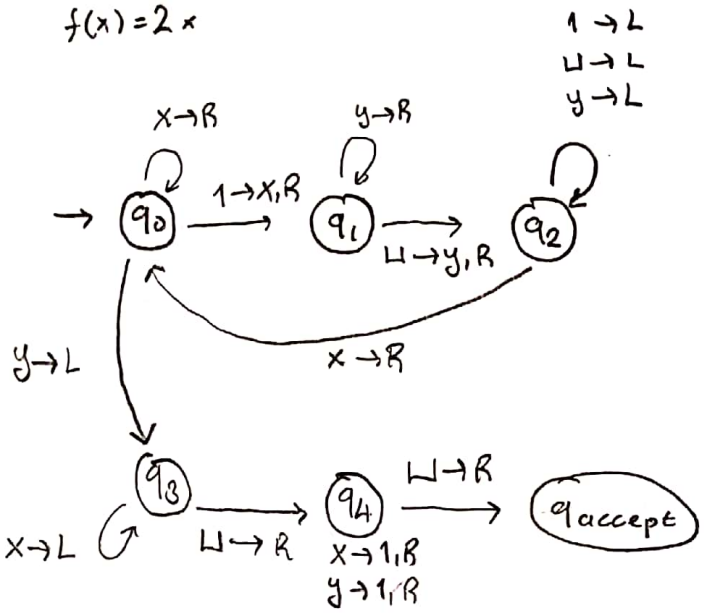


$x = 11$

$y = 11$

Example

$f(x) = 2x$



Example

$$f(x,y) = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{otherwise} \end{cases}$$

M = " for input string w

- Repeat

  Match a 1 from x with a 1 from y

  Until → all of x or y is matched

- If a 1 from x is not matched

  erase tape, write 1

  else

  erase tape, write 0 "

Turing's Thesis (1930):

Any computation carried out by mechanical means can be performed by a Turing Machine.

Definition

An algorithm for a problem is a Turing Machine which solves the problem.

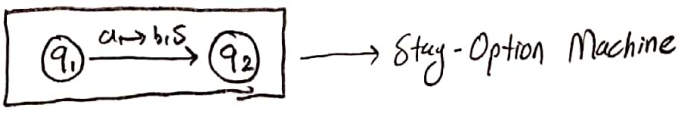
Variations of Turing Machine

Each variant of Turing Machine has the same power with Standart Turing Machine.
Same power of two machine means $L(M_1) = L(M_2)$

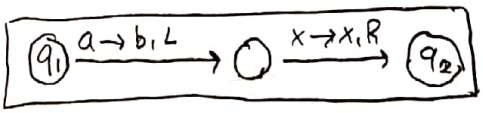
Turing Machines with Stay-Option

$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$

Possible head moves: Left, Right, Stay

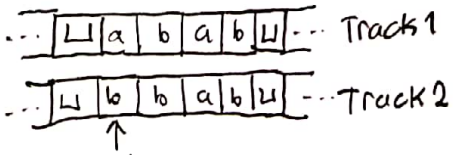


→ Stay-Option Machine



→ Simulation in Standart Machine

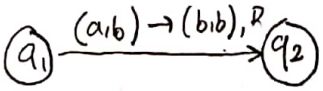
Multiple Track Turing Machines



head points symbol a and b together

one head
one symbol (a,b)

It is a standard turing machine since each tape alphabet describes a pair of two symbols.

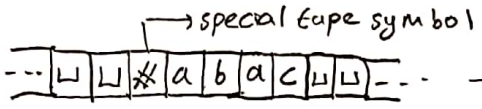


Theorem: Semi-infinite TMs have the same power as the standart TMs have.

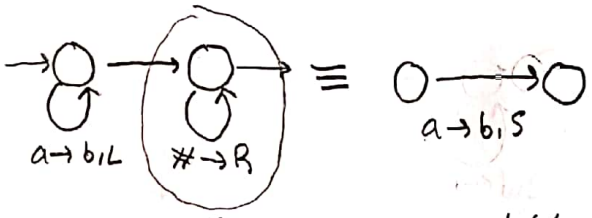
Proof:

1) Standart TMs simulate semi-infinite TMs

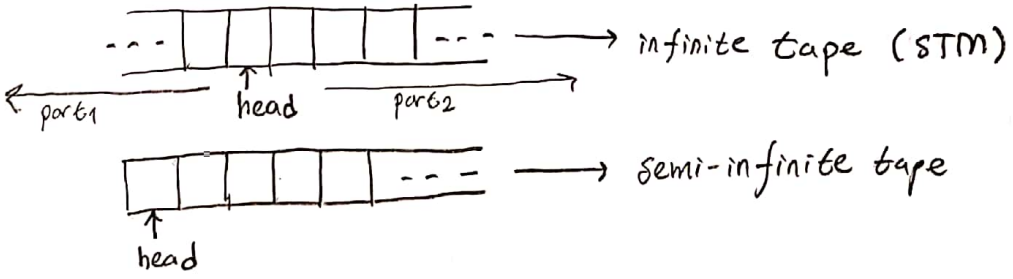
2) Semi-infinite machines simulates standart TMs.



Sharp represents the beginning of the tape.



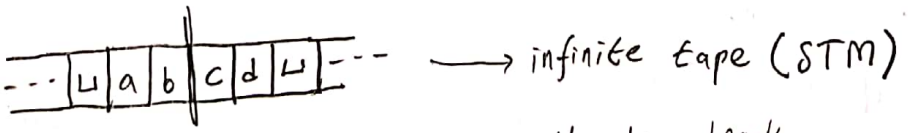
You cannot go to left of the sharp dve to this loop.
Sharp is the last point you can go at leftmost of the tape.



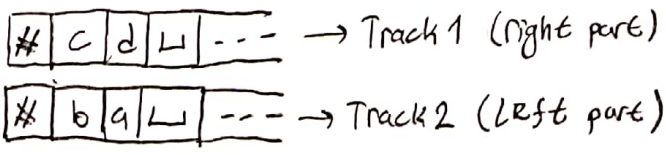
squeeze infinity of both direction to one direction.

part1 and part2 both are now semi-infinite tapes.

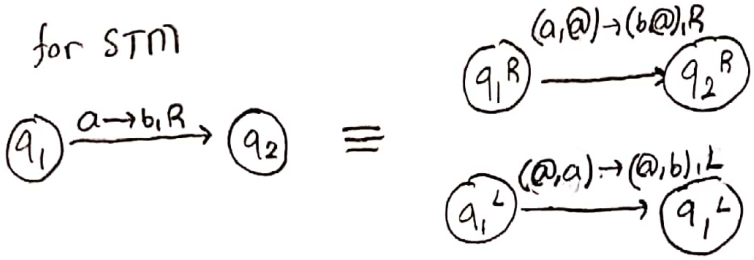
One infinite tape can be considered as the concatenation of two semi-infinite tape.



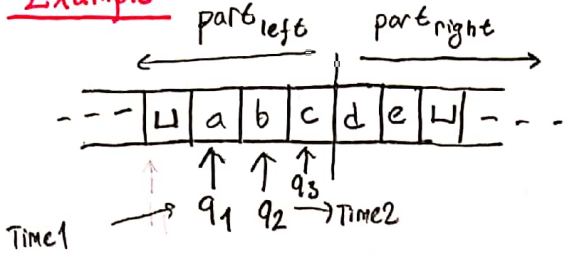
Consider it semi-infinite TM with two track

