1. Course outline and motivation

On the course:

- Regular Expressions
- Finite Automata
- Context Free Grammars
  - ↳ Programming languages
  - ↳ NLP
- DTDs

Limitations of software

- Decidability
- Intractability

AT gives you the tools to navigate these issues

Also:

- Proofs e.g. inductive proofs
- Abstract models and constructions.
- Pushdown Automata

Course outline:

- Regular languages
  - ↳ FA
  - ↳ NDFA
  - ↳ Regex
  - ↳ Algorithms to decide questions about Languages

- Context free languages
  - Context free grammars
  - Pushdown Automata
  - Decision and closure

- Recursive and enumerable languages
  - Turing Machines
  - Decidability

2. Finite Automata

What is a finite Automata?

- It is a sound system (mathematical)
- Remembers only a finite amount of info
- Automatas are comparable and can be analysed
- Info is represented by states
- Inputs cause state changes
- The rules that govern the state changes are called transitions.

Why study FA

- Used for design and verification.
- Text-processing
- Part of compilers
- Other uses

Example: Tennis

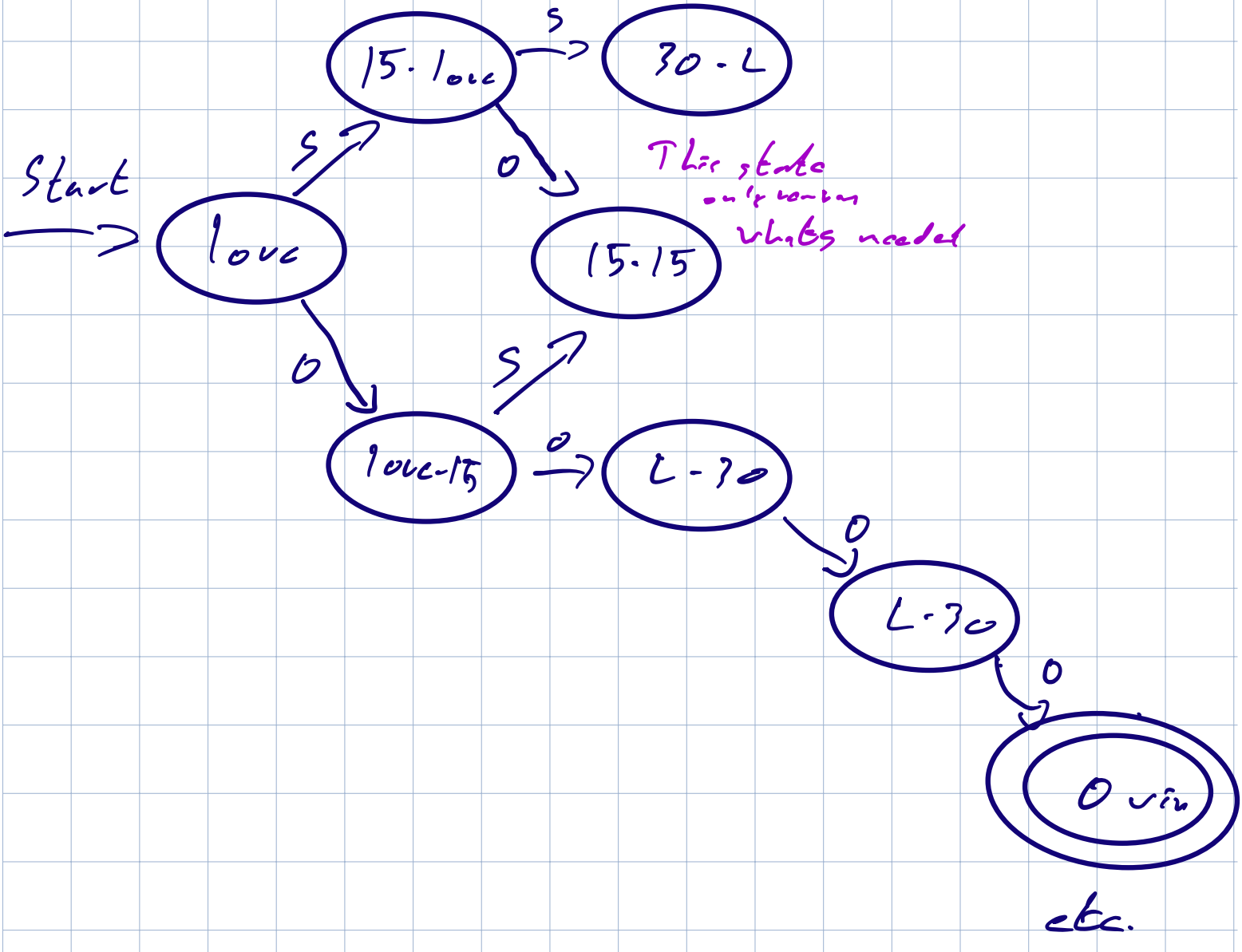
- Match 3-5 sets
- Sets 6 games

Game:

- 1 person serves entire game
- Win w/ 4 points min
- Win w/ 2 point diff

- Inputs are S = Server wins, O = opponent wins

FA are represented by graphs



The entire game of Tennis has 2 game winning states and many cycles

Acceptance of inputs

- Given a sequence of inputs (input string) begin at the start state and follow the transition from each input in turn
- The input is accepted if the sequence finishes in an accepting state
- Given a string of Tennis input symbols which describe a game e.g. sosososososs. We follow the state-diagram which shows The Server winning in an accepting state

Language of an Automaton

- The job of FA is to process a sequence of inputs and accept or reject it.
- The set of strings accepted by an automaton A is called a language. $L(A)$
- The symbols used in a language is called a alphabet
- The $L(A)$ is only made up of strings which finish in an accepting state.

  'ssss', 'oooo', etc are in L(Tennis)

  's', 'o', 'osos' are not

3 Deterministic Finite Automata.

Alphabets, Strings and languages

- Alphabet is the symbols used for input.
- Strings are sequences of symbols in the alphabet
- languages are the set of strings accepted by a F

Alphabet: finite set of symbols

String: Sequence of symbols

String over an Alphabet Σ is a list where each symbol in Σ

Σ^* set of all strings over alph Σ

ϵ empty string

language: $\{0,1\}^*$ language of strings including ϵ that can be made from 0's, 1's

Context of lang determines the type

languages are sets of strings

they can be finite sets or infinite sets but they must be made from finite alphabets

language is a subset of Σ^* for Alb Σ

e.g. set of 0's, 1s with no 2 consecutive 1's (fibonacci)

Deterministic Finite Automata:

- A formalism (formal definition) for defining languages consisting of
  1. A finite set of states: Q
  2. A finite input Alphabet: Σ
  3. A transition function: δ
  4. A start state: q_0 in Q
  5. A set of final states: $F \subseteq Q$
     (accepting)

Transition function:

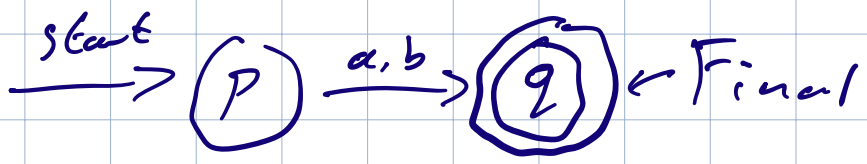
- Takes 2 arguments:
  - What state we're in
  - The input symbol
- $\delta(q_i, a)$ = The resulting state of input a at state q_i
- Note: There is always a next state - add a dead state if no transition
  DFA always has a result
  Dead state is one that is not accepting and does not transition away from itself
  Dead-end

Deterministic means there is a unique transition for EVERY state and input symbol

