CS259: Formal Languages

Alphabet: a non-empty finite set of symbols

$\Sigma_1 = \{a, b, c\}$ $\Sigma_2 = \{5, 8, 10\}$

Language: a (potentially infinite) set of finite strings over an alphabet.

$L_1 = \{ab, abc, aaab, ccc, ba\}$

$L_2 = \{a, aa, aaa, \ldots\}$

$\Sigma^* = \{$all finite strings (also called words) over the alphabet $\Sigma\}$

Every decision problem can be phrased in the form "is a given string present in the language L?"

for example

Does a given Java program have any syntax errors

Is the given string of symbols that comprise the input java program, present in the language $L = \{$all syntatically correct Java programs$\}$

Does the given graph have an indepent set of size 100?

Is the binary encoding of the given graph present in the language $L = \{$ all binary encoding graphs with an independent set of size 100$\}$

Finite set of states
- Q

Finite set called the ALPHABET
- Σ

In which state to start computation
- $q_0 \in Q$

What are the FINAL/ ACCEPTING states?
- $F \subseteq Q$

How to TRANSITION from one state to another?
- $\delta : Q \times \Sigma \to Q$

A machine M defined by the tuple $M = (Q, \Sigma, q_0, F, \delta)$ is called a

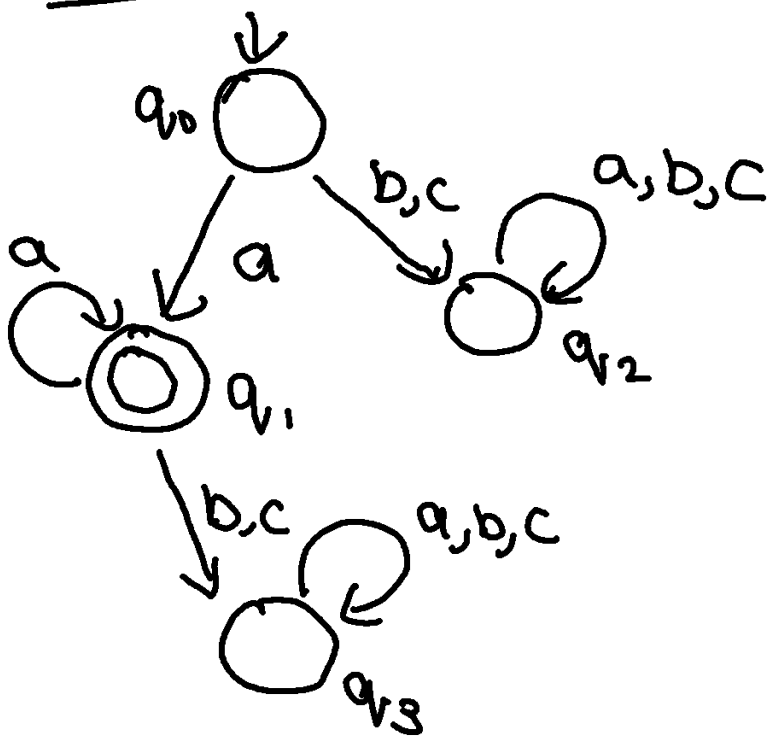
(Deterministic) Finite State Automation

or

(Deterministic) Finite State Machine

Representing DFA

State Diagram



State Transition Table

δ	a	b	c
q_0	q_1	q_2	q_2
q_1	q_1	q_3	q_3
q_2	q_2	q_2	q_2
q_3	q_3	q_3	q_3

Both are equivalent.

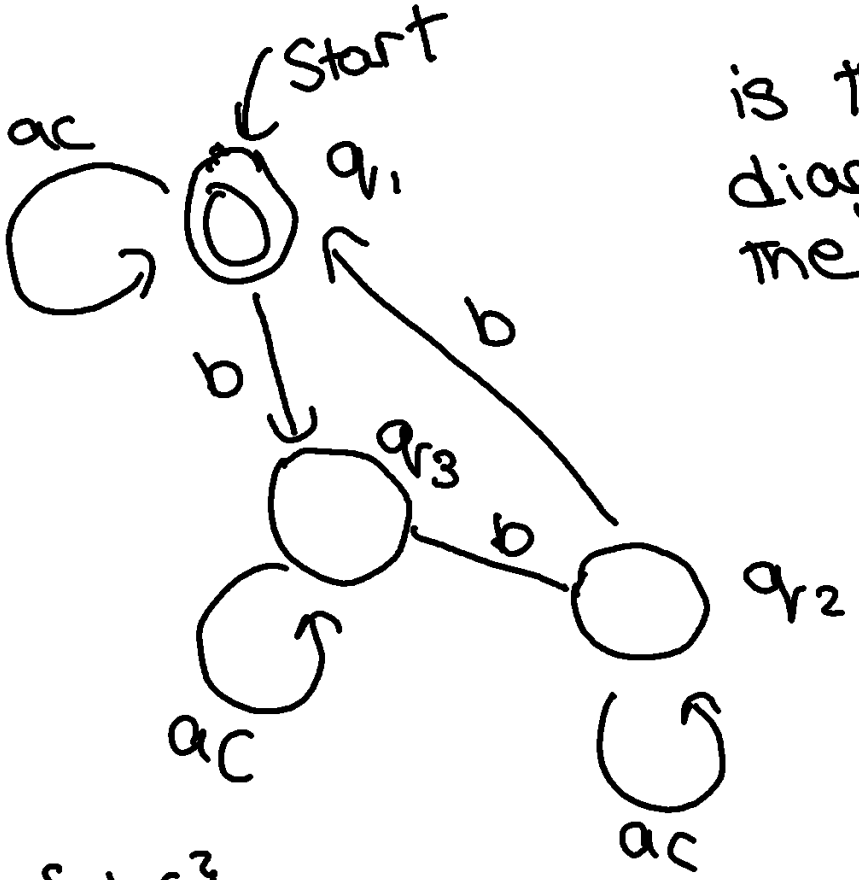
Input	"Output"
a	Accept
aa	Accept
abc abc	Reject
bca bc	Reject
ε	Reject

Language $L = \{a, aa, aaa, \ldots\}$

ε is the empty string, is in all States. (I think)

$\Sigma = \{a, b, c\}$

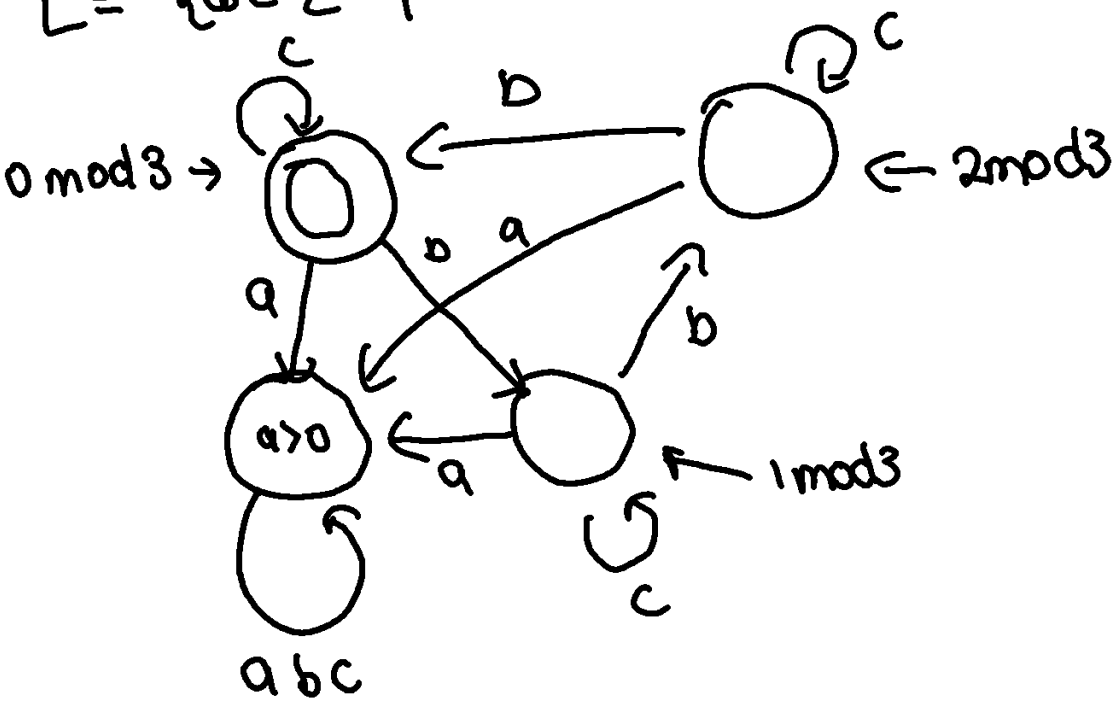
$L = \{w \in \Sigma^* \mid \#\ b\text{'s in } w \text{ is divisible by } 3\}$



is the state diagram for the DFA

$\Sigma = \{a, b, c\}$

$L = \{w \in \Sigma^* \mid \#\ b\text{'s divisible by } 3,\ \#\ a\text{'s} = 0\}$



Examples of whether DFA or not are in the slides (Lecture 2).

- need start state
- don't need final state
- need result of every symbol from every state

The Empty Word (ε)

$|\varepsilon| = 0$, length is 0

$L_1 = \{\}$ is the empty language

$L_2 = \{\varepsilon\}$ is a non-empty language

Σ^* always contains ε

Monoids

- Comprise of a set, an associative binary operation on the set with an identity element.
- $(\mathbb{N}_0, +, 0)$
- $(\mathbb{N}, \times, 1)$
- $(\Sigma^*, \circ, \varepsilon)$ ← string concatenation

are all examples of monoids

Definition The extended transition function of a DFA expresses the change in state upon reading a string. ($\hat{\delta}$)

Formally, the extended transition function $\hat{\delta}$ is a function st

$\hat{\delta}: Q \times \Sigma^* \to Q$

- $\forall q \in Q, \hat{\delta}(q, \varepsilon) = q$
- $\forall q \in Q, s \in \Sigma^*$ st $s = wa$ for some $w \in \Sigma^*, a \in \Sigma$

$$\hat{\delta}(q, s) = \delta(\hat{\delta}(q, w), a)$$

Language accepted by DFA

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

The language accepted or recognised by M is denoted by $L(M)$ and is defined as

$$L(M) = \{ s \in \Sigma^* \mid \hat{\delta}(q_0, s) \in F \}$$

Run of a DFA

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

String $s = s_1 s_2 \dots s_n$

$s_i \in \Sigma \quad \forall i \in [n]$ $\qquad [n] = \{1, \dots, n\}$

- The run of M on the empty word ε is just q_0.
- The run of M on the word s is a sequence of states $r_1, \ldots, r_n$, where:
  1. $r_0 = q_0$
  2. $\forall i \in [n]$, $r_i = \delta(r_{i-1}, s_i)$

- The run of M on a word s is called an accepting run if the last state in the run is an accepting run of M.
- A word s is said to be accepted by M if the run of M on w is an accepted run. That is,

$$L(M) = \{s \in \Sigma^* \mid \text{the run of } M \text{ on } s \text{ is an accepting run}\}$$

Definition: A language L is called regular if it is accepted by some Deterministic Finite State Automation (DFA).

$\{\}, \Sigma^*$ are regular

Is every language regular?

Σ^* is a countably infinite set, $\because \Sigma$ is finite.

no. of languages $P(\Sigma^*)$

by Cantors theorem, no surjection

$\therefore$ uncountable # of languages

Each dfa is depicted by directed graph which can be encoded into a bit strings

bitstrings are countable $\therefore$ There are a countable number of DFAs.

So not every language is regular.

Operations on Languages

- Languages are sets of strings, $\therefore$ any set operations can be applied to them. (plus more string specific operations)

for example,

Revesal - reverse x $\forall x \in L$ (Unary)

Truncate - $x[1:]$ $\forall x \in L$

Complement

Concatenation $\{x_i y_j \in \Sigma^* \mid x_i \in L_1, y_i \in L_2\}$

Union (Binary)

Intersection

Concat $L_1 = \{3$ with $L_2, L_3 = \{\}$

Are regular languages Closed under complement?

Yes! If $M = (Q, \Sigma, q_0, F, \delta)$ is a DFA for L, then $M' = (Q, \Sigma, q_0, Q \setminus F, \delta)$ is a DFA for L^c.

Are regular languages closed under intersections?

Yes? given M_1 as a DFA for L_1
and M_2 as a DFA for L_2

$\forall x \in L_1 \cap L_2$ M_1 accepts x as $x \in L_1$
M_2 accepts x as $x \in L_2$

if $x \in L_1 \cap L_2^c$ M_2 doesn't accept
$x \in L_1^c \cap L_2$ M_1 doesn't accept
$x \in L_1 \cap L_2^c$ both don't accepts

$\therefore L_1 \cap L_2$ can be checked using a program

Can we build a DFA M_3 st the run of M_3 on any string simulateeasly Simulates the run of M_1 and M_2 on the same string.

$M_1 = (Q_1, \Sigma, q_1, F_1, \delta_1)$ and $M_2 = (Q_2, \Sigma, q_2, F_2, \delta_2)$

$M_3 = (Q_1 \times Q_2, \Sigma, (q_1, q_2), F_1 \times F_2, \delta)$ where

$\delta((x,y), a) = (\delta_1(x,a) - \delta_2(y,a))$

$\forall a \in \Sigma \quad x \in Q_1 \quad y \in Q_2$

$L(M_3) = L(M_2) \cap L(M_1)$

Therefore, regular languages are closed under intersection.

What if M_1 and M_2 have different Σ's

Then $\Sigma = \Sigma_1 \cap \Sigma_2$

What about Union?

$F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$

different Σ's ?

$\Sigma = \Sigma_1 \cap \Sigma_2$

$\delta_1(x, a) = q_d \quad \forall a \notin \Sigma_1$ where

$\delta_2(y, a) = q_d \quad \forall a \notin \Sigma_2$ q_d is a dead state.

Regular languages are closed under union?

If L_1 and L_2 are regular, are L_1 / L_2 also regular?

$L_1 / L_2 = L_1 \cap L_2^c$, as regular languages are closed under complements/intersetions

If L_1 and $L_1 \| L_2$ are regular, Then L_2 is also regular?

False, Counter Example

$L_1 = \emptyset$ (regular)

L_2 = any non-regular language

$L_1 \| L_2 = \emptyset$ (regular)

Are regular languages closed under taking subsets?

$L = \Sigma^*$ (all langs are subsets of this)

L' non-regular

$|L'| \leq L \quad \therefore L' \subseteq L$

Regular Language closed uncler infinite union?

no, non-regular exist, express Language as union of Singleton languages (regular)

Regular Language Closed under string reversal?

Non-Determinitic Finite State Automata (NFA)

- Finite set of States — Q
- Finite Alphabet — Σ
- Initial State — $q_0 \in Q$
- Accept States — $F \subseteq Q$
- Transition function — $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$

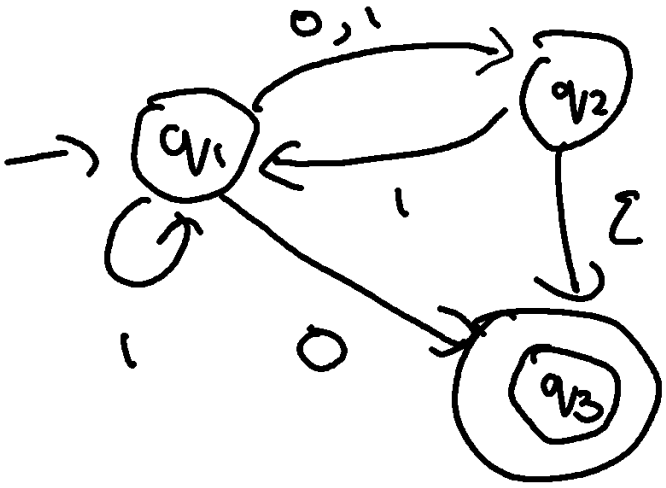
Difference

DFA has 1 transition out of State $\forall sym \in \Sigma$

NFA can have none or multible transitions out of a state on reading same symbol.

No ε-transistion

has ε-transition



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

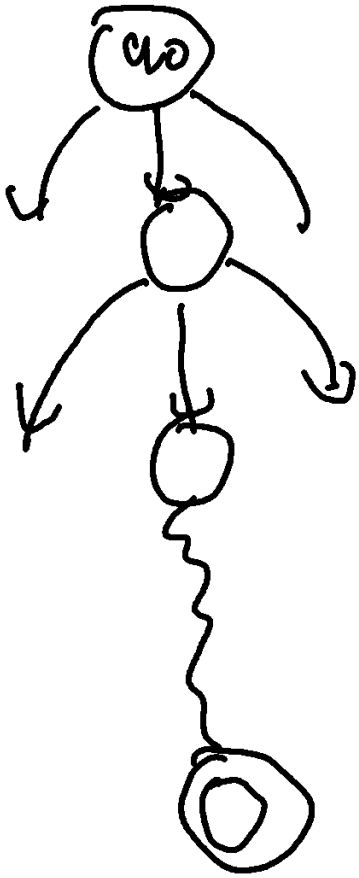
$\delta(q_1, 1) = \{q_1, q_2\}$

$\delta(\{q_3, 0\}) = \{\}$

DFA has exactly 1 possible run on a word

NFA can have many or no runs on a word

There is a unique run of a DFA on Input.



Each root-to-leaf in the computation tree is a run.

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

- A run of M on the words S is a sequence of states $r_0, r_1, \ldots, r_n$ st

1. $r_0 = q_0$
2. $\exists s_1, s_2, \ldots, s_n \in \Sigma \cup \{\varepsilon\}$ st $S = s_1, s_2, \ldots, s_n$ and $\forall i \in [n]$: $r_i \in \delta(r_{i-1}, s_i)$

- A run of M on a word S is called an accepting run if the last state in this run is an accepting state of M.
- A word S is said to be accepted by M if some run of M on w is an accepting run. Then the language accepted by the machine M is

$L(M) = \{S \in \Sigma^* \mid$ some run of M is accepting $\}$

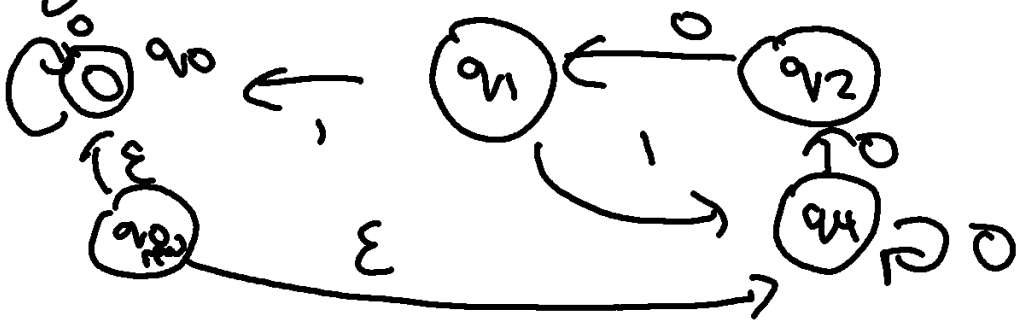
If I have regular language, do I have a NFA for the reversal of the language?

Yes, did an example in lecture. not convinced that it works for all regular Languages.

State transition table

δ	0	1	ε
q_{0new}	$\emptyset$	$\emptyset$	$\{q_4, q_2\}$
q_0	$\{q_0\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\{q_0, q_4\}$	$\emptyset$
q_2	$\{q_2\}$	$\emptyset$	$\emptyset$
q_3	q_3	q_3	$\emptyset$
q_4	$\{q_2, q_4\}$	$\emptyset$	$\emptyset$

State transition diagram

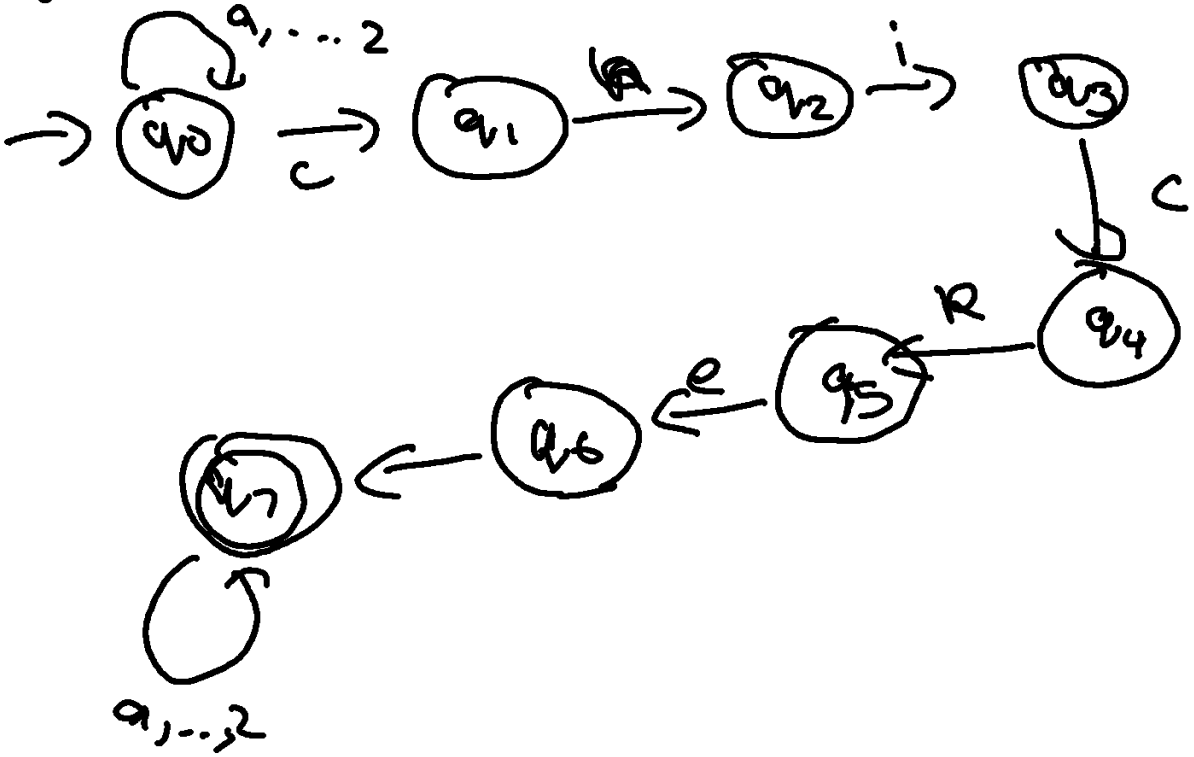


DFA can accept ε even without ε-transition. If initial is final.