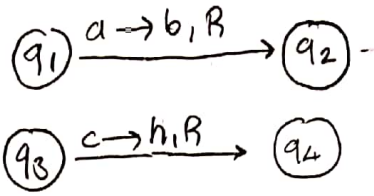


for STM

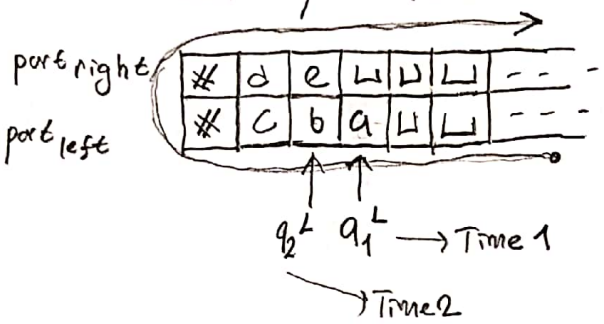


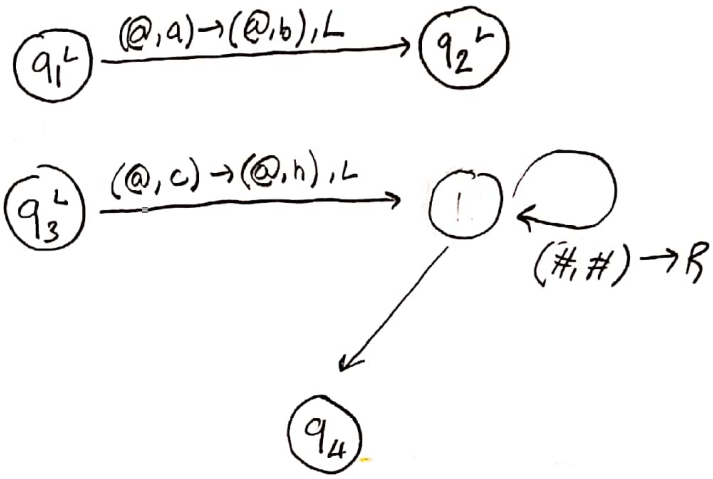
Example



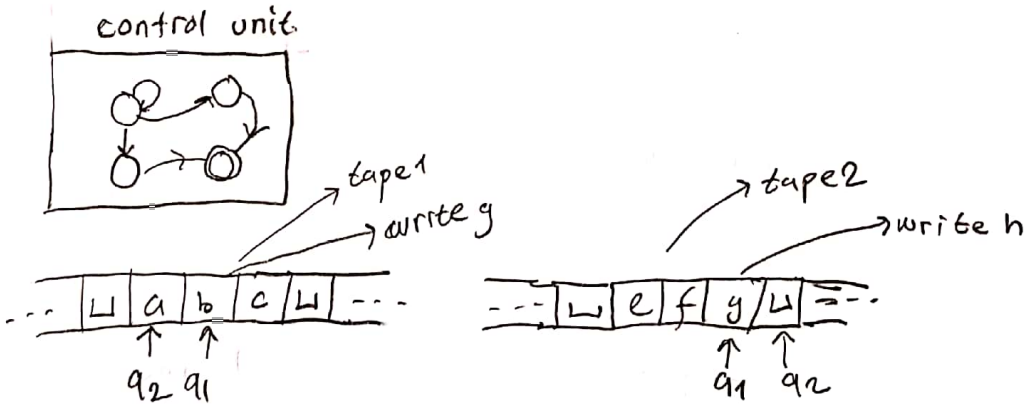


semi-infinite TM:

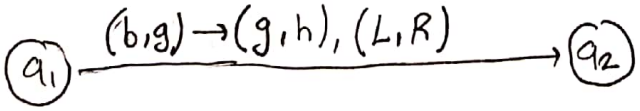




Multitape Tape Turing Machines



input string "abc" which is on tape1



Theorem: Multi-tape TMs have the same power with standart TMs.

Proof: 1) we have to show that multi-tape TMs simulate standart TMs.

2) Standart TMs simulate multi-tape TMs.

→ Use one tape. trivial.

→ In standart TMs we can use a multitrack tape to simulate multiple tapes.

- A tape of the multiple-tape TM corresponds to a pair of tracks.

Definition

If a language L is accepted by a Turing Machine M, then we say that the language is:

- Turing Recognizable
- Turing Acceptable
- Recursively Enumerable

Decidability

Decidable Languages

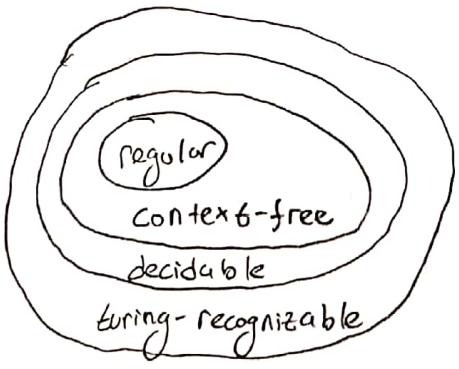
We focus on languages concerning automata and grammars.
For example, we present an algorithm that tests whether a string is a member of CFL. This problem testing whether a CFL generates a string is related to the problem of recognizing and compiling programs in a programming language

A language L is Turing Acceptable then there exists a Turing Machine M that accepts the language L.

$w \in L \Rightarrow M$ halts in a final state.

$w \notin L \Rightarrow M$ halts in a non-final state or loops forever.