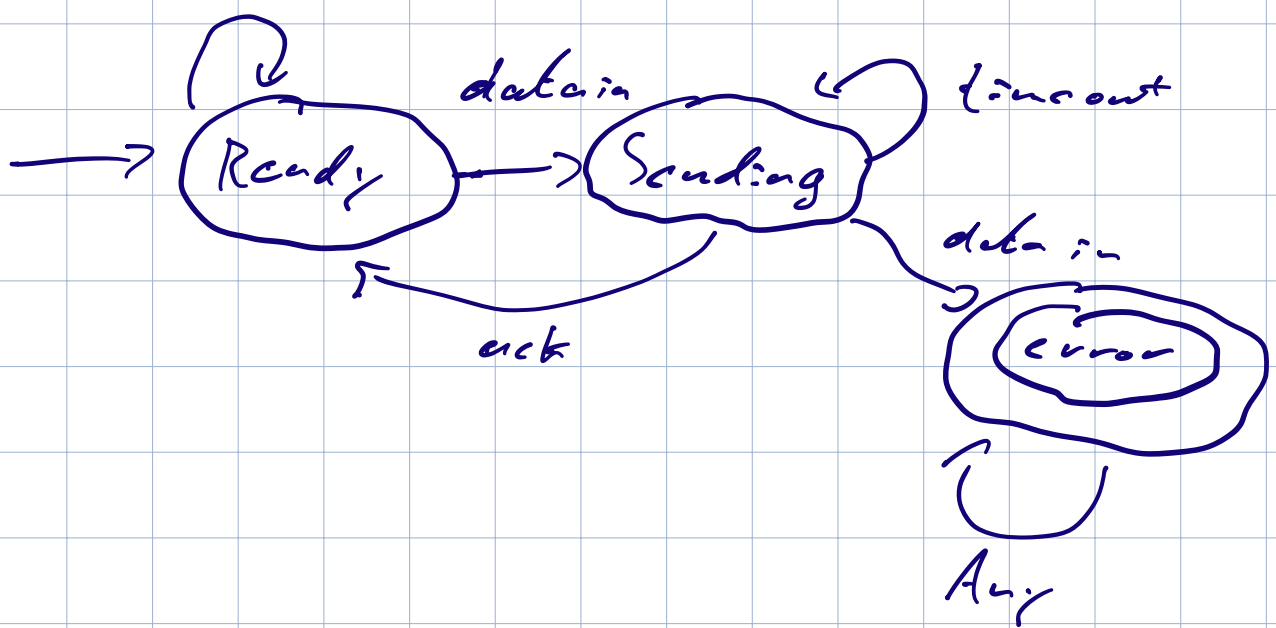
In tennis game the accepting states do not have any transitions out So we would add a dead state to handle any additional inputs

Any FA can be represented by a graph

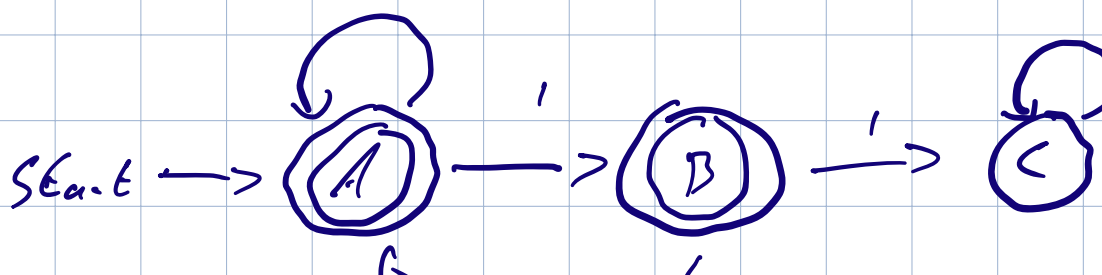
- Nodes = states
- Transitions are arcs



Sending data protocol



Strings w/out 11



dead state

0

we have seen 1, 1

Transition Table

* = final

		0	1
*	A	A	B
*	B	A	C
	C	C	C

↑ states

Conventions:

w, x, y, z are strings

a, b, c, .. are single input symbols

Extended Transition

- The effect of a string of inputs on a DFA by extending δ to a state and a string.
- Intuition: Extended δ is computed for q, $a_1, a_2 \ldots a_n$ by following the transition graph and the transition arc for each input in sequence

Inductive definition of extended δ

An induction on the length of the input string

Basis: $\delta(q, \epsilon) = q$

Induction on base case: $\delta(q, wa) = \delta(\delta(q,w), a)$

Remember w = string, a = symbol

This is essentially a recursive representation of working through symbols in w

	0	1
A	A	B
B	A	C
C	C	C

$\delta(B, 01) = \delta(\delta(B,0),1)$
$= \delta(A,1)$
$= B$

Extended δ is sometimes given a hat $\hat{\delta}$
we don't distinguish between δ or $\hat{\delta}$
because they agree on single symbols

Language of a DFA

- Automata of all kinds define languages
- If A is an Automaton, $L(A)$ is its language
- For DFA: $L(A) \subseteq \Sigma^*$ that takes A from start to accepting states.