NFA

alphabet {a, b, c, ... z}

words that contain chicken



Extended Transition Function (NFA)

$\hat{\delta}: Q \times \Sigma^* \to \mathcal{P}(Q)$ $\quad \forall q \in Q, w \in \Sigma^*$

$\hat{\delta}(q, w)$ = all states in Q to which there is a run from q upon reading the string w.

Examples

ε - Closure

$ECLOSE: Q \to \mathcal{P}(Q)$

Informally: ECLOSE(q) denotes all states that can be reached from q by following ε- transitions alone.

Formally ECLOSE(q) is the smallest set st

- $\forall q \in Q$, $q \in ECLOSE(q)$
- $\forall p, q, r \in Q$ if $p \in ECLOSE(q)$ and $r \in \delta(p, \varepsilon)$ $\Rightarrow r \in ECLOSE(q)$

ECLOSE can be extended naturally to set of states in the natural way.

$$\forall X \subseteq Q \quad ECLOSE(X) = \bigcup_{x \in X} ECLOSE(x)$$

$$ECLOSE(\emptyset) = \emptyset$$

Inductive Definition of Extended Transition Function

Formally, the extended transition function $\hat{\delta}$ for an NFA $(Q, \Sigma, q_0, F, \delta)$ is a function $\hat{\delta}: Q \times \Sigma^* \to \mathcal{P}(Q)$ and is defined as

- $\forall q \in Q \quad \hat{\delta}(q, \varepsilon) = ECLOSE(q)$
- $\forall q \in Q$ and word $s \in \Sigma^*$ st $s = wa$ for some $w \in \Sigma^*$ and $a \in \Sigma$,
  $\hat{\delta}(q, s) = ECLOSE\left(\bigcup_{q' \in \hat{\delta}(q,w)} \delta(q', a)\right)$.

Consider an NFA $M = (Q, \Sigma, q_0, F, \delta)$. The language accepted or recognised by M is denoted by

$$L(M) = \{ s \in \Sigma^* \mid \hat{\delta}(q_0, s) \cap F \neq \emptyset \}$$

NFAs vs DFAs

How much more powerful are NFAs than DFAs?

Not that much! Any NFA can be simulated by a DFA with potentially more (but still finite number of) states.

Main Idea:
Design a DFA whose extended transition function mimics that of a NFA.

Let $N = (Q, \Sigma, q_0, F, \delta)$ be the NFA we want to determine.
Denote using $M = (Q_1, \Sigma, q_1, F_1, \delta_1)$ the target DFA.

$Q_1 = 2^Q$

$q_1 = \text{ECLOSE}(q_0)$

$F_1 = \{X \subseteq Q \mid X \cap F \neq \emptyset\}$

$$\delta_1(X, a) = \bigcup_{y \in X} \text{ECLOSE}(\delta(y, a))$$

$$= \{z \mid \text{for some } y \in X, z \in \text{ECLOSE}(\delta(y, a))\}$$

$$\hat{\delta}(q_0, \varepsilon) = \hat{\delta}_1(q_1, \varepsilon)$$

$$= \text{ECLOSE}(q_0)$$

NFA-DFA equivalence

The following statements are equivalent

- Language L is regular
- L is accepted by some DFA
- L is accepted by some NFA

Regular Expressions

Give a syntax for pattern of strings

$R = a$ for some $a \in \Sigma^*$ $L(R) = \{a\}$

$R = \varepsilon$ $L(R) = \{\varepsilon\}$

$R = \emptyset$ $L(R) = \emptyset$

$R = R_1 + R_2$ (also written as $R_1 \cup R_2$)

$L(R) = L(R_1) \cup L(R_2)$

$R = R_1 . R_2$ $L(R) = L(R_1) . L(R_2)$

$R = R_1^*$ $L(R) = (L(R_1))^*$

$$S^* = \{\varepsilon\} \cup S \cup S^2 \cup S^3 \cup \ldots \text{ (definition)}$$

Operator Procedence

*	Highest
concatenation	
+	Lowest

Examples $\Sigma = \{a,b\}$ $R = (a+b)^*$

whats $L(R)$? $= \Sigma^*$

$(a+b)^* = (\{a\} \cup \{b\})^* = \Sigma^*$

$R = (a+b^*)(a+bb)$

$L(R) = \{\omega \in \Sigma^* \mid \omega$ ends in 'a' or 'bb'$\}$

$R = (aa)^* (bb)^* b$

$L(R) = \{\omega \in \Sigma^* \mid$ even # a's, odd # b's$\}$

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

1) no b's occour

$R = a^*$

2) atleast 1 b

$R = (a+b)^* b (a+b)^*$

3) every b followed by an a

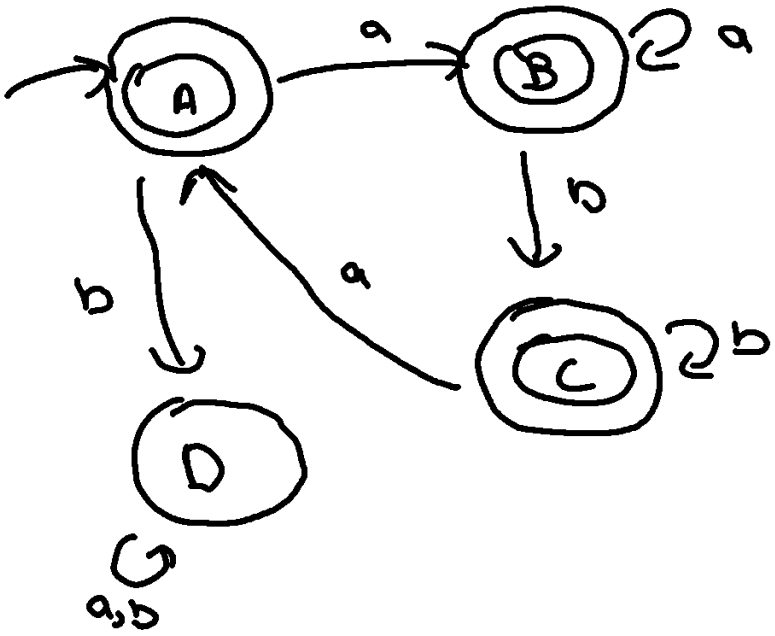
$R = (a + ba)^*$

4) last symbol is b

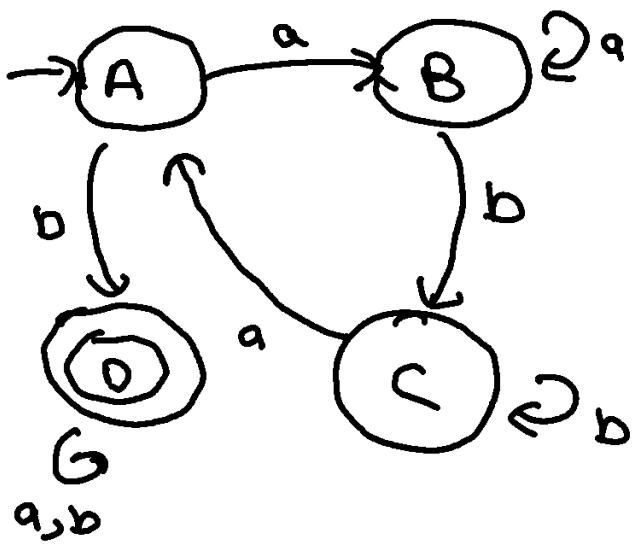
$R = (a+b)^* a$

5) all cons. blocks of a have even length

$R = (b + aa)^*$



M

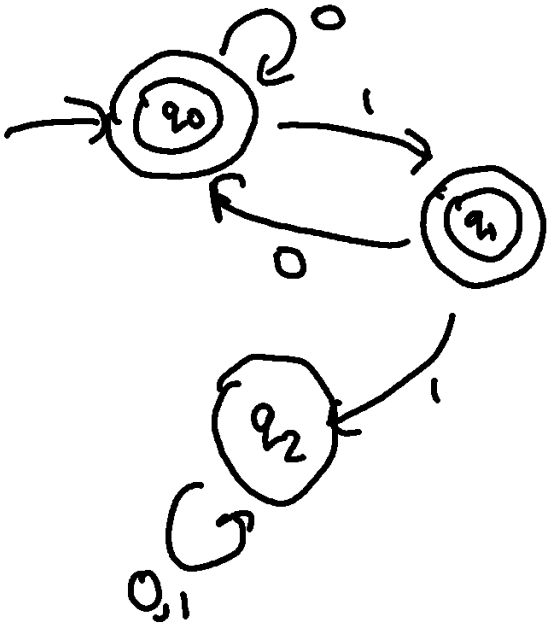


M'

$L(M) = \overline{L(M')}$

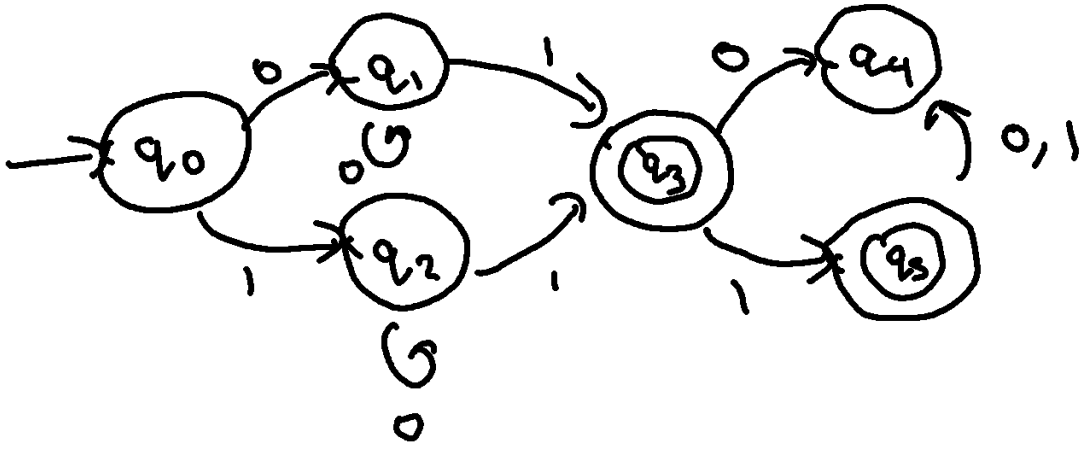
$R = (aa^*bb^*a)^* b(a+b^*)$

$L(M^*) = L(R)$



Regular expression generated by language accepted by this NFA.

$(0 + 1.0)^* . (\varepsilon + 1)$



$(0+1)0^*1(\varepsilon+1)$

A language is accepted by a NFA/DFA iff it is generated by a regular expresion.

Regex => NFA/DFA

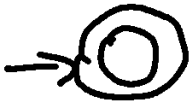
Idea: Define NFA for each atomic part then describe how to

a) Combine NFA's for R_1 and R_2 to get an NFA for R where $R = R_1 \oplus R_2$ and $\oplus \in \{+, .\}$

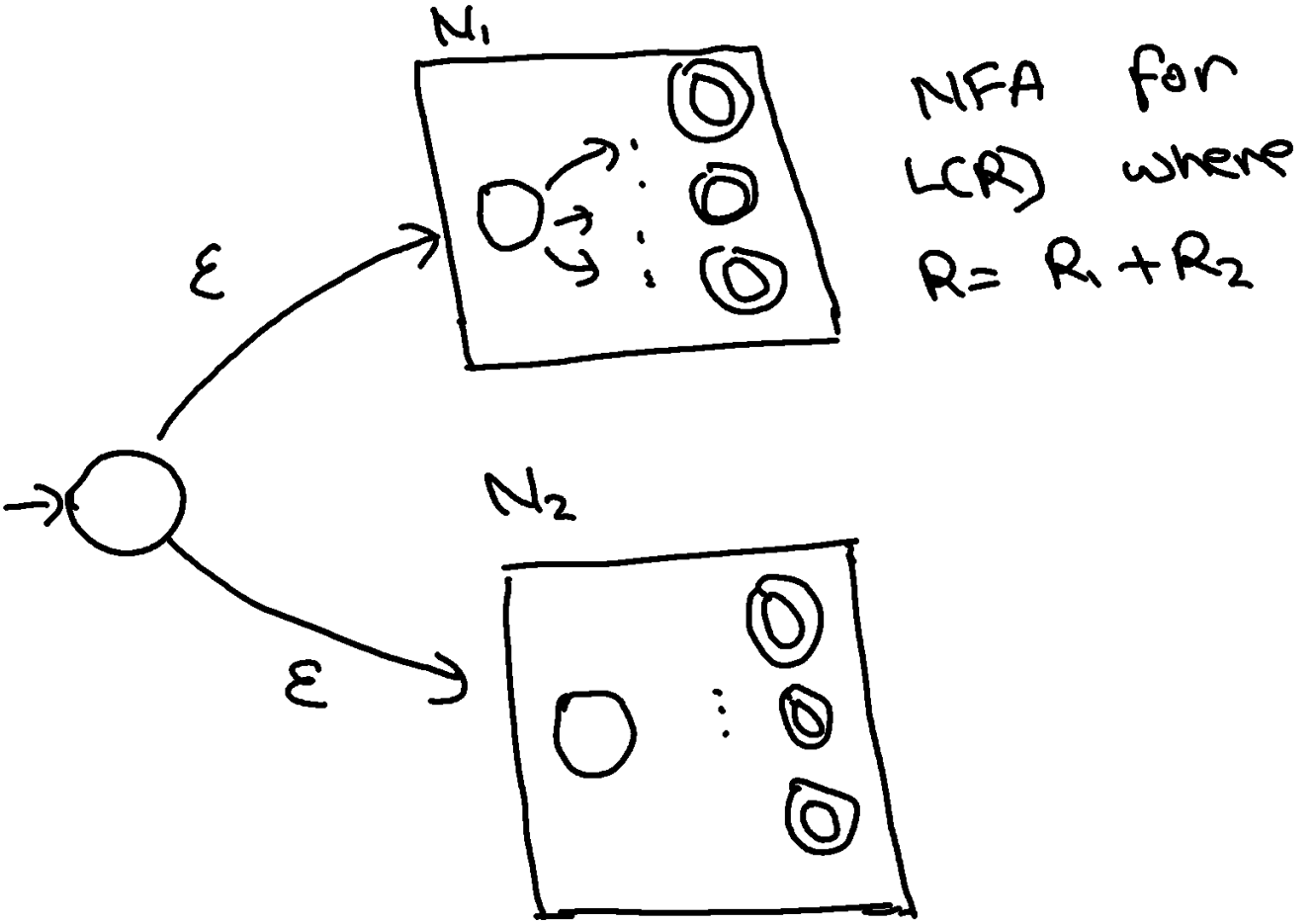
b) Use an NFA for R_1 to get one for $R = R_1^*$

$L(R) = \{a\}$ if $R = a$ $(a \in \Sigma)$

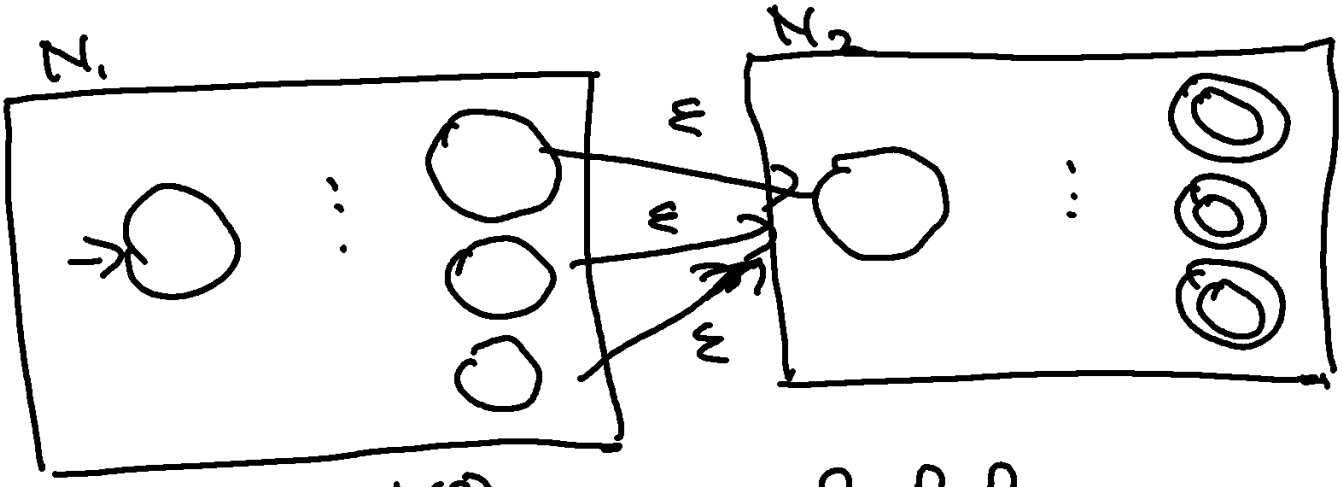
$L(R) = \{\varepsilon\}$ if $R = \varepsilon$



$L(R) = \emptyset$ if $R = \emptyset$



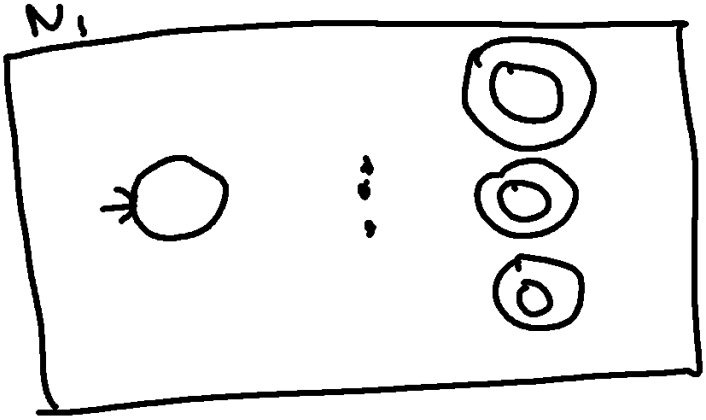
NFA for $L(R)$ where $R = R_1 + R_2$

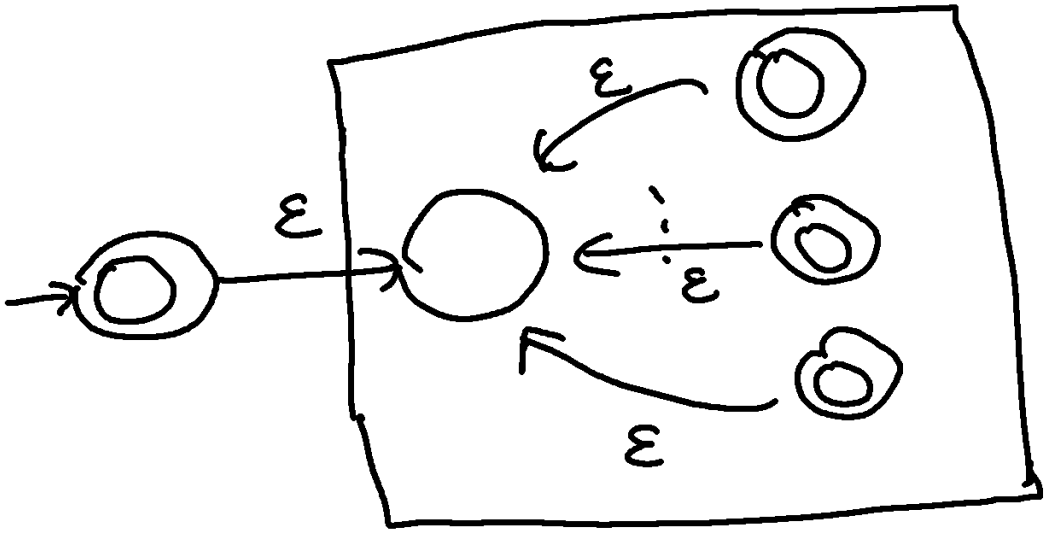


NFA for $L(R)$ where $R = R_1 . R_2$

N_1 is an NFA for $L(R_1)$

N_1



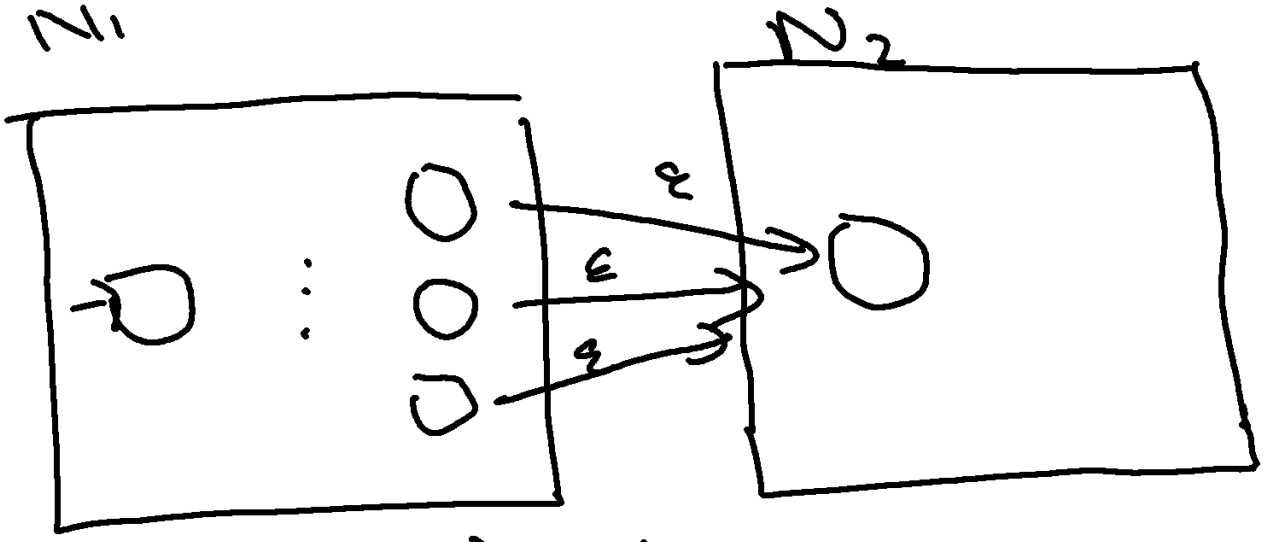


Note 1: We need to ensure that the start state of the new NFA is also an accept state because $L(R_1^*)$ contains the empty string while $L(R_1)$ itself might not.