A language L is decidable if there exists a Turing Machine (decider) M which accepts L and halts on every input string. Deciders never loop.
Also called recursive languages.

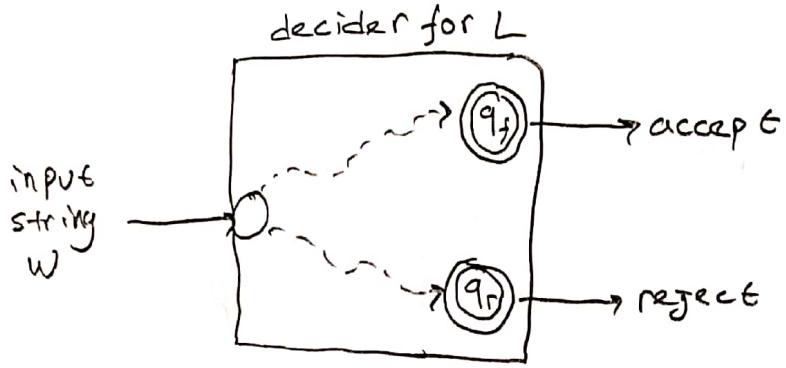


→ q_{accept} and q_{reject}

→ We can convert any TM to have single accept state and reject state.

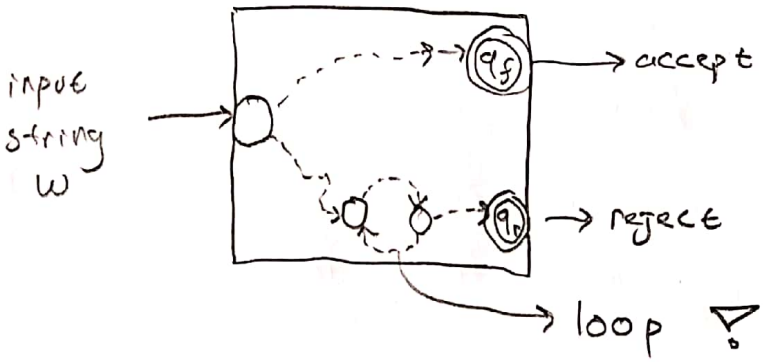
→ we can convert reject states to non-reject states, which means that we reduce the number of reject states to 1.

→ for a decidable language L



- Visualization- Difference Between Deciders and Turing-recognizable

→ for a Turing acceptable language L



Example

PRIMES $\{2, 3, 5, \ldots\}$

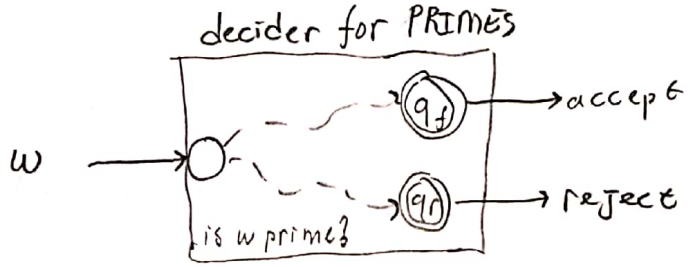
Decider for PRIMES

D_1 = "On input string w:

1- Divide w with all possible numbers between 2 and $\sqrt{w}$

2- If, any of them divides w, reject.

else, accept."



[redacted]

Decidable Problems Concerning Regular Languages

We give algorithms for testing whether a finite automaton is empty, and whether two finite are equivalent.

For example, the acceptance problem for DFAs of testing whether a particular DFA accepts a given string can be expressed as a language A_{DFA}.

$$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts } w.\}$$

Theorem: A_{DFA} is a decidable language.

Proof Idea: D = " On input $\langle B, w\rangle$

1- simulate B on w.

2- If simulation ends in an accept state, accept. If it ends in a nonaccepting state, reject."

Proof: Let's examine the input $\langle B, w\rangle$. Its representation of a DFA B together with a string w. One reasonable representation of B is simply a list of its five components; Q, Σ, δ, q_0 and F. When D receives its input, D first determines whether it properly represents a DFA B and a string w. If not, reject.

Then D carries out the simulation directly. It keeps track of B's current state and B's current position in the input w by writing this information on its tape. Initially, B's current state is q_0 and B's current input position is the leftmost symbol of w. The states and position are updated according to the specified transition function δ. When D finishes processing the last symbol of w, D accepts the input if B is in an accepting state; D reject the input if B is in a non-accepting state.