Note 2: We needed to add a new start state because simply keeping the start state of N and making it a final state can result in non-accepting runs of N becoming accepting runs, which then renders the eventual construction incorrect.

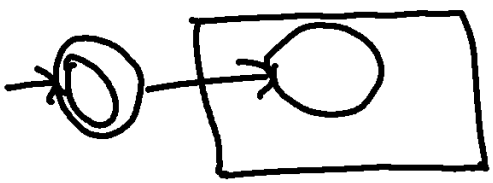
Questions?

1) If r is some regex, what is $L(r \cdot \emptyset)$?



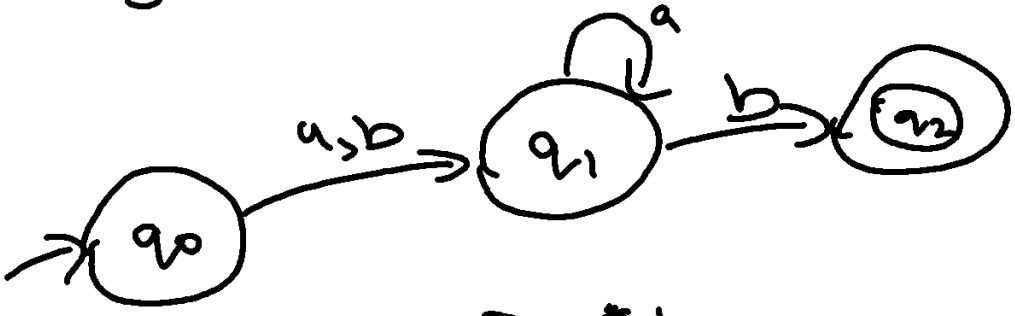
So, $L(r \cdot \emptyset) = \emptyset$

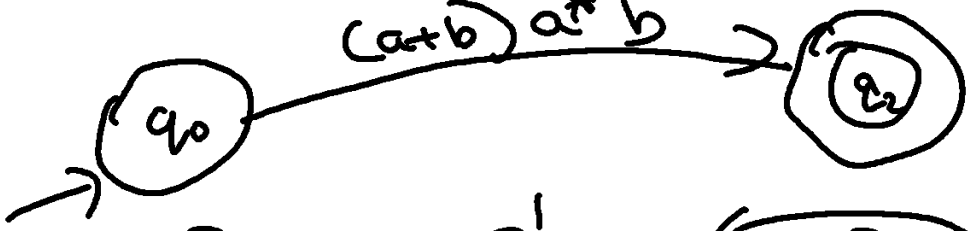
2) what is $L(\emptyset^*)$? $= L(\varepsilon)$

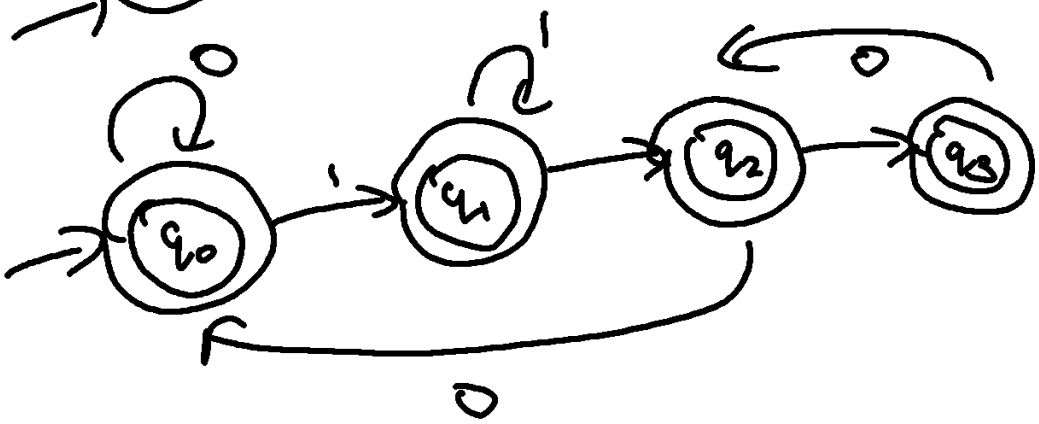


$L(\emptyset^*) = \{\varepsilon\}$

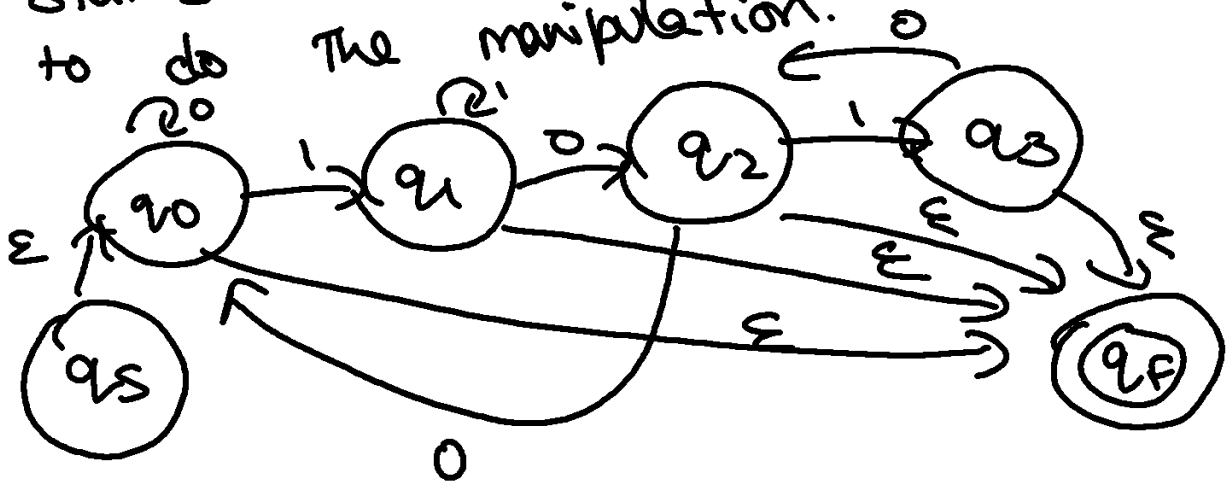
If a language is accepted by an NFA/DFA, then it is generated by regular expression.





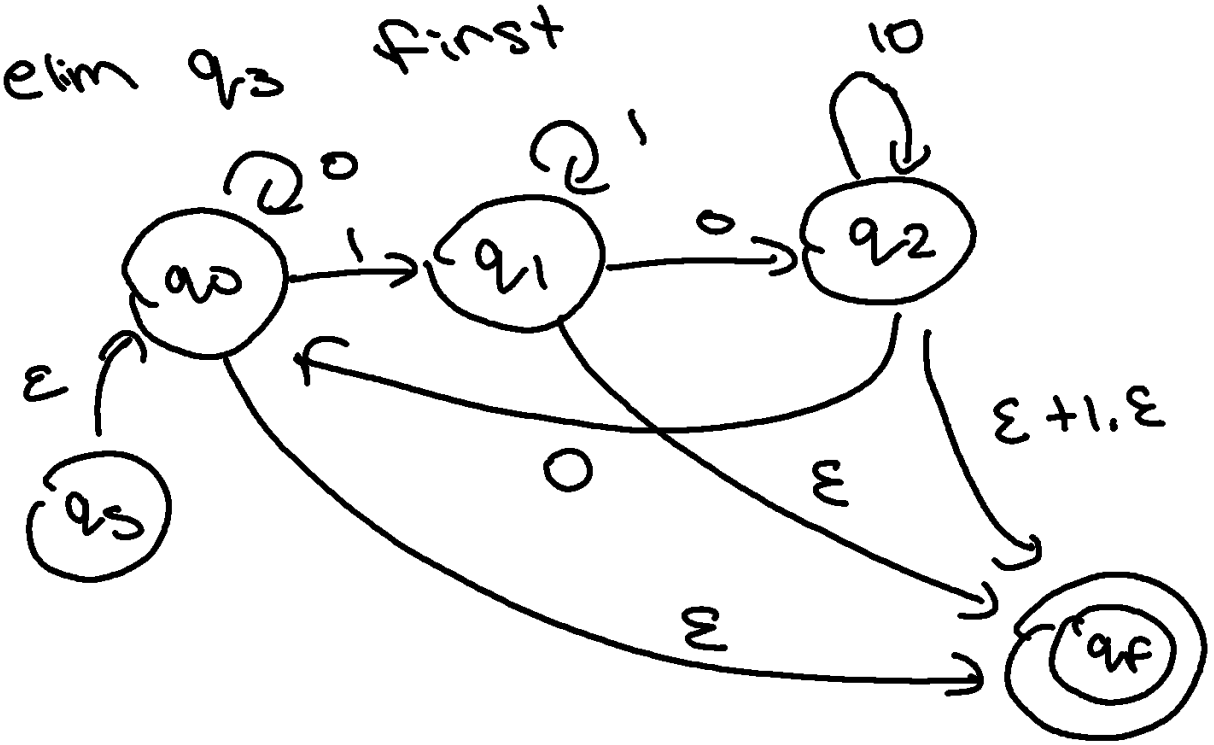


So many final states, since final states are important I want to eliminate states so that it becomes easier to do the manipulation.

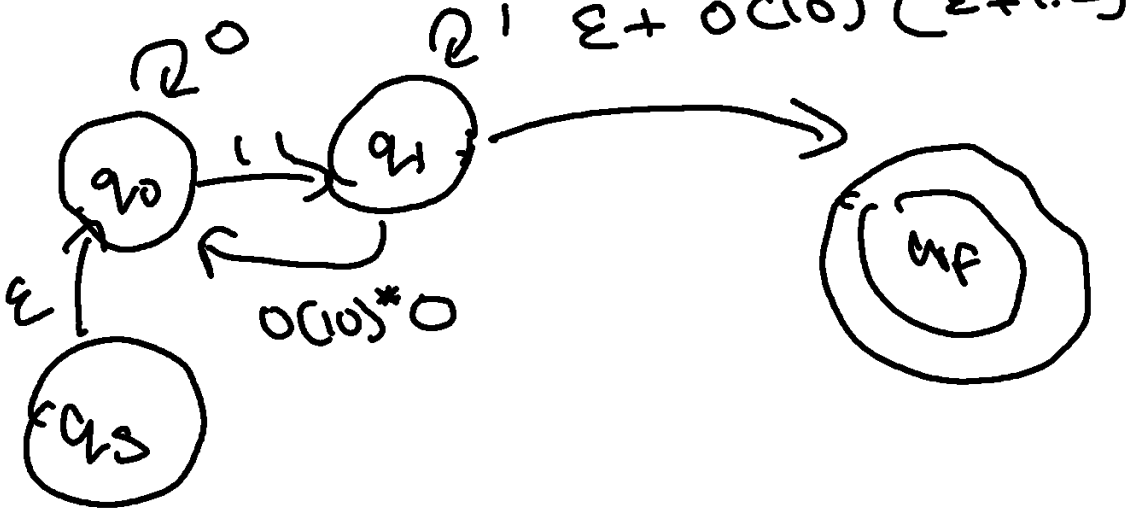


Start state should be a source and final state a sink.

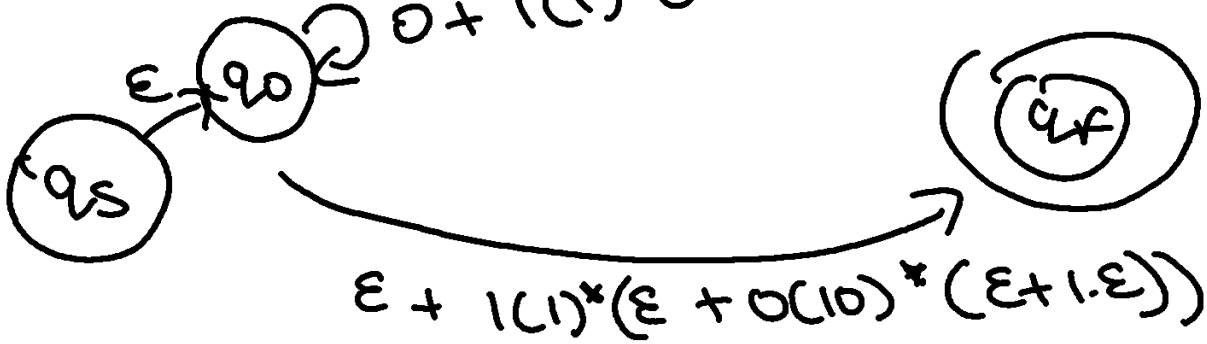
elim q_3 first



eliminate q_2 now

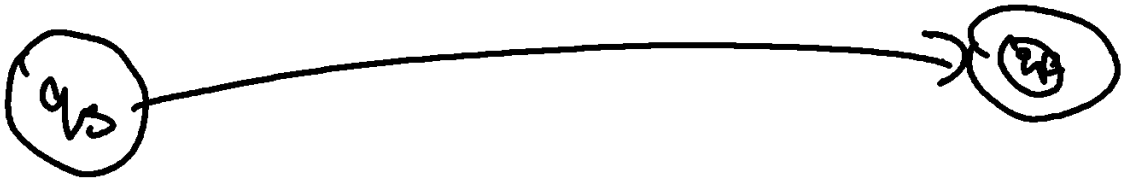


eliminate q_1



eliminate q_3

$(0 + 1.1^*0.(10)^*.0)^* (\varepsilon + 1.1^*(\varepsilon + 0(10)^*(\varepsilon+1.\varepsilon))$



Generalized NFA

A generalize non-deterministic finite automaton is a 5-tuple, $(Q, \Sigma, \delta, q_{start}, q_{accept})$, where

1. Q is the finite set of states
2. Σ is the input alphabet
3. $\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\})$
   → R is the transition (set of all regex over the alphabet)
4. q_{start} is the start state
5. q_{accept} is the accept state

q_0 - source and has out going transitions to all other states

q_1 - sink and has in-coming transitions from all other states.

$\forall\, q \in Q \setminus \{q_0, q_f\}$

- q has outgoing transitions to all other states except q_0
- q has in-coming transitions from all other states except q_f.
- q has a self transition

Consider a GNFA $M = (Q, \Sigma, q_0, q_f, \delta)$ and a word $s \in \Sigma^*$

A run of M on the word s is a sequence of states $r_0, r_1, \ldots r_n$ st

1. $r_0 = q_0$
2. $\exists s_1, \ldots s_n \in \Sigma^*$ st $s = s_1 \ldots s_n$ and $\forall i \in [n]$ $s_i \in L(\delta(r_{i-1}, r_i))$

- A run of M on a word s is called an accepting run if the last state in this run is the accepting state of M.
- A word s is said to be accepted by M if some run of M on w is an accepting run. Then the language accepted by the machine M can be defined as

$L(M) = \{s \in \Sigma^* \mid$ Some run of M on s is an accepting run

Claim
Every NFA can be converted to equiv GNFA.

Proof sketch

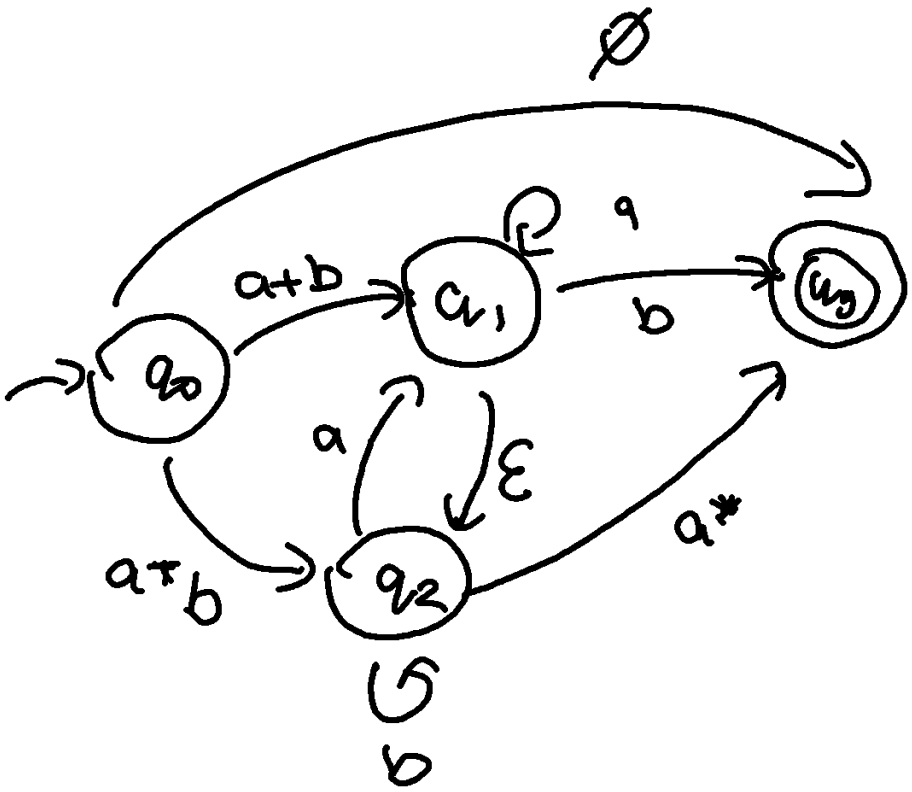
1 new final state, add ε transitions from old final states to new one

2 add all poss missing transitions that

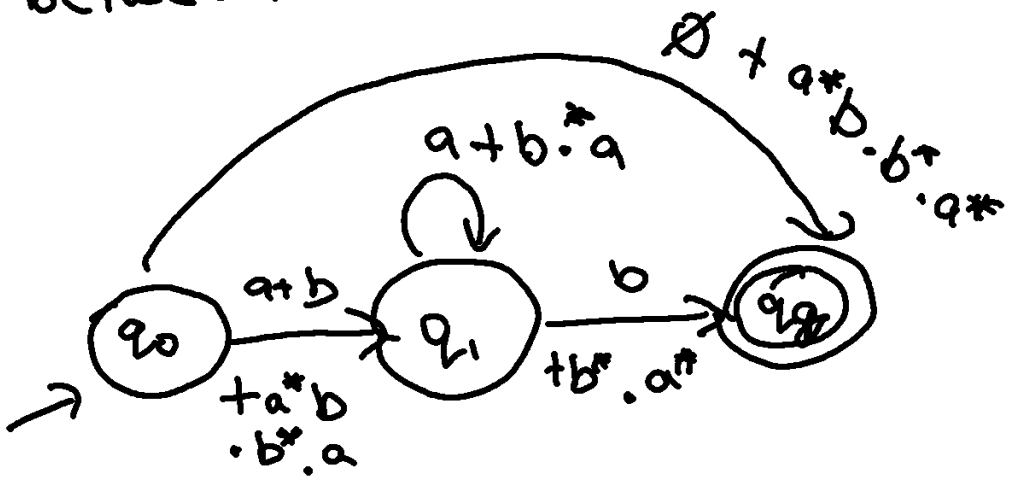
- leave a state other than q_f
- those that enter a state other than q_0

Label all these transitions $\emptyset$

GNFA to regex : example

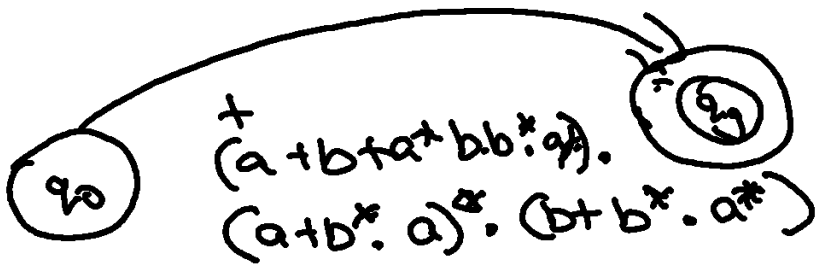


empty set transitions when no transition between 2 states.



eliminate q_2

Eliminate q_1 $\emptyset + a^*b.b^*.a^*$



$r = a^*bb^*a^* + (a+b+a^*bb^*a).(a+b^*a)^*.$
$(b+b^*a^*)$

Proving specific Languages non-regular

Let $L = \{0^n 1^n | n \geq 0\}$

is L regular

Intuitevly, some kind of counting seems neccesary

but DFA only has finite memory

- DFA has a memorlyless property.
- Consider words that transition q_i to q_i
- This is an equivalence relation with a finite number of equivalence classes (due to finite number of states)
- Strings x and y are distinguishible by L if $\exists_{string} z$ st $x.z \in L \wedge y.z \notin L$ or vice-versa.

The string "z" is certificate / witness that x and y are distinguishable by L.

Strings x and y are indistinguishible by L otherwise, in which case we write

$$x \equiv_L y$$

↖ equiv relation

the index of $\equiv_L$ is the number of equiv classes.

L Regular $\Rightarrow \equiv_L$ finite index
$\equiv_L$ infinite $\Rightarrow$ L irregular

Myhill-Nerode theorem is if and only if

to prove L is irreregular, we need to provide an infinite set of strings that are indistinguishable.

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

1. 0
2. 00
3. 000

$\vdots$

0^r

to show 0 and 0^r are indistinguishable

consider certificate = "1"

Similarly to show 0^m 0^r are indistinguishable

consider certificate = 1^m

∴ L can't be regular as infinite number of equiv classes

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

1 a certificate = bb

2 a a certificate = bbb

3 a a a —11— = bbbb

⋮ ⋮

r a^r —11— = b^{r+1}

note for 2) certificate = bbb a, aa are not indistinguishible but I've already shown that they are indistinguishible

Claim

Language $\{a^n b^n c^n \mid n \geq 0\}$ over $\{a, b, c\}$ is not regular

Proof

by Myhill-Nerode / Index of the language.

Consider the infinite set of strings generated by a^* For any a^i, a^j where $i \neq j$ observe $a^i a^{m-i} b^m c^m = a^m b^m c^m \in L$ while $a^j a^{m-i} b^m c^m$ $\notin L$. => set of strings generated by a^i.

Let $L = \{1^n \mid n \text{ is prime}\}$

1. 1
2. 1 1
3. 1 1 1

r. 1^r

Claim. I^i is distinguishible From I^j $\forall$ i,j st $i \neq j$

Let $i < j$

Pick Prime(p) st $p > \max(i,2)$

as $p > 2$ and $j - i > 0$ we Know

$p + j - i$ is prime

$p + p(j-1)$ is not prime

$= p(j)$ = composite

$p + 0(j-i)$

$p + 1(j-1)$

$\vdots$

$p + (p-1)(j-1)$ $\leftarrow$ not prime

let $1 \leq R \leq p$ be such a point

$p + (R-1)(j-1)$ is prime

$p + R(j-i)$ is not prime

$$I^i \cdot I^{[p+(R-1)(j-1)]-i} = I^{[p+(R-1)(j-1)]} \in L$$

$$I^j \cdot I^{[p+(R-1)(j-1)]-i} = I^{p+R(j-1)} \notin L$$

Proof of Myhill Nerode Theorem

L is regular <=> L has finite index

Testing if a DFA accepts an infinite language

if there is a cycle reachable from start state and which can reach an accept state in the state diagram of the DFA M,

iff

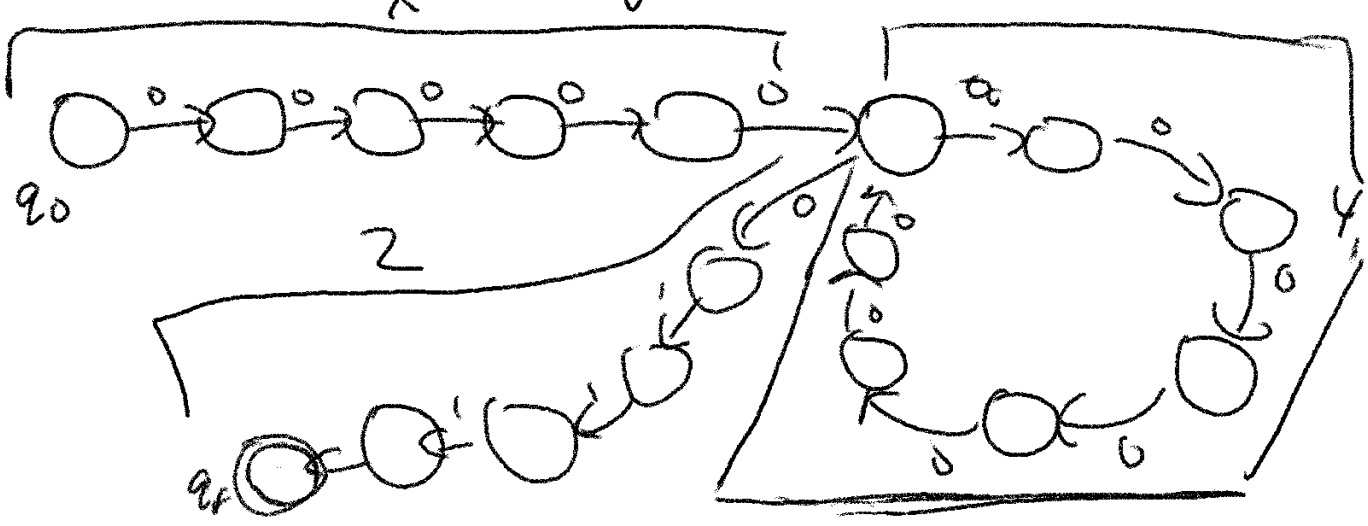
L(M) is infinite $\iff$ L(M) contains at least one string of length > # of states for M

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

L - regular? If yes then there is a DFA M: L(M) = L, let p be the number of states in M.

Consider run on $w = 0^{p+1} 1^{p+1}$

Must loop (some state occurs at least twice) before reading a 1.



$w = 0^{p+1} 1^{p+1}$

$= x \cdot y \cdot z$

$= 0^{\alpha} \cdot 0^{\beta} 0^{\gamma} 1^{p+1}$

$\alpha \geq 0 \quad \beta > 0 \quad \alpha + \beta \leq p$

$\alpha + \beta + \gamma = p+1$

$xz = 0^{\alpha+\gamma} 1^{p+1}$

$\alpha + \beta + \gamma = p+1$ and $\beta > 0$

$xz \notin L$ contradiction $\perp$

Pumping Lemma for Regular Lang.

IF L is a R.L, Then $\exists m > 0$ st $\forall w \in L : |w| \geq m$ $\exists$ a decomposition $w = xyz : |y| > 0, |xy| \leq m$ st $\forall i \geq 0$ $x y^i z \in L$

Proof Let $M = (Q, \Sigma, q_0, F, \delta)$ be a DFA accepting L.

Set $m = |Q|$

Consider any $w \in L : |w| \geq m$ Consider the run of w on M.

[If w doesn't exist, Than the Statement is vacuously true]

Let the run of w on M be denoted by $(\alpha_0, \alpha_1, \ldots, \alpha_{|w|})$, where $\alpha_0 = q_0$

$\alpha_{|w|} \in F$ and $\forall j \in \{0, \ldots, |w|\}$, $\alpha_j \in Q$

By pigeon hole $\exists j_1, j_2 : j_1 < j_2$ and $\alpha_{j_1} = \alpha_{j_2}$

choose smallest possible pair

Note
$j_1 = j_1 + 1$

By our choice of j_1 either $j_1 = 0$ or the "partial" run $(\alpha_0, \ldots \alpha_{j_1})$ has no repeating states $\therefore j_1 \leq |Q|$

Moreover, the partial run $(\alpha_{j_1}, \ldots, \alpha_{j_2})$ is a loop in the state transition diagram of M

Define
x = The substring of w consumed by M in the partial run $(\alpha_0, \ldots, \alpha_{j_1})$

y = The substring of w consumed by M in the partial run $(\alpha_{j_1}, \ldots, \alpha_{j_2})$

z = substring of w st $w = xyz$

We have already argued that
- $|y| > 0$
- $|xy| \leq |Q| = m$

Remains to prove that $\forall i \geq 0$, $xy^i z \in L$

But this is true since $\forall i \geq 0$, the run $(\alpha_0, \ldots \alpha_{j_1}) \cdot (\alpha_{j_1}, \ldots \alpha_{j_2})^i \cdot (\alpha_{j_2}, \ldots \alpha_{|w|})$ is an accepting run in M.

Statement of pumping Lemma Written nicely

Let L be a regular language. then $\exists m \in \mathbb{Z}_{>0}$ st $\forall w \in L$ with $|w| \geq m$ $\exists$ decomposition $w = xyz$ where

- $|xy| \leq m$
- $|y| \geq 1$
- $\forall i \in \mathbb{N}_0$, $xy^i z \in L$

To show non-regular, consider contrapositive.

$\forall m \in \mathbb{Z}_{>0}$, $\exists w \in L$ with $|w| \geq m$
$\forall$ decompositions $w = xyz$ with

- $|xy| \leq m$
- $|y| \geq 1$
- $\exists i \in \mathbb{N}_0$, $xy^i z \notin L$

$\Rightarrow$ L is non-regular

So the Proof Structure is
Step 1. Suppose L is regular, let m be the pumping length of L.

Step 2. Choose a string $w \in L$ st $|w| \geq m$

Step 3. Let $w = xyz$ be an arbitrary decomposition of w st $|xy| \leq m$ $|y| \geq 1$

Step 4. Carefully pick an integer i and argue that $xy^iz \notin L$.

Example

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

1. Suppose L is regular with pumping length m.
2. pick $w = 0^m 1^m$, notice $|w| = 2m \geq m$
3. Consider the partition $x = 0^\alpha$, $y = 0^\beta$ and $z = 0^\gamma 1^m$, where $\beta > 0$ and $\alpha + \beta + \gamma = m \Rightarrow \alpha + \beta \leq m \therefore |xy| \leq m$
4. pick $i = 0$, the $xy^0z = 0^\alpha 0^\gamma 1^m$ Since $\alpha + \gamma < m \therefore \alpha + \gamma \neq m \therefore$ $xy^0z \notin L \therefore$ our initial assumption of the regularity of L must be false.

$L = \{1^n \mid n \text{ is prime}\}$

1. Suppose L is regular with pumping length m.

2. let $w = 1^q$, where q is smallest prime $> m+1$ ($\exists$ an infinite no. of primes)

3. $w = 1^\alpha 1^\beta 1^\gamma$

   $\alpha+\beta+\gamma = q \Rightarrow \alpha+\beta \leq m$

   $\beta > 0$

   $\alpha+\gamma > 1$ as $|1^q| > m+1$

4. pick $i = \alpha+\gamma$

   $1^{\alpha+(\alpha+\gamma)\beta+\gamma} = 1^{(\beta+1)(\alpha+\gamma)} \notin L$

   as $(\beta+1)(\alpha+\gamma)$ is not prime.

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

1. let language be regular with pumping length m

2. Choose $w = a^m b^{m+1}$

3. $x = a^\alpha \quad y = a^\beta \quad z = a^\gamma b^{m+1}$

4. pick $i = \frac{m+1}{\beta}$

$$xy^iz = a^\alpha a^{m+1} a^\gamma b^{m+1}$$
$$= a^{\alpha+m+1+\gamma} b^{m+1}$$

as $\alpha + \gamma \geq 0 \quad \therefore \quad \alpha + m + 1 + \gamma \geq m+1$

thus $xy^iz \notin L$

Another Strategy

$L = \{w \in \{0,1\}^* \mid n_0(w) = n_1(w)\}$

Suppose L is regular

$L_1 = L(0^*1^*)$ is also regular

Since regular languages are closed under intersection $\therefore L \cap L_1$ is regular

however, $L \cap L_1 = \{0^n 1^n \mid n \geq 1\}$ which we know to be non-regular. $\therefore$ L cant be regular.

Similarly $L = \{w \in \{0,1\}^* \mid n_0(w) \neq n_1(w)\}$
Suppose regularity of L then L^c must be regular, but we just prove it is not $\therefore$ L is not-regular.

Formal definition of a grammar

$G = (V, \Sigma, R, S)$

V = finite set of variables/non-terminals
Σ = finite alphabet
R = finite set of production rules

(a set of strings of the form

"$\alpha \to \beta$" where $\alpha, \beta \in (V \cup \Sigma)^*$ and α is non-empty)

$S \in V$ is the start Variable

eg. $G = (\{S\}, \{0,1\}, R, S)$

R: $S \to 0S1$
$S \to \varepsilon$

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111$

This is a derivation of the string "000111" from S (or the grammar G)

$S \overset{*}{\Rightarrow} 000111$

S yields $0S1$
$0S1$ yields $00S11$

Some Notation

$\alpha \Rightarrow \beta$ if α can be rewritten as β by applying a production rule

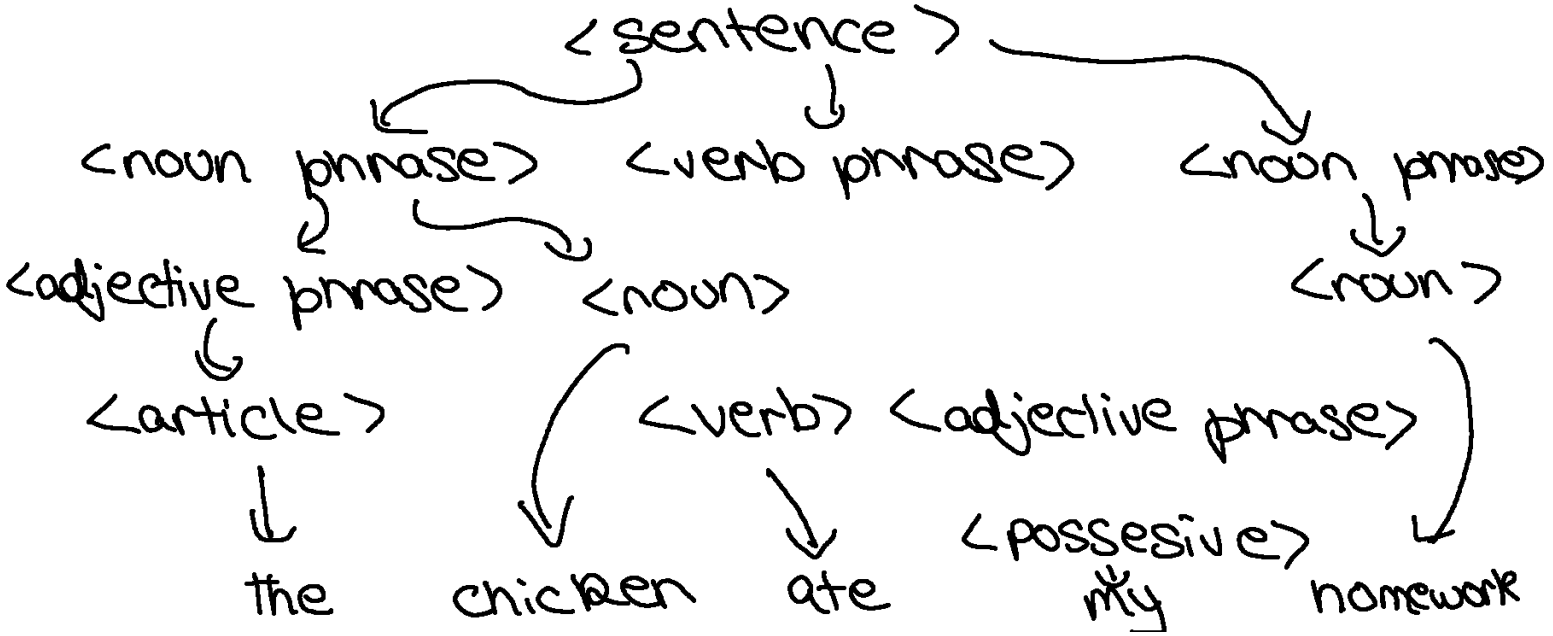
$\alpha \overset{*}{\Rightarrow} \beta$ if α can be rewritten as β by applying a finite number of production rules in succesion

$G = (V, \Sigma, R, S)$
$L(G) = \{w \in \Sigma^* \text{ st } S \overset{*}{\Rightarrow} w\}$ is the language generated by this grammar

(Left/Right) - most derivation

always apply rule to (left/right) most available variable.

Parse Trees



A grammar G is ambigious if it can generate the same string with multiple parse trees.

↑↓

A grammar G is ambigious if the same string can be derived with two left-most derivations.

Some ambigious grammars can be rewritten as an equivalent unambigious grammar.

=> Some grammars "inherently ambiguous" grammars.

Q. Can we have an algorithm that takes a grammar description as input and figures out whether or not it is ambiguous.

- Brute force runs forever ∴ this problem is undecidable.

Chomsky Hierarchy of Grammars

$G = (V, \Sigma, R, S)$

$A, B \in V$

$\alpha, \beta, \gamma, w \in (V \cup \Sigma)^*$

$x \in \Sigma^*$

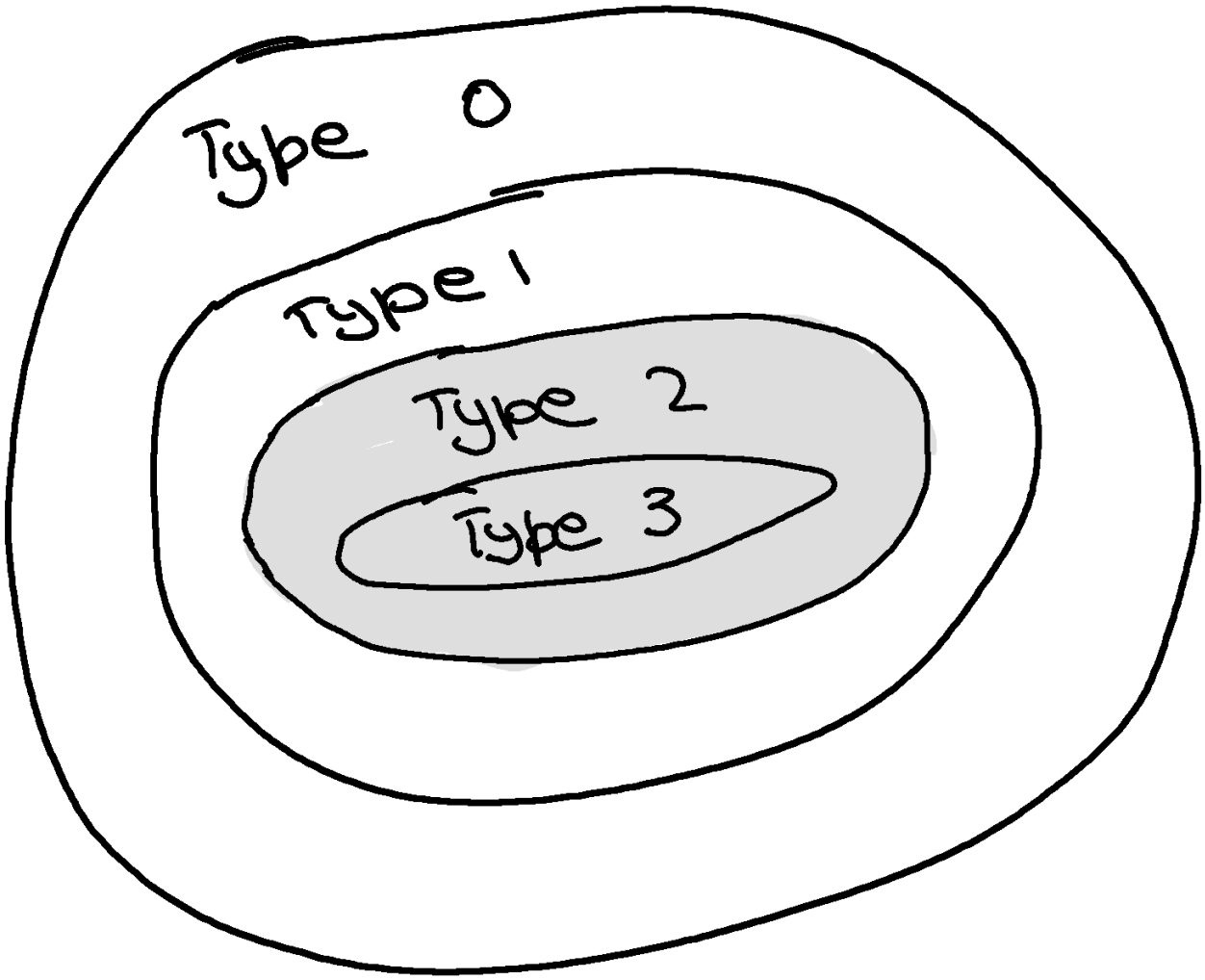
Type 3 Regular (Right / Left) - linear
$A \to xB$ OR $A \to Bx$
$A \to x$ $A \to x$