→ We can prove similar theorem for NFA. (Tip: convert. NFA B to DFAC)

$$A_{NFA} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts } w.\}$$

Example

Does DFA M accept the empty language $L(M) = \emptyset$

$EMPTY_{DFA} = \{$The set of DFAs that accept empty language $\emptyset\}$

$\langle M \rangle$ = Description of DFA M.

$\langle M \rangle$ = 1101110011101111 - - -

binary representation of DFA M

five-tuple definition.

$EMPTY_{DFA} = \{\langle M \rangle \mid M$ is a DFA that accepts empty language $\emptyset\}$

↳ is this language decidable?
is there a decider for language $EMPTY_{DFA}$?
is this problem resolvable?

D_2 = " for input $\langle M \rangle$

1- Determine whether there is a path from the initial state to any accepting state."

My algorithm
it is solvable.

Theorem: if language L is decidable then $\bar{L}$ is decidable as well.

Proof: We build a TM M' that accepts $\bar{L}$ and halts on every input string (M' is a decider for $\bar{L}$)

Assume that M is a deterministic machine and L is the language of this M. When we change the final and reject states to the reject and final states respectively, this new machine recognizes $\bar{L}$.

Turing Machine M' = " on each input string w,

1- Let M be the decider for L

2- Run M with w

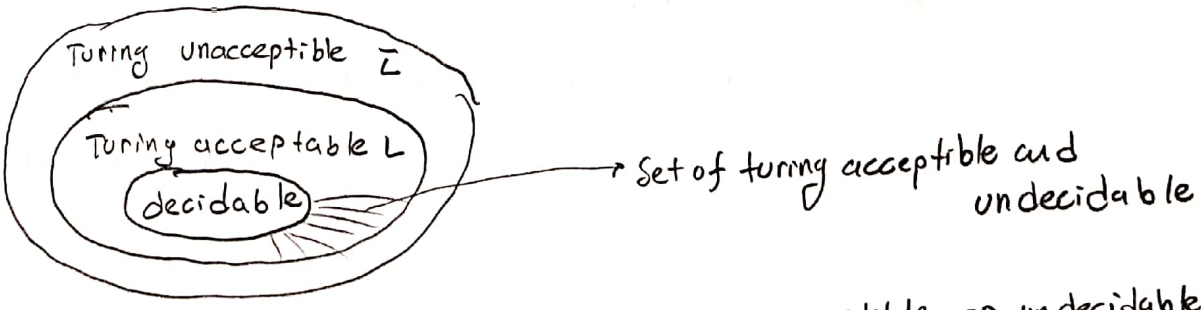
if M accepts then reject.

if M rejects then accept. //

Undecidable Problems

There are some undecidable languages. If the language L is undecidable, then we say that there is no decider for L. The machine may halt and decide for some input strings. For an undecidable language, the corresponding problem is undecidable.

There is no TM (algorithm) that yields an answer YES or NO for every input string.



L is the set of languages that can be either Turing acceptible or undecidable.

We will try to prove that two particular problems are unsolvable.

1 - Membership Problem

2 - Halting Problem

Theorem: $A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w.\}$, A_{TM} is undecidable.

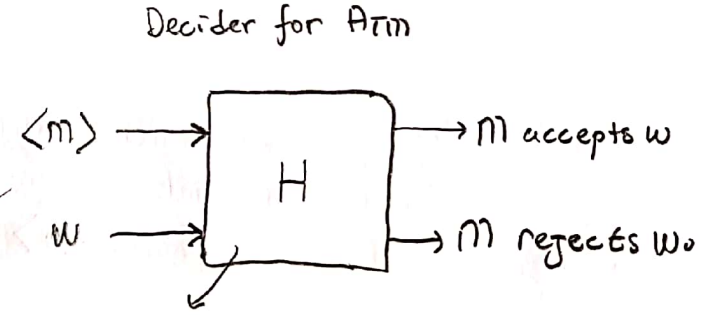
(Input: Turing Machine M, string w; $w \in L(M)$?)

Proof: We will assume that A_{TM} is decidable

⇓

We will then prove that every Turing-Acceptible languages also decidable (contradiction)

So, suppose that A_{TM} is decidable.



Let L be Turing-recognizable and M_2 be the TM that accepts L.