Type 2 Context - Free
$A \to w$

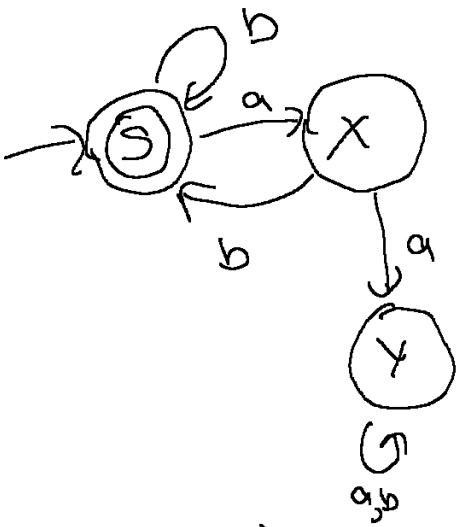
Type 1 / Context Specific
$\alpha A \gamma \to \alpha B \gamma$

Type 0 / Recursively-enumerable
$\alpha \to B$
(assuming α is non-empty)

- focus on this for the next few weeks



DFAs to right-linear grammars



$G = (\{S, X, Y\}, \{a, b\}, R, S)$

all strings that take me to some accept state if I start from •

- $S \to b.S \mid a.X \mid \varepsilon$
- $X \to b.S \mid a.Y$
- $Y \to a.Y \mid b.Y$

b followed by a way to get to an accepting state from S

$R :=$

$S \to b.S \mid a.X \mid \varepsilon$

$X \to b.S$

(no way to get to accepting state from Y)

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

let $G = (V, \Sigma, R, S)$

- $V = Q$, $S = q_0$
- R:
  - $\forall q, q' \in Q, a \in \Sigma$ if $\delta(q, a) = q'$ then add rule $q \to a\, q'$
  - $\forall q \in F$, add rule $q \to \varepsilon$

Claim $\forall s \in \Sigma^*, q \in Q$
$\hat{\delta}(q, s) \in F$ iff $q \stackrel{*}{\Rightarrow} s$

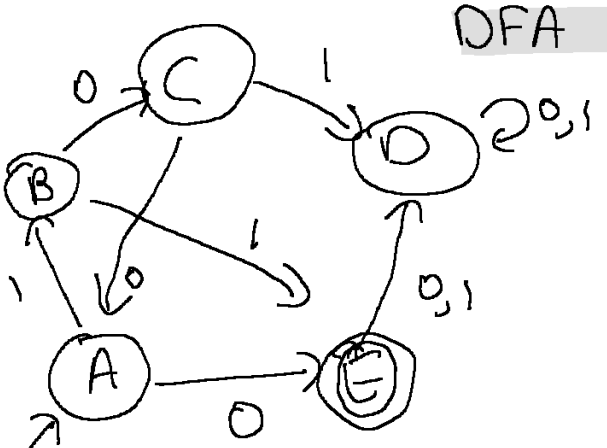
Consequence $\forall s \in \Sigma^*$

$s \in L(M)$ iff $\hat{\delta}(q, s) \in F$

iff

$q_0 \stackrel{*}{\Rightarrow} s$ iff $s \in L(G)$

DFA to right-linear Grammar



$G = (\{A, B, C, D, E\}, \{0, 1\}, R, E)$

$A \to 1.B \mid 0.E$
$B \to 1.E \mid 0.C$
$C \to 1.D \mid 0.A$
$D \to 0.D \mid 1.D$
$E \to 0.D \mid 1.D \mid \varepsilon$

Simplifying Grammar (D is a dead-state)

$A \to 1.B \mid 0.E$
$B \to 1.E \mid 0.C$
$C \to 0.A$
$E \to \varepsilon$

DFA to left-linear Grammar

Same setup as $A \to$ all strings that will take me to A from the start state

Given DFA $M = (Q, \Sigma, q_0, F, \delta)$
let $G = (V, \Sigma, R, S)$

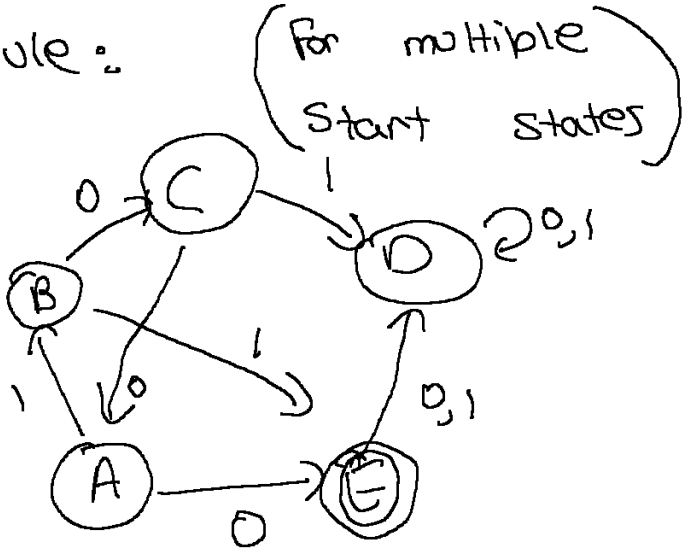
- $V = Q$, $S = q_0$

R: - add $q_0 \to \varepsilon$

- $\forall q, q' \in Q$, $a \in \Sigma$

if $\delta(q, a) = q'$ then add rule:
$q' \to q\,a$

- $\forall q \in F$, add rule:
$q^* \to q$



$A \to \varepsilon \mid C.0$
$B \to A.1$
$C \to B.0$
$E \to B.1 \mid A.0$

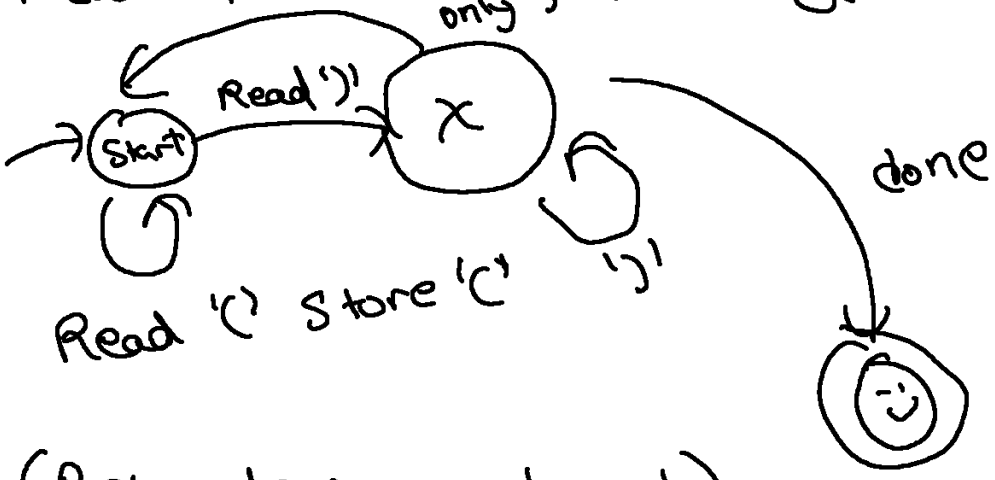
CFL's be counting

2) $L = \{$String of balanced paranthesis$\}$

$S \to (S) \mid SS \mid \varepsilon$

DFA's be crying due to a lack of counting

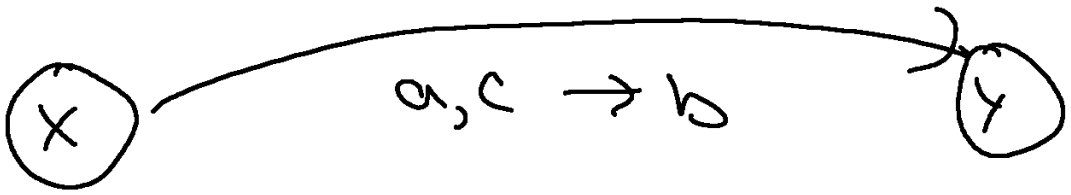
New machine



(Push down automata) ↘
Finite automation + Stack ↘ just allow non-determinism

pushdown automata is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ Q, Σ Γ and F are all finite sets

Q - set of states
Σ - input alphabet
Γ - Stack alphabet
$\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ - transition function
$q_0 \in Q$ Start state
$F \subseteq Q$ set of accep states



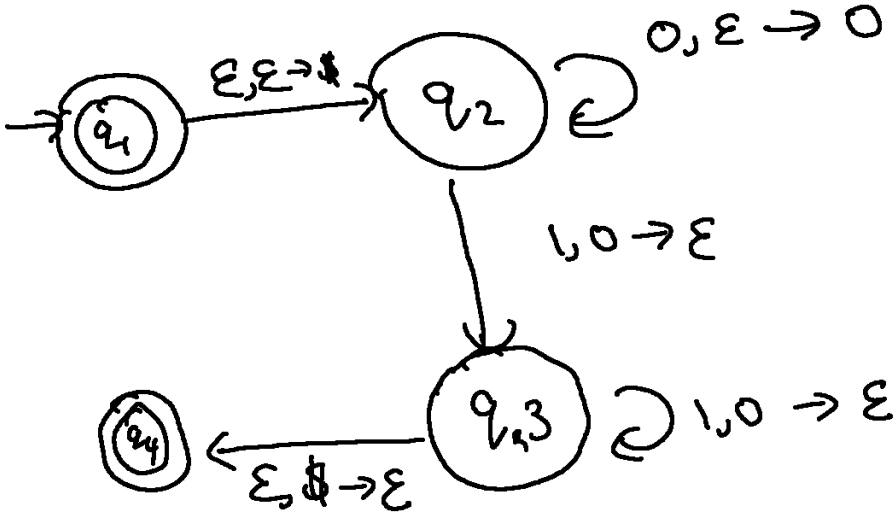
From X read 'a' pop 'c' push 'b' and move to Y.

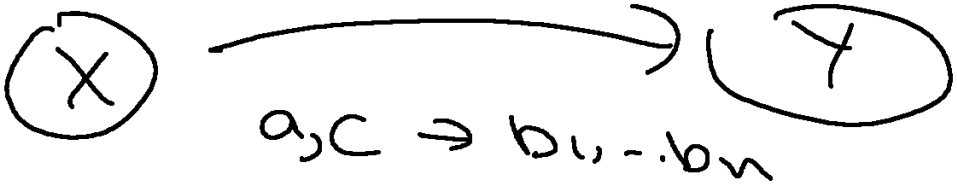
$(Y, b) \in \delta(X, a, c)$ (transition Function)

$\varepsilon, \varepsilon \to \varepsilon$ do nothing to stack

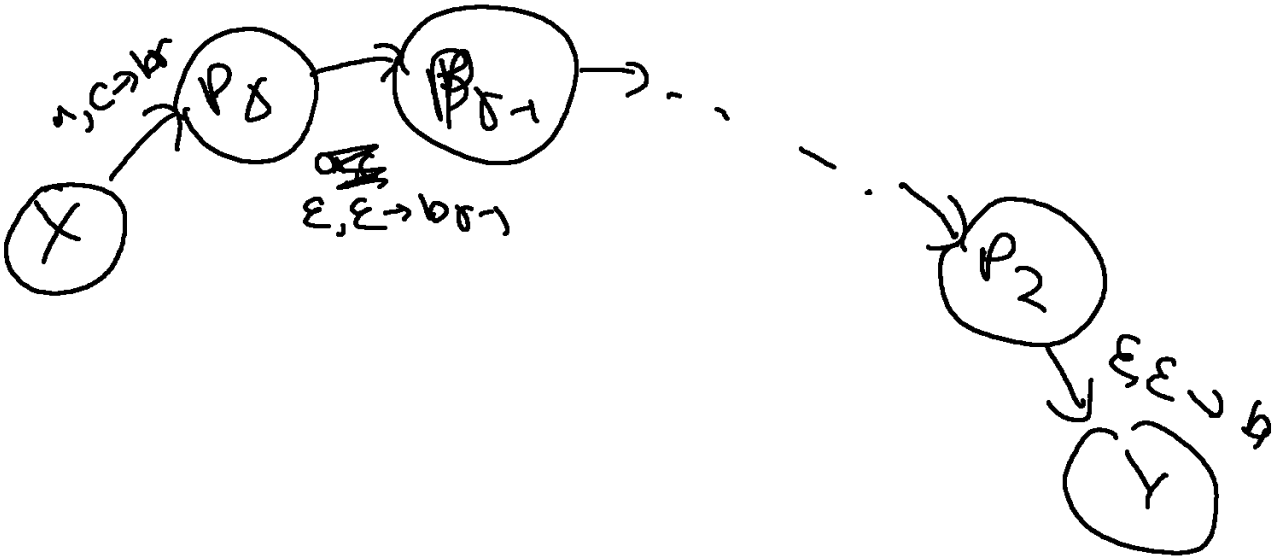
pop \$ $\varepsilon, \$ \to \varepsilon$

push \$ $\varepsilon, \varepsilon \to \$$





is equiv to



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

M accepts $\omega \in \Sigma^*$
if $\exists\ \omega_1, \ldots \omega_\gamma \in \Sigma \cup \{\varepsilon\}$
$\exists\ x_0 x_1 \ldots x_\gamma \in Q$,
$\exists\ S_0, \ldots S_\gamma \in \Gamma^*$

ST $\omega = \omega_1 \ldots \omega_\gamma$ and
i) $x_0 = q_0$, $S_0 = \varepsilon$
ii) $x_\gamma \in F$
iii) $\forall i \in \{0, \ldots \gamma_{i-1}\}$

$\delta(\omega_i, \omega_{i+1}, a) \rightarrow (x_{i+1}, b)$

$S_i = a.t$ S_{i+1} $b.t$ for
$a, b \in \Gamma \cup \{\varepsilon\}$ and $t \in \Gamma^*$

Pushdown automation = Context Free Lang

Proof (=>) and (<=)
(<=)

Non Deterministic Algorithm to compute a left-most derivation for a given string

1. currString = Start Variable (S)
2a) If leftmost element is a variable A then select one of the grammar rules for A and replace left-most A in currString with string on the RHS of rule
(keep a record of this application)

b) if left-most elem is a terminal a, then read the next input symbol and check if it is the same as a. If so, then remove the left most element of currString and repeat. Else stop this branch of the Algorithm

c) If currString is empty, then we have completed the derivation and so output the recorded grammar rules in reverse order and stop.

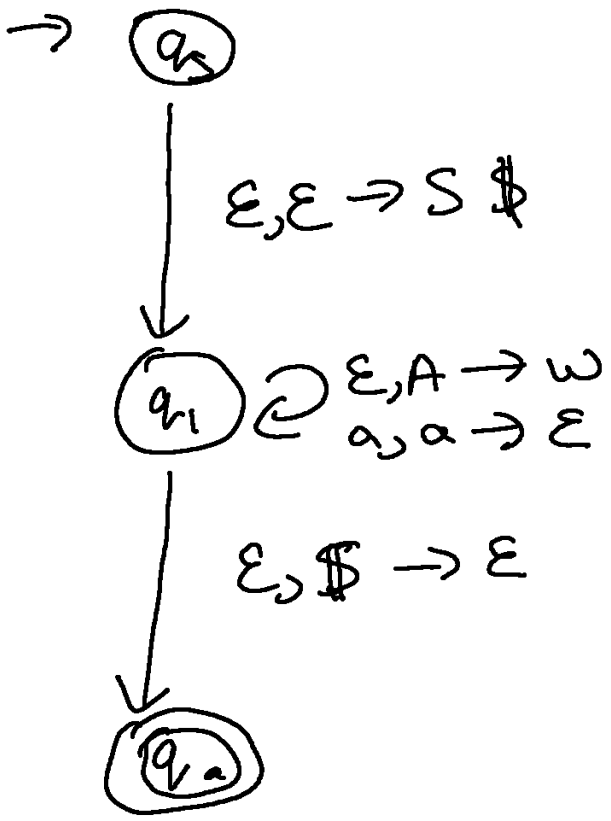
PDA's algorithm to check if a left-most derivation of a given string exists

1. Push S onto Stack

2. a) If Peak(Stack)= Variable then Pop(A), Push application of rule

   b) if Pop(Stack) is terminal a, then peak Stack and check if same as a. If so then do nothing else reject branch of PDA

   c) if top is $, then accept



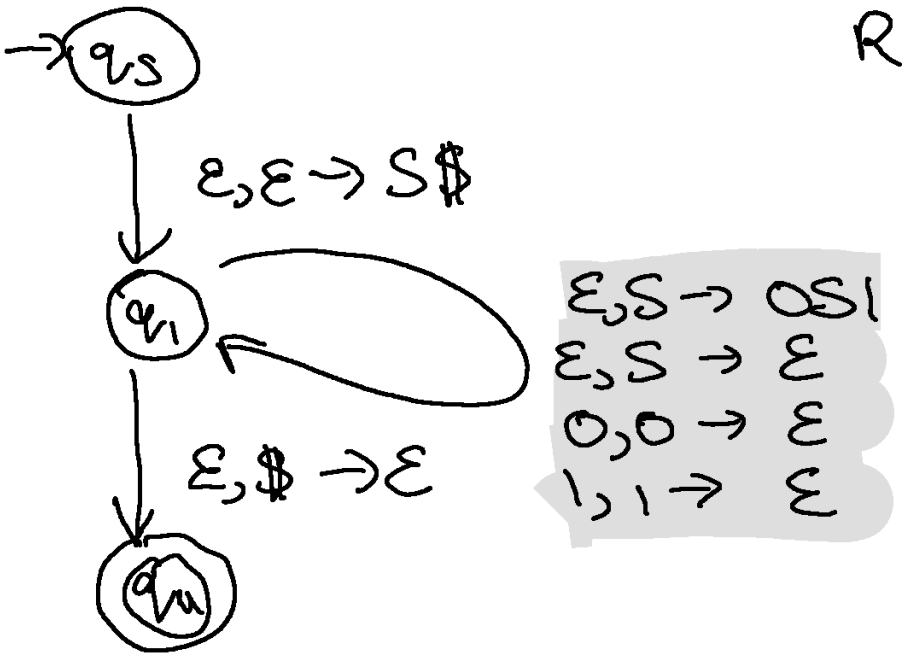
Construction of PDA from a given CFG will rely on the following 3 states

- Start State
- Loop State q_1 (Simulation of grammar rules)
- Accept State

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

$G = (\{S\}, \{0,1\}, R, S)$

R: $S \to 0S1$
$S \to \varepsilon$



PDA → normalized PDA → CFG

Normalized PDA (Proof diagrams in slides)
1. Single accept state q_{accept}
2. Empties stack before accepting
3. Each transition either pushes a symbol onto the stack or pops one of the stack, but it doesn't do both at the same time.

There is a non-terminal A_{pq} for each pair of states p and q.

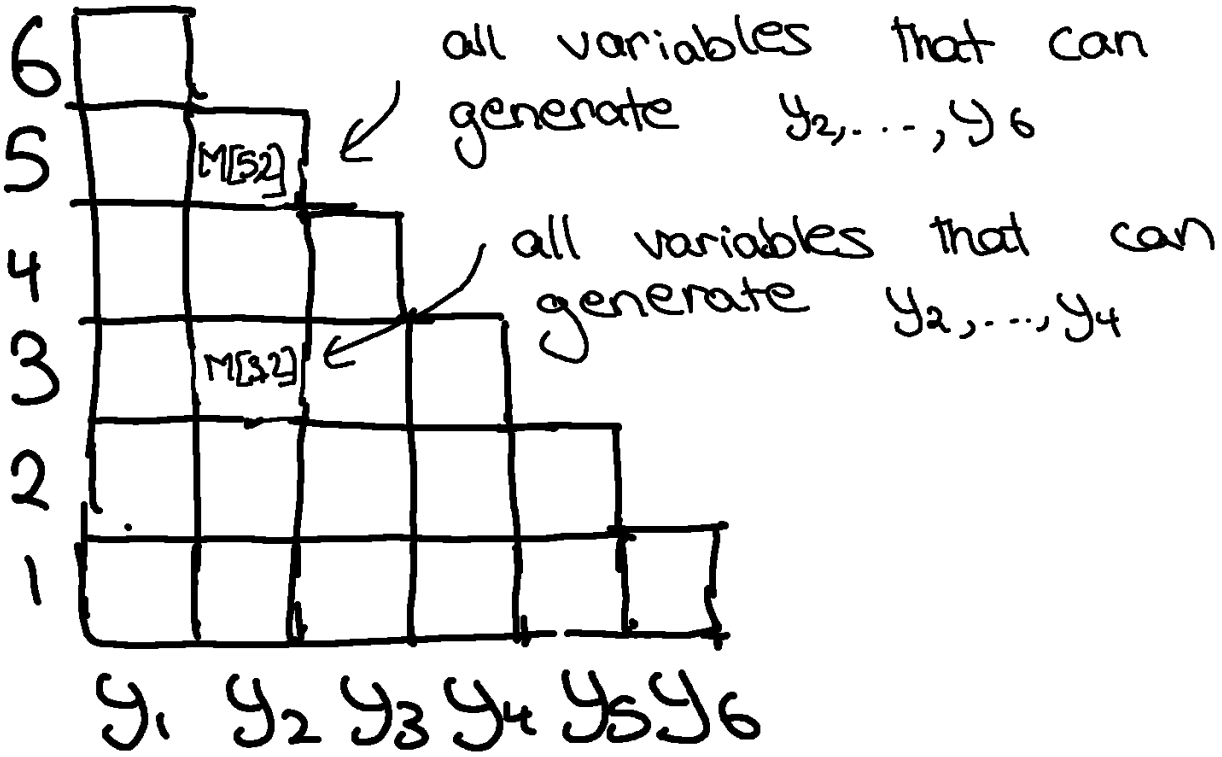
A_{pq} generates all strings which takes us from state p (empty stack) to state q (with empty stack)

Language accepted by the PDA is the set of strings which can be derived from the non-terminal $A_{q_{initial} q_{final}}$

Chomsky Normal Form of Grammars

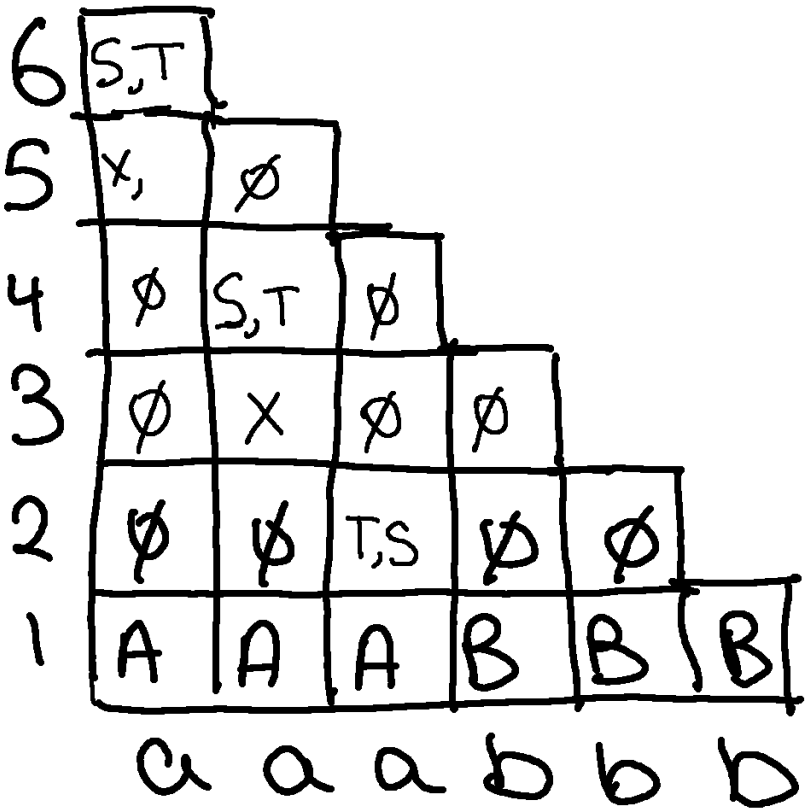
A grammar $G=(V,\Sigma,R,S)$ is in chomsky normal if every production has one of the following shapes

- $S \to \varepsilon$
- $A \to x$ ($x \in \Sigma$ is a terminal)
- $A \to BC$ ($B, C \in V \setminus B, C \neq S$)


6
5
4
3
2
1
M[5,2]
M[3,2]
all variables that can generate $y_2, \ldots, y_6$
all variables that can generate $y_2, \ldots, y_4$
y_1 y_2 y_3 y_4 y_5 y_6

CYK Algorithm

$S \to \varepsilon \mid AB \mid XB$
$T \to AB \mid XB$
$X \to AT$
$A \to a$
$B \to b$



Take a CFG $G = (V, \Sigma, R, S)$

$LHS(P \times Q) = \{J \in V \mid G \text{ has a rule } J \to XY \text{ where } X \in P \text{ and } Y \in Q\}$

```
for i = 2, ... l
    for j = 1, ... l-(i-1)
        for p = 1, ..., i-1
            M[i,j] = M[i,j] ∪ LHS(M[p,j] × M[i-p, j+p])
```

Cool matrix multiplication algorithm

w can be derived from G iff $M[l,1]$ contains S

Advantages of having CFG in Chomsky Normal form

1. The CYK algorithm can be used to perform a bottom up parsing of strings in polynomial time

2. Every string of length n can be derived in exactly $2n-1$ steps.

Theorem
Any context-free language is generated by a context-free grammar in Chomsky normal form

Converting to Chomsky-Normal form

$$G = (V, \Sigma, R, S)$$

1. Create a new start variable S_0 and add $S_0 \to S$

2. Eliminate $A \to \varepsilon$ productions where $A \neq S$

by going over every rule and all subsets of the sets of occurences of A on the RHS of this rule and for each subset, add a new RHS where the occurences of A in this subset are replaced with ε.
(don't add in ε-production if it is produced)

$S_0 \to S$
$S \to ASB$
$A \to aAS \mid a \mid \varepsilon$
$B \to SbS \mid A \mid bb$

→

$S_0 \to S$
$S \to ASB \mid SB$
$A \to aAS \mid a \mid aS$
$B \to SbS \mid bb \mid \varepsilon \mid A$

$S_0 \to S$
$S \to ASB \mid SB \mid AS \mid S$
$A \to aAS \mid a \mid aS$
$B \to SbS \mid bb \mid A$

can just be removed

3. Eliminate unit productions ($A \to B$)

Eliminate $A \to B$ production by removing it and adding a $A \to \omega$ production for every $B \to \omega$ productions.

$S_0 \to S$
$S \to ASB \mid SB \mid AS$
$A \to aAS \mid a \mid aS$
$B \to SbS \mid bb \mid aAS \mid a \mid aS$

→

$S_0 \to ASB \mid SB \mid AS$
$S \to ASB \mid SB \mid AS$
$A \to aAS \mid a \mid aS$
$B \to SbS \mid bb \mid aAS \mid a \mid aS$

4. Add new variables and productions to eliminate remaining violations in rules of the form $A \to \omega$, where $|\omega| > 2$ and in rules of the form $A \to xy$, where x, y are both terminals.

$S_0 \to ASB \mid SB \mid AS$

$S \to ASB \mid SB \mid AS$

$A \to aAS \mid a \mid aS$

$B \to SbS \mid bb \mid aAS \mid a \mid aS$

$S_0 \to AU_1 \mid SB \mid AS$

$S \to ASB \mid SB \mid AS$

$A \to aAS \mid a \mid aS$

$B \to SbS \mid bb \mid aAS \mid a \mid aS$

$U_1 \to SB$

$S_0 \to AU_1 \mid SB \mid AS$

$S \to AU_2 \mid SB \mid AS$

$A \to aAS \mid a \mid aS$

$B \to SbS \mid bb \mid aAS \mid a \mid aS$

$U_1 \to SB$

$U_2 \to SB$

$S_0 \to AU_1 \mid SB \mid AS$

$S \to AU_2 \mid SB \mid AS$

$A \to aU_3 \mid a \mid aS$

$B \to SbS \mid bb \mid aAS \mid a \mid aS$

$U_1 \to SB$

$U_2 \to SB$

$U_3 \to AS$

$S_0 \to AU_1 \mid SB \mid AS$

$S \to AU_2 \mid SB \mid AS$

$A \to aU_3 \mid a \mid aS$

$B \to SU_4 \mid bb \mid aU_5 \mid a \mid aS$

$U_1 \to SB$

$U_2 \to SB$

$U_3 \to AS$

$U_4 \to bS$

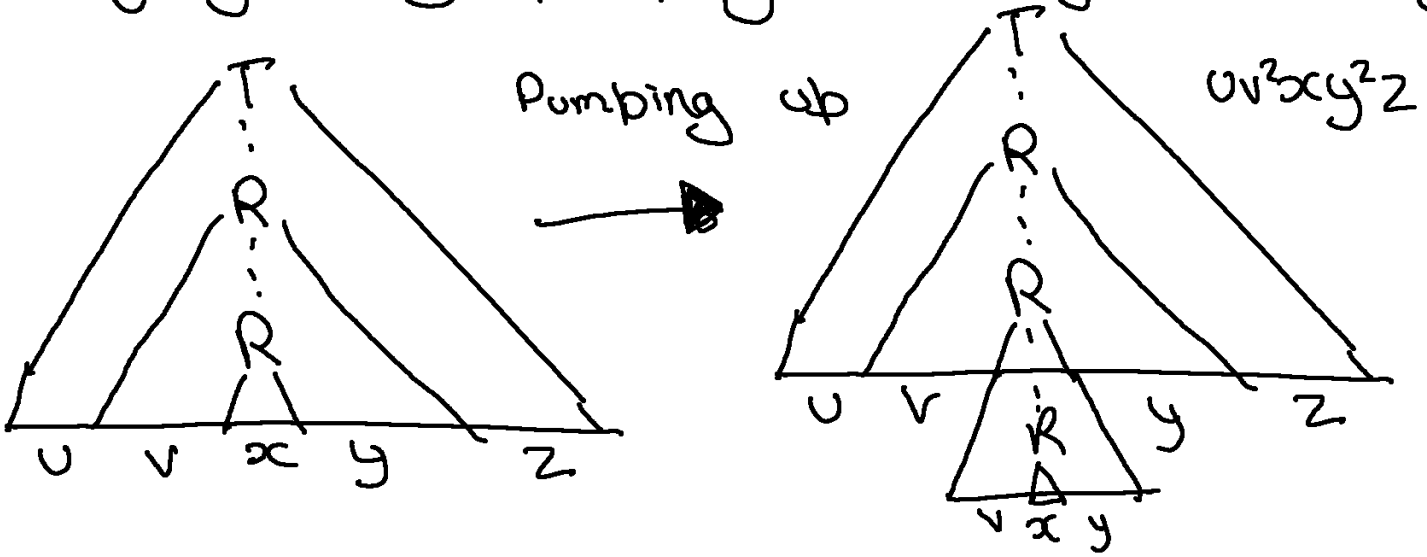
$U_5 \to AS$

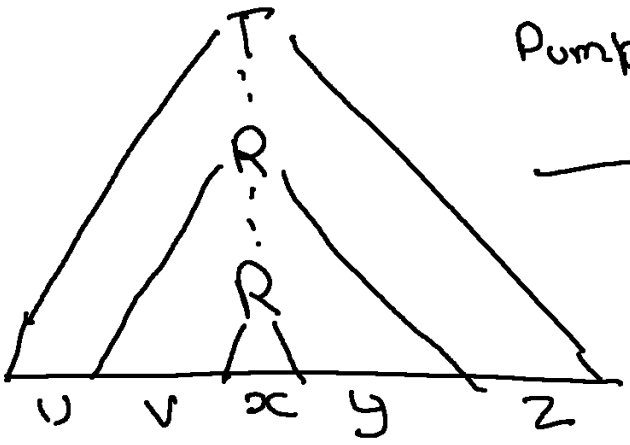
$S_0 \to AU_1 \mid SB \mid AS$
$S \to AU_2 \mid SB \mid AS$
$A \to V_1U_3 \mid a \mid aS$
$B \to SU_4 \mid V_2V_1 \mid V_1U_5 \mid a \mid V_1S$
$U_1 \to SB$
$U_2 \to SB$
$U_3 \to AS$
$U_4 \to bS$
$U_5 \to AS$
$V_1 \to a$
$V_2 \to b$

Idea for Pumping Lemma for Context free languages

If a derived string is too long then it must repeat a non-terminal somewhere in the parse tree.

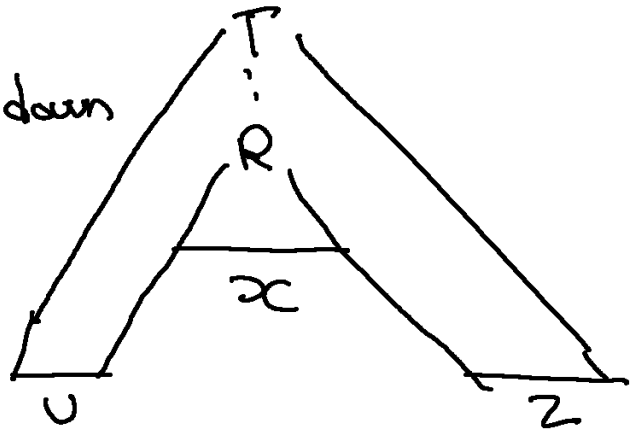
Use the repeated non-terminal to come up with an infinite list of strings in the language by "pumping" the right substrings.





uvxyz

Pumping down →



uv^0xy^0z

Pumping Lemma for a Context-free Language

Let L be a CFL. Then $\exists\, m \in \mathbb{Z}_{>0}$ s.t. $\forall w \in L$ with $|w| \geq m$ $\exists$ decomposition $w = uvxyz$ where

- $|vxy| \leq m$
- $|vy| \geq 1$
- $\forall i \in \mathbb{N}_0$, $uv^ixy^iz \in L$

Take a context-free grammar $G = (V, \Sigma, R, S)$

If the length of the longest string in the RHS of any production rule is b, then any node in any parse tree yielding a string in $L(G)$ has at most b children.

R: $A \to w$ $\quad |w|, |w'| \leq b$

$B \to w'$

If every node of a rooted tree has at has at most b children and the height of this tree is h then the number of leaves is b^h.

If the height of the tree is greater than $|V|$ then some variable must repeat.

∴ any string of length at least $b^{|V|+1}$ must have this property.

∴ $m = b^{|V|+1}$

Assume for a contradiction $|vy| = 0$
⇒ $uvxyz = uv^0xy^0z = uv^ixy^iz \; \forall i \geq 2$
∴ conditions are trivially met for all words.

Assume for a contradiction $|vxy| > m$,
⇒ can redefine decomposition so that $|vxy| \leq m$ and the interesting structure is preserved.

CFL PL

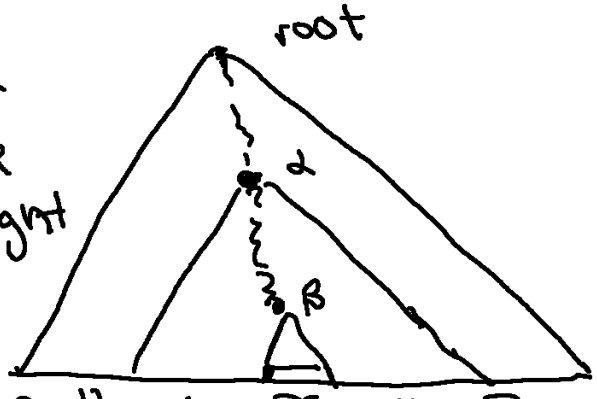
If L is a CFL, then $\exists m > 0$ st $\forall w \in L : |w| \geq m$
$\exists$ a decomposition $w = uvxyz : |vy| > 0 \; |vxy| \leq m$
$\forall i \geq 0$, $uv^i xy^i z \in L$

Proof

Let $G = (V, \Sigma, S, R)$ be a CFG generating L.
Let $b = \max_{x \to \gamma \in R} \{|\gamma|\}$
Set $m = b^{|V|+1}$
Considier any $w \in L : |w| \geq m$
Consider a smallest parse tree T of w

Observe height $(T) \geq |V| + 1$ (v internal node)
by P.H.P. $\exists A \in V$ and nodes $\alpha, \beta \in I(T)$ st $v(\alpha) = v(\beta) = A$, α is an ancestor of β in T.

W.L.O.G assume subtree at α has $\leq b^{|V|+1}$ leaves. This can be ensored by picking α of height $\leq |V| + 1$.



Let x be the yield of the subtree rooted at β. Let v, y be st vxy is the yield of the subtree rooted at α.

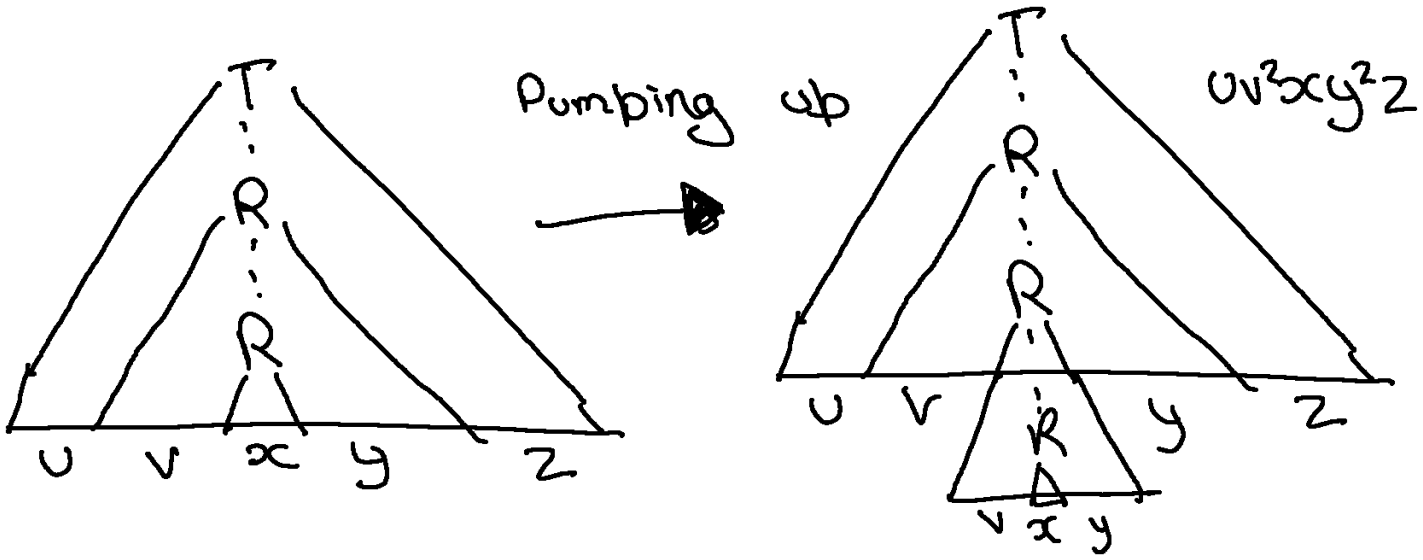
Let u, z be st $uvxyz$ is the yield of the subtree rooted at the root.

i.e. $w = uvxyz$

- $|vxy| \leq m$ since the height of d is at most $|V|+1$ and $|vxy| \leq |b|^{|V|+1} = m$

- $|vy| > 0$ since T is a smallest parse tree of w

- $\forall i \geq 0$ $uv^i xy^i z \in L$ since a parse tree can be extended i times by pumping up



(or down).

Using Pumping Lemma to prove a language is not a CFL

- Assume L is CFL
- Let m be the pumping length
- choose a string $w \in L$ st $|w| \geq m$
- Take an arbitrary decomposition $w = uvxyz$, $|vxy| \leq m$ $|vy| > 0$
- choose an i st $uv^i xy^i z \notin L$ contradiction

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

Assume CFL, m = pumping length

$w = a^m b^m c^m \in L$ ($|w| \geq m$ trivially)

$w = uvxyz$ arbitrary $|vxy| \leq m$

$|vy| > 0$

Case 1: $vxy \in a^*$

$a^m b^m c^m = \underbrace{a \cdots a}_{vxy} b \cdots b c \cdots c = uvxyz$

Similarly for the 4 other cases

$uv^0xy^0z \notin L$ so $i=0$ works as $|vxy|$ contains at least 1 symbol from $\{a,b,c\}$ and at most 2 symbols

$\therefore uv^0xy^0z \notin L \Rightarrow L$ is not a CFL.

How to check whether language generated by $G = (V, \Sigma, R, S)$ is finite?

If $L(G)$ contains no strings of length longer than the pumping length m, then the language is finite.

If L(G) contains even one string of length longer than m it contains a string of length at most 2m-1.
(anything bigger, we can pump it down as interesting structure is preserved)

Closure Properties for CFLs

1. Union

$G_1 = (V_1, \Sigma_1, R_1, S_1)$ $G_2 = (V_2, \Sigma_2, R_2, S_2)$

$G_3 = (V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S_3 \to S_1 | S_2\}, S)$

assumed to be disjoint

$L(G_3) = L(G_1) \cup L(G_2)$

if V_1, V_2 aren't disjoint then just rename them

2. Intersection - Counter Example

$P = \{a^i b^j c^j | i,j \geq 0\}$

$Q = \{a^i b^i c^j | i,j \geq 0\}$

$P \cap Q = \{a^i b^i c^i | i \geq 0\}$ which is not a CFL

3. Concatenation

$G_1 = (V_1, \Sigma_1, R_1, S_1)$ $G_2 = (V_2, \Sigma_2, R_2, S_2)$

$G_3 = (V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S_3 \to S_1 S_2\}, S)$

V_1, V_2 assumed to be disjoint

$L(G_3) = L(G_1) \cup L(G_2)$

4. Complementation - No

Assume for contradiction

L_1, L_2 be CFL $\Rightarrow L_1^c, L_2^c$ also CFL

$L_1 \cup L_2$ is CFL $\Rightarrow (L_1^c \cap L_2^c)^c$ is a CFL but $L_1^c \cap L_2^c$ is not neccesarily a CFL.