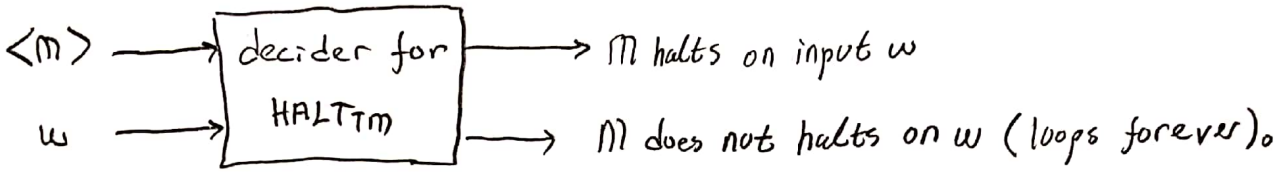
↳ We will prove that L is also decidable => we will build a decider for L.

Therefore, L is decidable since language L is chosen arbitrarily, every Turing Acceptible language is decidable. ⟶ However, we know that, there is a Turing-acceptable language which is undecidable.

So it is Turing-acceptable but not decider.

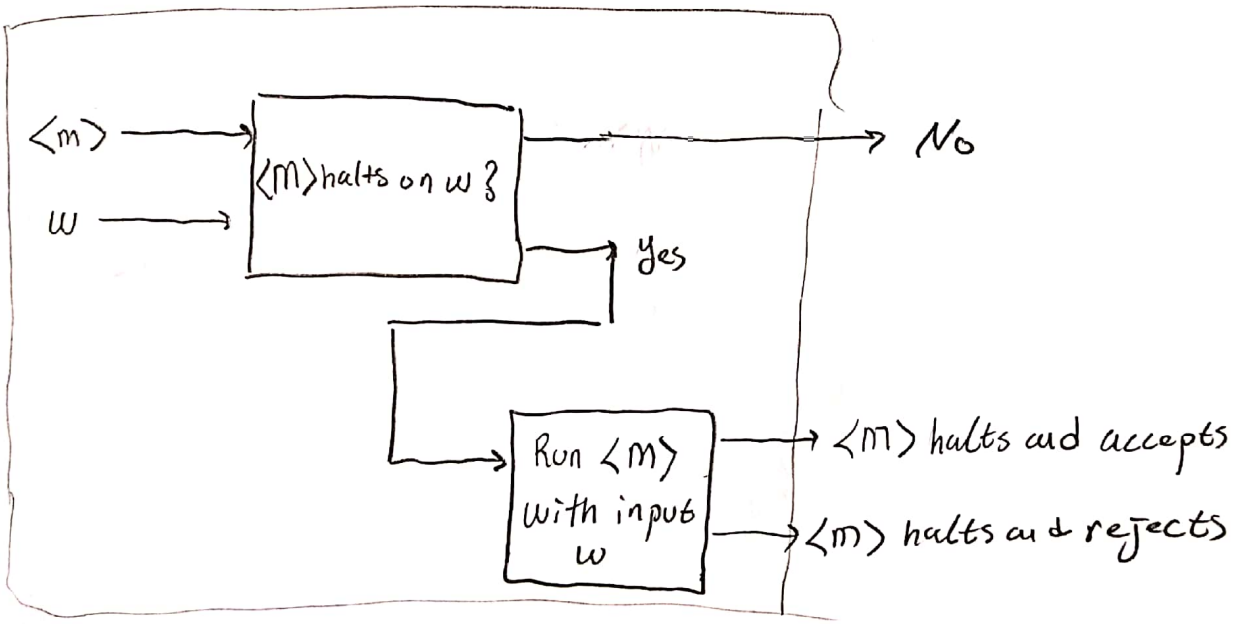
Theorem: $HALT_{TM} = \{\langle M, w\rangle \mid M \text{ halts on string } w\}$, $HALT_{TM}$ is undecidable.

Proof: Suppose that $HALT_{TM}$ is decidable. Then, we will prove that every Turing-acceptable language is also decidable.



Let L be a Turing-acceptible language, M_L be the Turing Machine that accepts L.

↳ We will prove that L is decidable.



↳ This TM never loops forever
Thus, it is a decider for L
Then, L is decidable

Since L is chosen arbitrarily, every Turing-acceptable language is decidable. However there is a Turing-acceptable language which is undecidable.