5. Kleene Closure

$G_1 = (V_1, \Sigma_1, R_1, S_1)$

$G = (V_1, \Sigma_1, R_1 \cup \{S \to \varepsilon \mid S_1 S\}, S)$

Is the intersection of a CFL with a regular language also a CFL?

Let $M_1 = (Q_1, \Sigma, q_1, F_1, \delta_1)$: DFA

$M_2 = (Q_2, \Sigma, \Gamma, q_2, F_2, \delta_2)$: PDA

Define $M = (Q, \Sigma, \Gamma, q, F, \delta)$ where

$Q = Q_1 \times Q_2$

$q = (q_1, q_2)$

$F = F_1 \times F_2$

$\forall a \in \Sigma_1, b, c \in \Gamma$ $\alpha \in Q$, $\beta \in Q_2$

we have

$$\delta((\alpha, \beta), a, b) \ni (\alpha', \beta', c)$$

where $(\beta', c) \in \delta_2(\beta, a, b)$ and

$$\alpha' = \begin{cases} \alpha & \text{if } a = \varepsilon \\ \delta_1(\alpha, a) & \text{o/w} \end{cases}$$

Claim

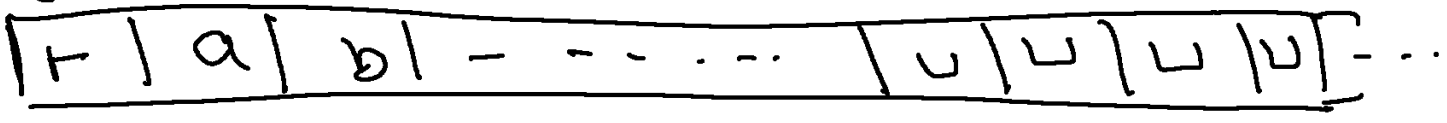
$$L(M) = L(M_1) \cap L(M_2)$$

Turing Machines (always going to be deterministic)

A turing machine is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where

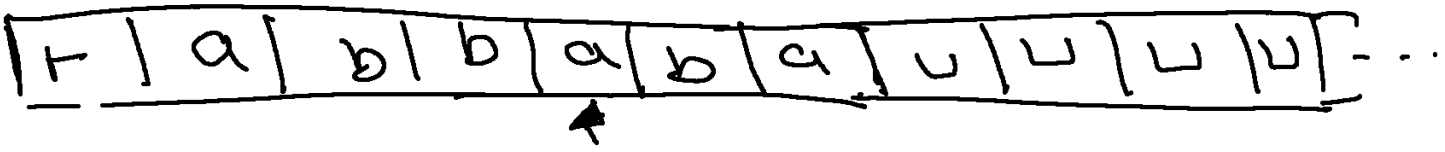
1. Q is a finite set of states
2. Σ is the input alphabet not containing blank symbol $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state where $q_{accept} \neq q_{reject}$

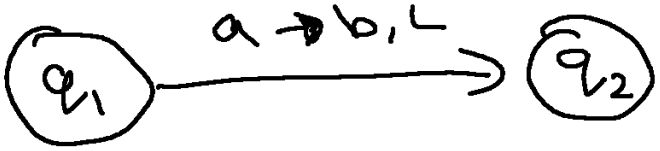
(The input is presented on the tape)

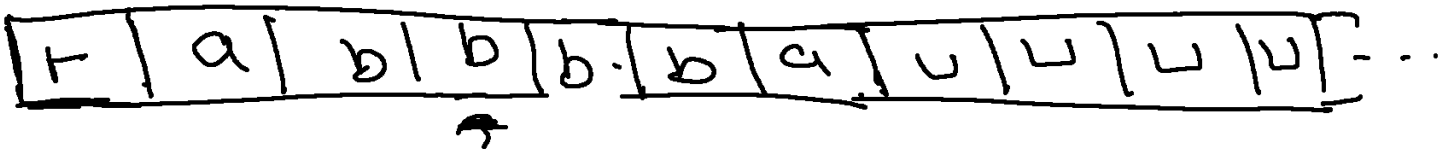


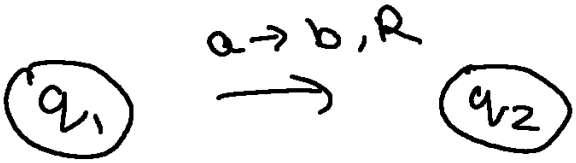
denotes the start

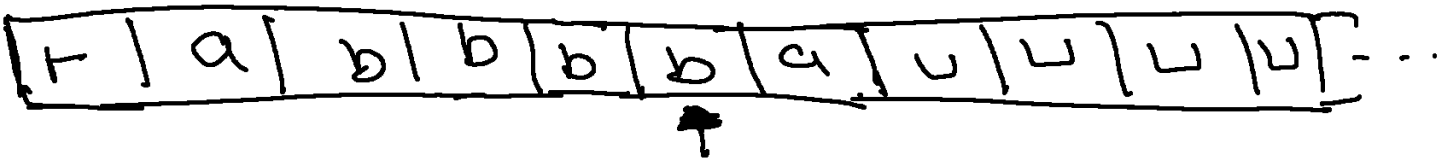
denotes empty memory

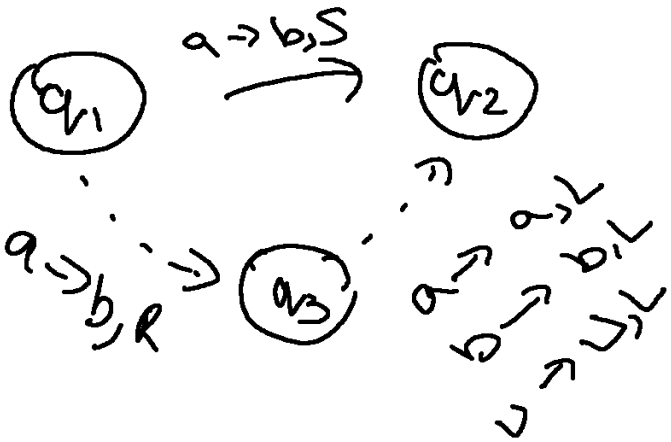

⊢ a b b a b a ∪ ∪ ∪ ∪ ...


a → b, L
q1
q2


⊢ a b b b b a ∪ ∪ ∪ ∪ ...


a → b, R
q1
q2


⊢ a b b b b a ∪ ∪ ∪ ∪ ...

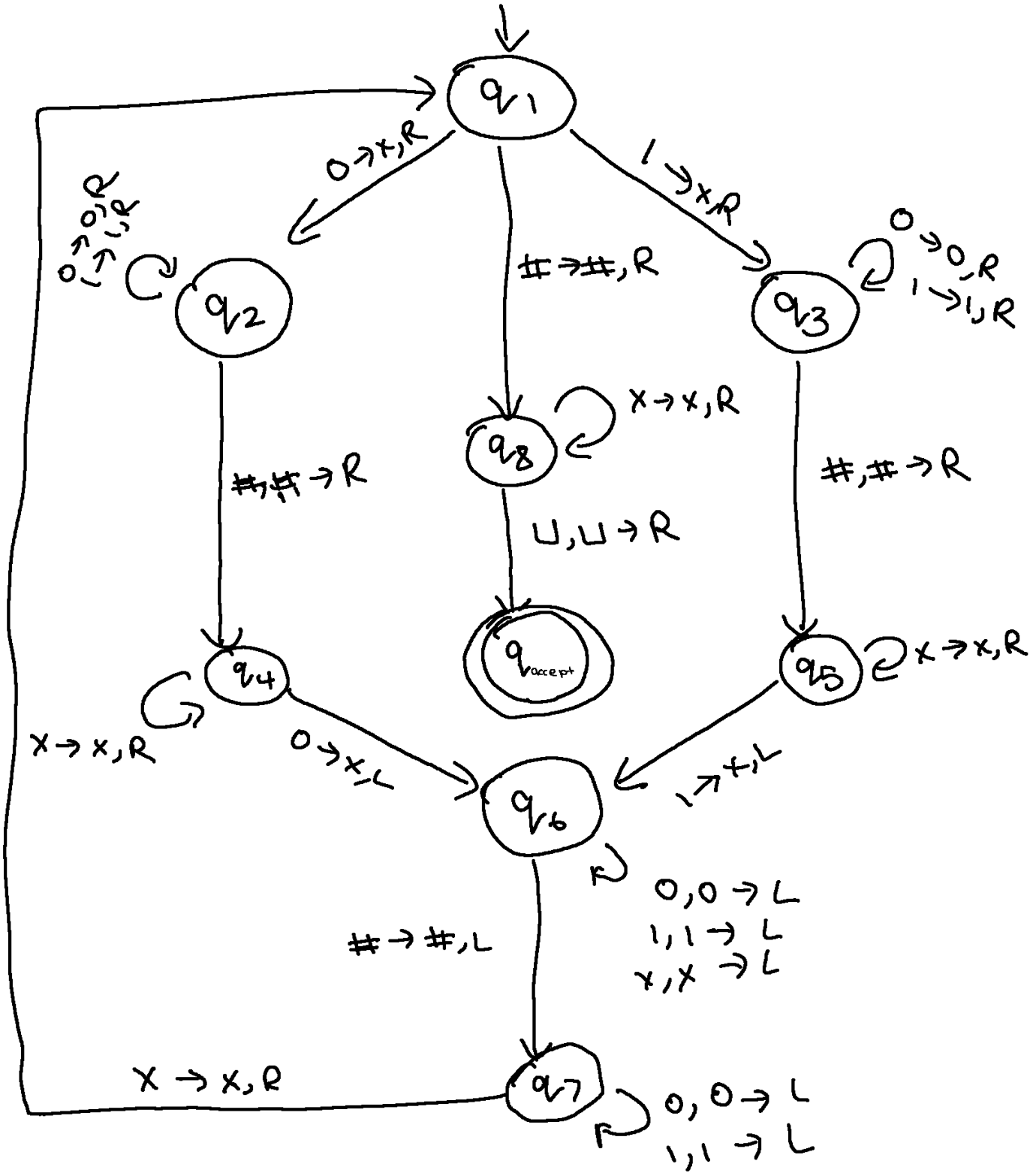

a → b, S
q1
q2
q3
a → b, R
a → a, L
b → b, L
∪ → ∪, L

$L = \{w \# w \mid w \in \{0,1\}^*\}$

M_1 = " On the input string w:

1. Zig-Zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they don't 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 symbols remain, reject; otherwise, accept."



Question could ask for informal or a formal description of a turing machine.

$L = \{0^{2^n} \mid n \geq 0\}$

M_2 = "On input string w"

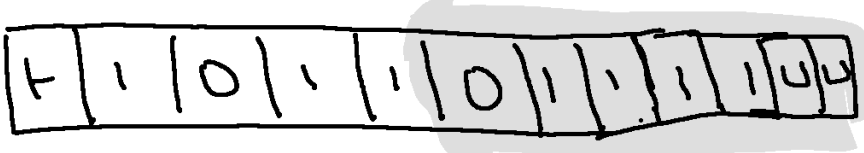
1. Sweep left to right across the tape, crossing off every other 0
2. If in stage 1. The tape contained a single 0, accept.
3. If in stage 1 the tape contained more than a single 0s and the number of 0s was odd reject
4. Return head to left end of tape.
5. Go to stage 1.

Configurations in Turing Machines

$X = (u, q, v)$

current state

uv is the string in tape, where head points to first symbol of v.



in this case $(\vdash 1011, q, 0111)$

Configuration $X = (u, q, v)$ yields Configuration $Y = (u', p, v')$ if the toring machine can legally go from X to Y in one step.

Start Configuration $= (\vdash, q_0, w)$
Accepting Configuration $= (u, q_{accept}, v)$
Rejecting Configuration $= (u, q_{reject}, v)$

Accepting or Rejecting Configuration
$=$
Halting Configuration

Consider a seqence of configuration $C_1, \ldots, C_r$ st C_1 is the start configuration and for $i = 1, \ldots, r-1$ the configuration C_i yields C_{i+1}

This is called the Run of M on w

for a run to be accepting C_r has to be the accepting configuration

(runs stops at halting configuration)

$$L(M) = \{ w \mid w \text{ is accepted by } M \text{ or the run of } w \text{ on } M \text{ is an accepting run} \}$$

(/Recursively enumerable)

A language L is turing recognisable if it is accepted by some turing machine.

Turing machine

i.e. $\exists\, n\, M$ st $\forall\, w \in L$, w is accepted by M

no string not in the language is accepted by the turing machine.

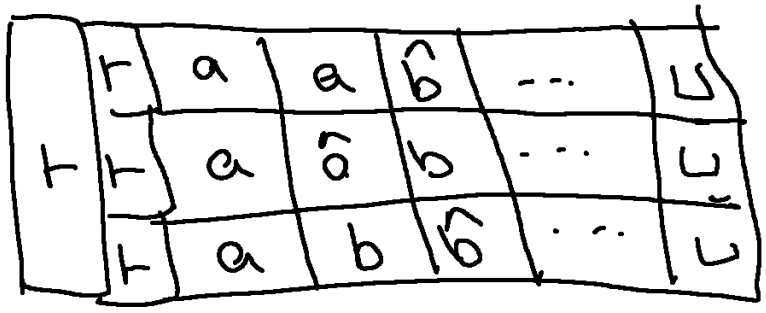
A decider / total TM is a Turing Machine that on any input either halts & rejects or halts & accepts.

For a decider M, L(M) is the language decided by M.

A language L is turing decidable (/Recursive) if it is the language accepted by some decider.

Robustness of Turing Machines as a model of Computation

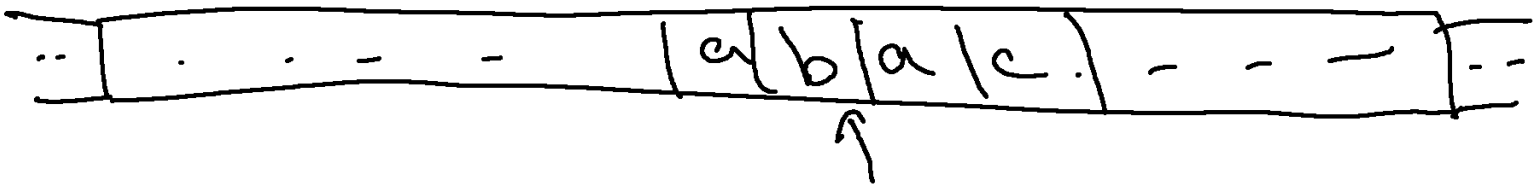
What if the machine has 3 tapes?
We can simulate three tapes using one.



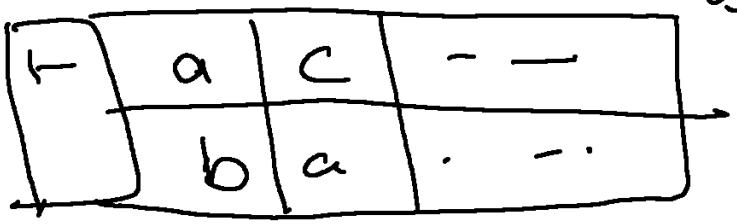
^ - denotes pointer

use 3 tapes is faster

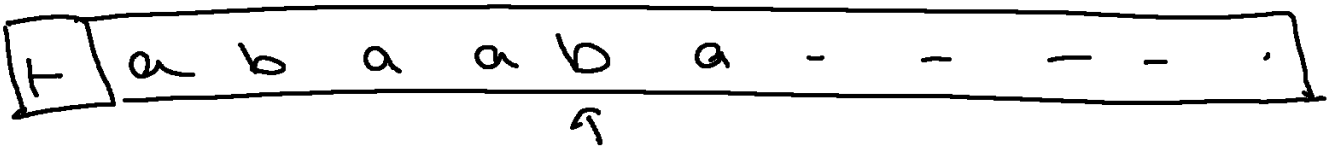
What if tape was infinitely long in both directions

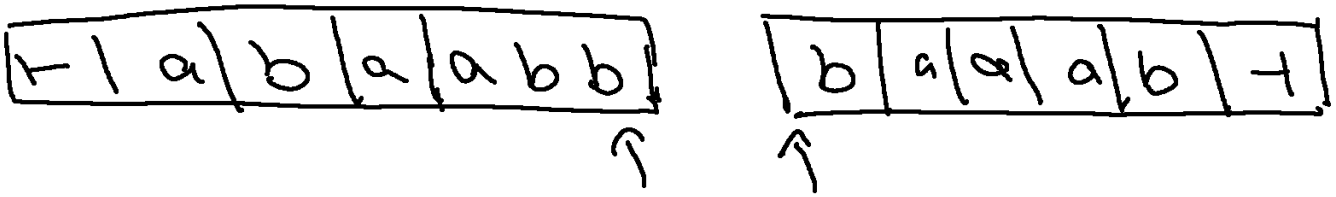


Just fold here



Equivalence with 2-stack PDAs





moving pointer left is equivalent to popping from Right stack and pushing that onto left stack.

Enumeration Machine

Same as a Turing Machine, but has a dedicated output tape.

- Finite set of states, with one special enumeration state
- Read / write work tape
- Write only output tape

Always starts with blank work tape

When it enters an "enumeration" state, the word contained in the output tape is said to be "enumerated".

The machine then erases the output tape, sends the write only head back to the beginning of the output tape, leaves the work tape

untouched, and continues.

L(E) = the language enumerated by the enumerator M

$= \{w \mid w \text{ is enumerated by } M\}$

> L is Turing Recognizable iff some enumerator enumerates it

Recursively Enumerable

=

Turing Recognizable

Enumerators vs Turing Machines

Given an enumerator E, how do we convert it to a Turing Machine M accepting L(E)?

M = "On input w:

1. Run E. Every time that E outputs a string compare it with w.
2. If w ever appears in the output of E, accept."

Note: If $w \notin L(E)$, then M may never halt.

Given a Turing Machine M, Construct Enumerator E st $L(E) = L(M)$.

E = 1. Repeat the following for $i = 1, 3$

2. Run M for i steps on each input $s_1, s_2, \ldots s_i$
3. If any computations accept print out the corresponding input s_j

This way, $\forall s, M$ runs for a sufficient amount of steps on s eventually.

Turing Machines = Recursively Enumerable

A universal turing machine U takes as input the string "$Enc(M) \# w$" and simulates M on w.

we write $\langle M, w \rangle$

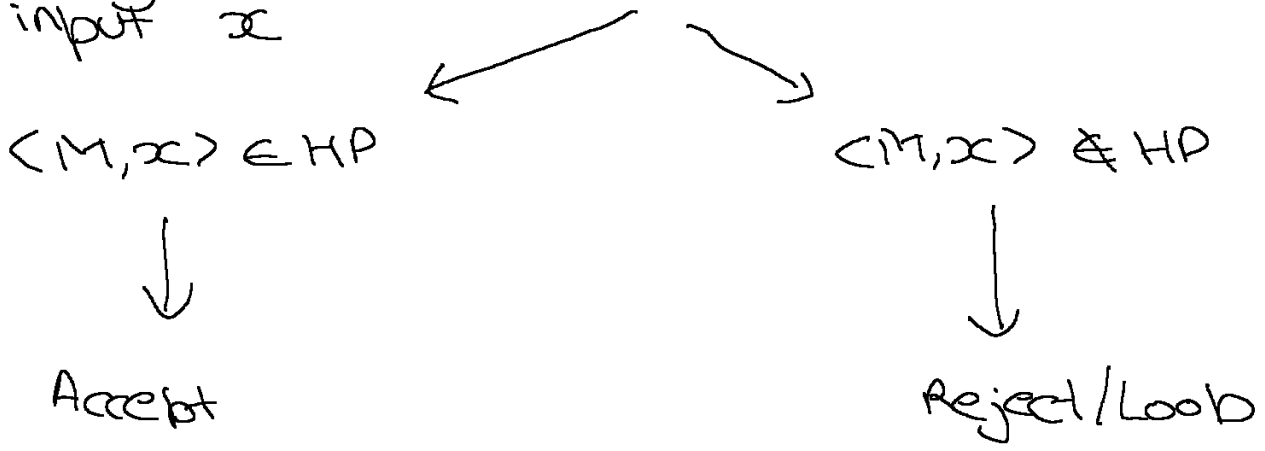
The Halting and Membership Problem

$HP = \{\langle M, x \rangle \mid M \text{ halts on } x\}$
$MP = \{\langle M, x \rangle \mid M \text{ accepts } x\}$

Note M accepts $x \iff x \in L(M)$

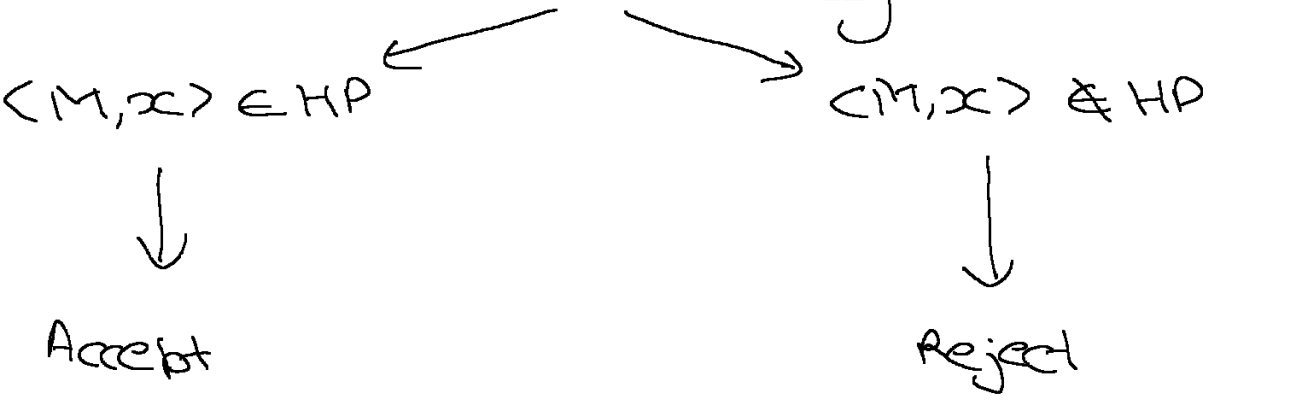
Is HP Turing-Recognizable?

Turing Machine U: just simulate M on input x



So HP = L(U), Yes.

Is there such a Turing Machine U?



No.

Proof

Treat every binary string as representation of some Turing Machine

	ε	0	1	00	- -	—
M_ε	H	L	H	H		
M_0	L	L	H	H		
M_1	L	H	H	L	- - - - -	-
M_{00}	L	H	L	H		
M_{01}	H	L	H	L		
'	'		'			
'			'			
			'			
			'			
'			'			

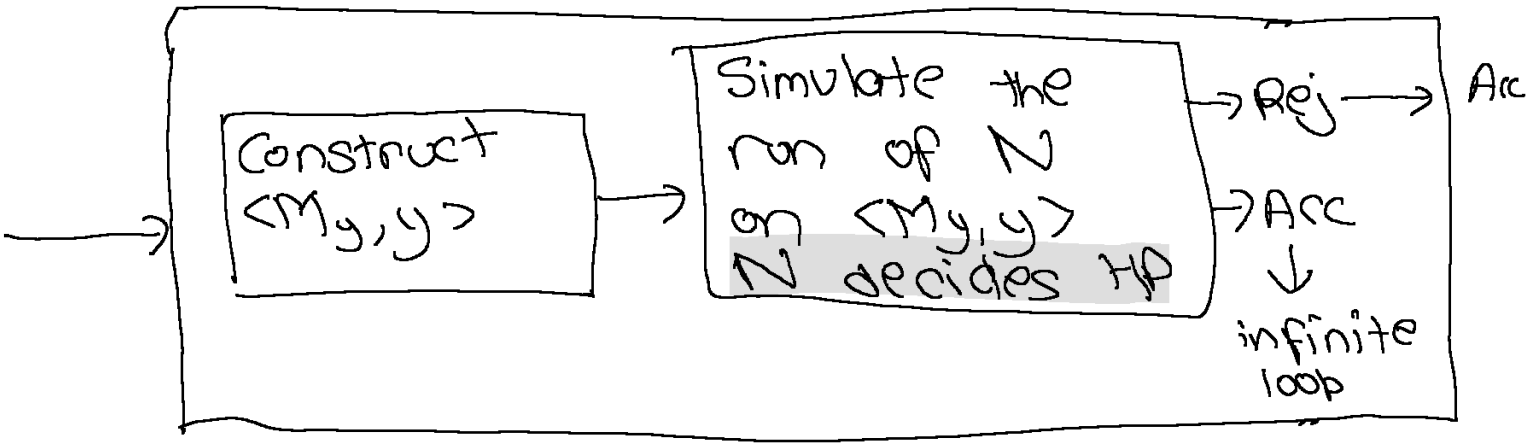
Every Turing Machine appears as a row (Every TM can be encoded as a binary string thus above statement)

Suppose ∃decider (N) for HP, then we can construct a universal Turing Machine K which is not present in this table → ⊥

	ε	0	1	00	- -	—
M_ε	H	L	H	H		
M_0	L	L	H	H		
M_1	L	H	H	L	- - - - -	-
M_{00}	L	H	L	H		
M_{01}	H	L	H	L		
'	'		'			
'			'			
			'			
			'			
'			'			

On input $\langle M, x \rangle$ the machine N
↳ halts and accepts if it determines that M halts on input x
↳ halts and rejects if it determines that M does not halt on input x.

Define the Turing Machine K which on input Y, writes $\langle M_Y, y \rangle$ on the input tape, writes the encoding of N on the description tape, simulates the run of the string $\langle M_Y, y \rangle$ on N, and finally
↳ halts and accepts if N rejects
↳ goes into an infinite loop in N accepts.



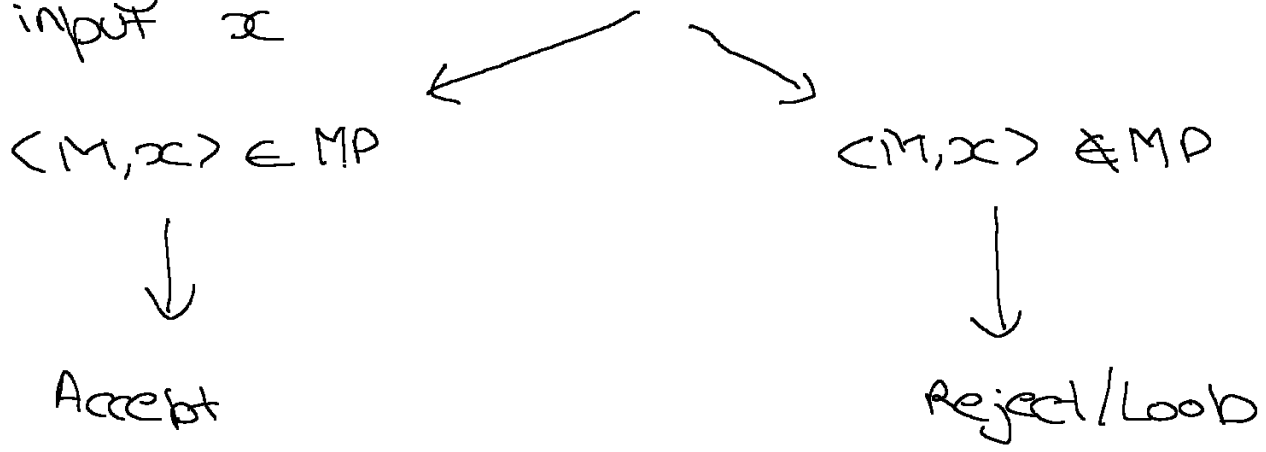
K

Intuition: K invents behaviour along the diagonal

=> K is not present in this table → ⊥

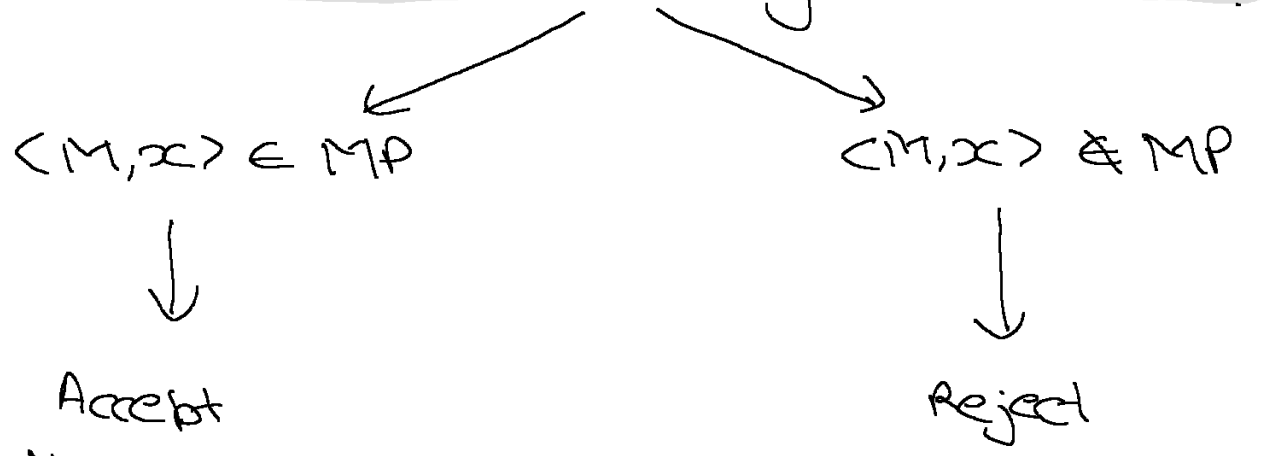
Is the Membership Problem Turing Recognizable?

Turing Machine U: just simulate M on input x



So MP = L(U), Yes.

Is there such a Turing Machine U?

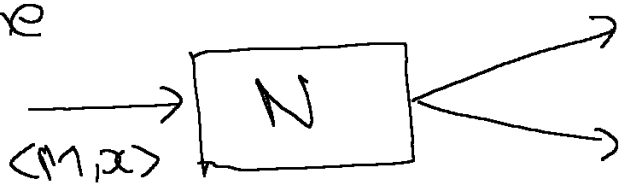


No.

Proofs (Can do diagonalization)(Gonna do reduction)

Suppose MP is decidable and let N be a decider for MP. We will show that N can be used to construct a decider K for HP, which would be a contradiction

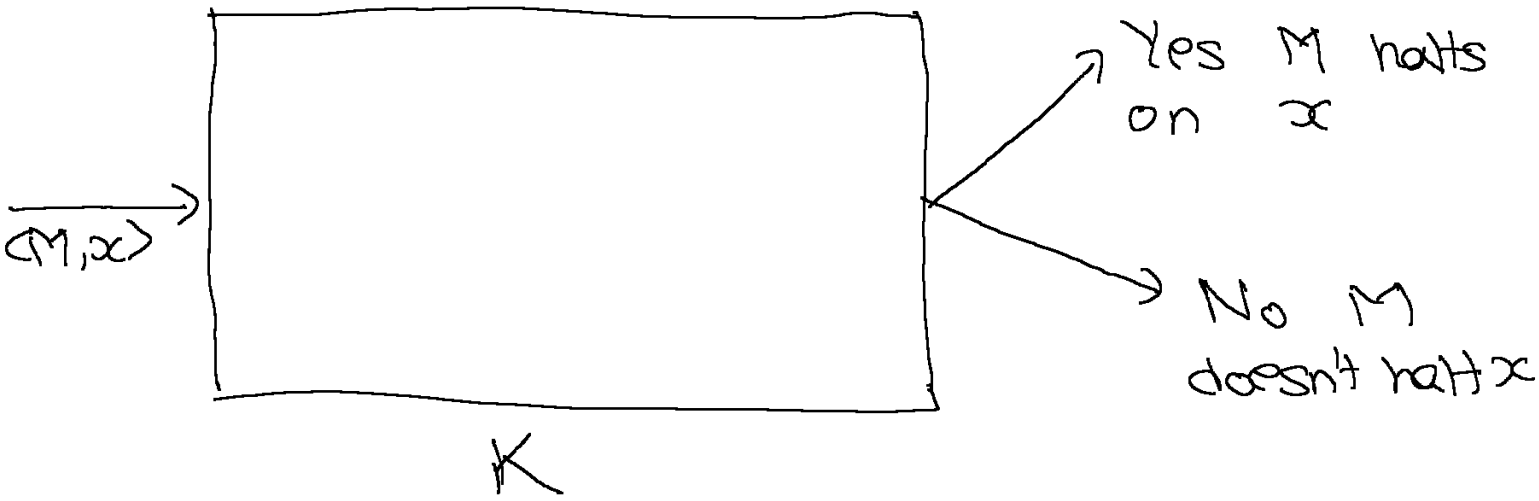
Assume



$x \in L(M)$

$x \notin L(M)$

Want



Idea

Show $M \longmapsto M'$ st.

$(M,x) \in HP \iff (M',x) \in MP$

Define a TM K as follows: On input $\langle M,x \rangle$

1. Construct a new machine M' obtained from M as follows
   - Add a new accept state to M
   - Make all incoming transitions to old accept and reject state go to the new accept state
2. Simulate the assumed decider N on input $\langle M', x \rangle$ and accept iff N accepts.

Proof is a reduction from halting problem to the membership problem to show that the membership problem is undecidable.

Reductions
for $A \subseteq \Sigma^*$, $B \subseteq \Delta^*$
we say that $A \leq_m B$ if

$\sigma : \Sigma^* \to \Delta^*$ (is called a mapping reduction from A to B if)

1. $\forall x \in \Sigma^* \quad x \in A \Leftrightarrow \sigma(x) \in B$
2. σ is a computable function

> $\sigma : \Sigma^* \to \Delta^*$ is a computable function if there is a decider that on input x, halts with $\sigma(x)$ written on the tape

Another way of reducing HP to MP

Consider $\langle M, x \rangle$ instance of HP
Define TM M'_x as follows

$M'_x(y)$
- simulate M on x. If M halts then accept

Claim: $\langle M, x \rangle \in HP$ iff $\langle M', x \rangle \in MP$

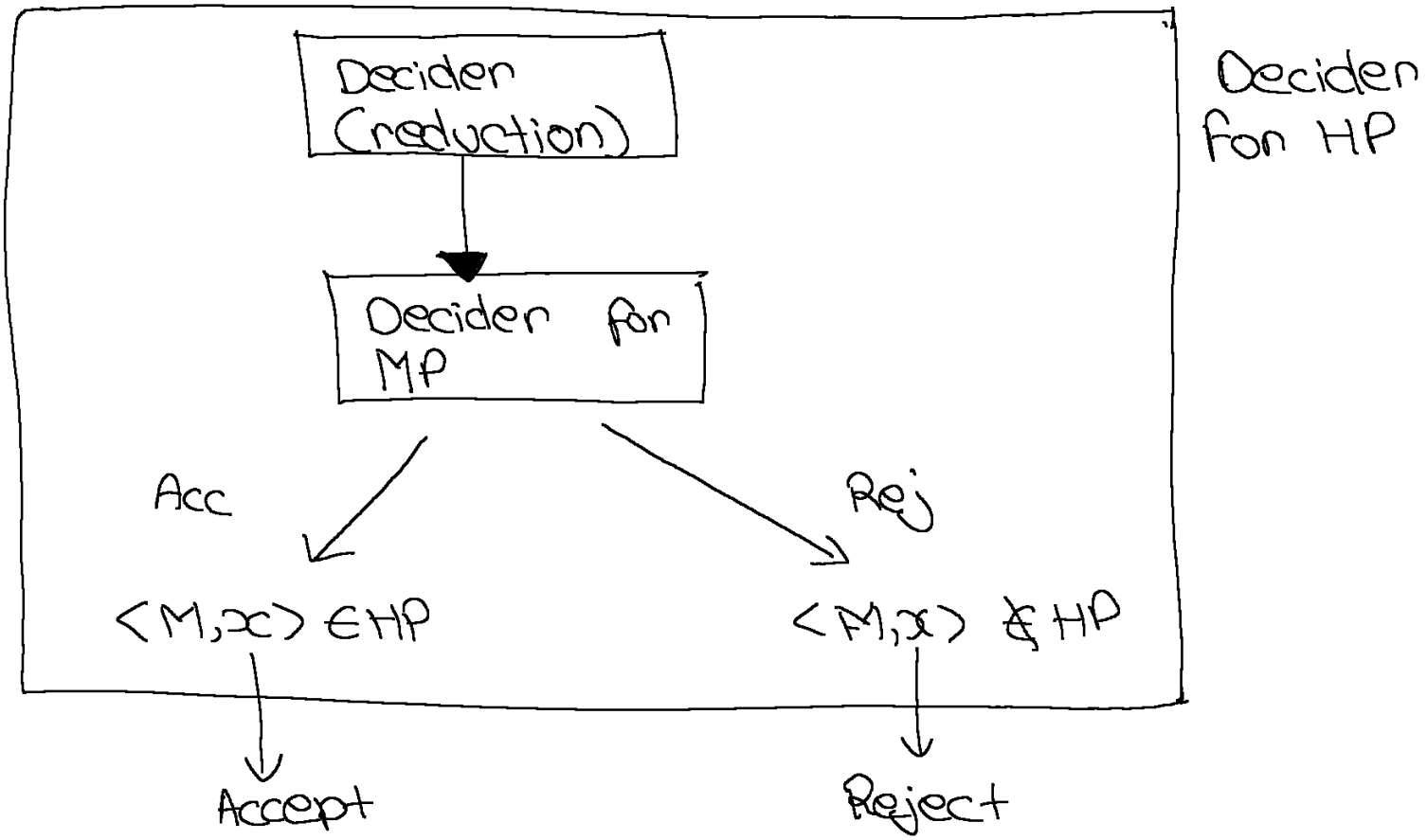
Proof (⇒) $\langle M,x\rangle \in HP$

(⇐) M halts on x

(⇐) $M'_x(x)$ accepts

(⇐) $\langle M', x\rangle \in MP$

we want to know if $\langle M,x\rangle \in HP$



More Interesting Problems

I Σ- Acceptance $= \{\langle M\rangle \mid \Sigma \in L(M)\}$

II $\exists$ - Acceptance $= \{\langle M\rangle \mid L(M) \neq \emptyset\}$

III $\forall$ - Acceptance $= \{\langle M\rangle \mid L(M) = \Sigma^*\}$

Claim $\langle M,x\rangle \in HP$ iff $\langle M'_x\rangle \in \Sigma$-Acceptance

$\langle M,x\rangle \in HP$ iff $\langle M'_x\rangle \in \exists$-Acceptance

$\langle M,x\rangle \in HP$ iff $\langle M_x\rangle \in \forall$-Acceptance

If $A \leq_m B$ and B is decidable then A is decidable

Proof
Let M be the decider for B
f be the reduction from A to B

N = "On input w
1. Compute $f(w)$
2. Run M on $f(w)$ and output whatever M outputs"

Clearly, N is a decider for A.

If $A \leq_m B$ and B is Turing-Recognizable then A is Turing Recognizable

Same proof, replace "decider" with "recognizer"

Theorem: A Language L is decidable iff L and $\overline{L}$ are both Turing Recognizable.

Proof ($\Rightarrow$) If L is decidable, then a decider for L also function as a recogniser for L. For $\overline{L}$, use the same decider and complement the answer.

($\Leftarrow$) If L and $\overline{L}$ are Turing Recognisable (with Recognizers P and Q), then the following decider M is a decider for L:

Step 1: On input x, run P and Q simultaneously with input x. if P accepts, halt and accept. If Q accepts halt and reject.

If A is turing recognisable $A \leq_m \bar{A}$, then A is decidable.

Claim: $A \leq_m \bar{A}$ implies that $\bar{A} \leq_m A$ under the same reduction.

Proof of Claim: Excercise

Using Claim, Reduction properties and Theorem, True.

Q) T/F
$L = \{\langle G \rangle \mid G$ is an ambigious CFG$\}$ is recursively enumerable.

Ans) True

M(<G>)
for $i = 1, 2, \ldots$
- Compute all possible parse trees of G of height at most i
- Write down all generated words in this iteration
- If some word has been generated by two parse trees then accept.

Q) T/F
$L = \{\langle P \rangle \mid P$ is PDA st $L(P) \neq \emptyset\}$ is decidable

Ans) Given CFG, is $L(G) \neq \emptyset$

Exhaustively do
if $\exists$ variable X and a rule $X \to \gamma$ where every variables in γ is unmarked then mark X

Claim
$L(G) \neq \emptyset$ iff start variable is marked

Is there a decider D, which given a DFA M, decides whether M accepts at least 1 string with more 0's than 1's?

Let P := PDA for $\{ w \in \{0,1\}^* \mid n_0(w) > n_1(w) \}$

$L(P) \cap L(M)$ is a CFL
$=$
$\emptyset$?

Are the following languages decidable?

$L_1 = \{\langle M \rangle \mid M$ is a TM and $L(M)$ is countable$\}$

$L(M) \subseteq \Sigma^*$ which is countable thus
$L_1 = \{\langle M \rangle \mid M$ is a TM$\}$

$L_2 = \{\langle M, x \rangle \mid M$ is DFA that halts on $x\}$

DFA always halts thus $L_2 = \{\langle M, x \rangle \mid M$ is a DFA$\}$

$L_3 = \{\langle M \rangle \mid \exists x, y \in \Sigma^* \; x \in L(M) \text{ or } y \notin L(M)\}$

$L_3 = L_1$ trivially

$L_4 = \{\langle M \rangle \mid \exists TMs \; M_1, M_2 : L(M) \subseteq L(M_1) \cup L(M_2)\}$

for every TM $L(M) \subseteq L(M) \cup L(M)$
thus $L_4 = L_1$

$L_4' = \{\langle M \rangle \mid \exists \overset{\text{distinct}}{\hat{}} TMs \; M_1, M_2 : L(M) \subseteq L(M_1) \cup L(M_2)\}$

$L(M_1) = \Sigma^* \quad L(M_2) = \emptyset$
thus $L_4' = L_4$