Formally:

$L(A)$ = set of strings w such that $\hat{\delta}(q_0, w)$ is in F

Set former: It describes sets

For the running example a set former is:

$\{ w \mid w \text{ is in } \{0,1\}^* \text{ and } w \text{ does not have two consecutive 1's} \}$

strings of set w

Proof of Set equivalence:

- Proving set A = set B using their descriptions

Set S is: The language of this DFA

Set T is: Set of strings of 0's, 1's w/out consecutive 1's

To prove $S = T$ we need to show $S \subseteq T$, $T \subseteq S$

1. if w in S, then w in T
2. if w in T, then w in S

Part 1: $S \subseteq T$

To prove: if w in DFS, then w has no consecutive 1's

- Proof is an induction on length of w
- Important Trick: Expand the inductive hypothesis H to be more detailed than the statement you are trying to prove

Inductive Hypothesis in 2 parts:

1. If $\delta(A, w) = A$ then w has no consecutive 1's and does not end in 1
2. If $\delta(A, w) = b$ then w has no consec... 1's but does end in 1

$|w|$ = length

Basis: $|w| = 0$, i.e., $w = \epsilon$

- (1) holds because there are no 1's at all
- (2) holds vacuously since $\delta(A, \epsilon)$ is not B

Important: The 'if' part of 'if ... then' is false then the statement is true

Inductive step

- Assume (1), (2) are true for strings shorter than w where $|w|$ is at least 1
- because w is not ϵ we can write $w = xa$ where a is the last symbol, x is the string that proceeds it. (possibly none)
- Since x is shorter than w we assume the IH for x because we're not solving for x we can allow it to be true

Now we prove both statements for $w = xa$

(1) If $\delta(A, w) = A$ then w has no consecutive 1's, does not end in 1

$\delta(A, x) = A$ is true $\therefore$ x ends in 0

$\delta(A, xa) = A$

$\delta(\delta(A, x), a)$ w does not end in 1

$\delta(A, 0) = A$ $\therefore a = 0$

Since $\delta(A, w) = A$

$\delta(A, x) = A \mid B$

$\delta(A \mid B, a) = A$ then a must be 0

2. $\delta(A, w) = B$

$\delta(A, x) = A$ or B

if B then w cannot end in 1 or 0 so x does not end in 1

So $\delta(A, x) = A$, x ends in 0

$\therefore \delta(A, xa) = A$ iff x ends in 0, $a = 1$

2. Prove $T \subseteq S$

Prove that if w has no 11's it is accepted by the DFA.

We can prove the contra positive:

If w is not accepted by S then it has 11

Contrapositive takes the Hyp if 'x. then y' is equivalent to if not y then not x

X: w has no 11's

Y: w is accepted by DFA

Y': w is not accepted by DFA

X': w has 11's

Because every state in a DFA has a unique transit... for every input each w arrives at exactly 1 state

Only way for w to not be accepted is to get to C.

For $\delta(A, w) = C$

$w = x1y$ where $\delta(A, x) = B$

x must end in 1 $\therefore$ $w = x1y$ has consec 1's

$\delta(A, x) = B$

$x = z1$ for some z

$\therefore$ $w = z11y$, has 11

Regular languages:

- A language L is <u>regular</u> if it is the language accepted by some DFA.

  Note: DFA must accept only strings in L, no other
- Some languages are not regular

e.g. non-reg language

$L_1 = \{ 0^n 1^n \mid n \geq 1 \}$ DFA does not remember the count as part of its state $\therefore$ cannot verify 0^n or 1^n

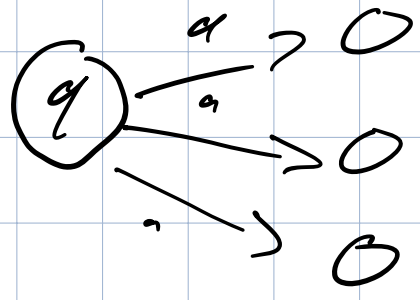
Another e.g. $L_2 = \{ w \mid w \text{ in } \{(,)\}^* \text{ and } w \text{ is balanced} \}$

Same as above but $0 = ($, $1 =)$

Many languages are regular.

4. Non-deterministic Finite automata

NDA transitions: Many states, 1 input



Chessboard

1	~~2~~	3
~~4~~	5	~~6~~
7	~~8~~	9

S	r	b
1	2, 4	5
2	4 6	1 3 5
3	2.6	5
4	7, 8	1 5 7
5	2 4 6 8	1 3 7 9
6	2 8	3 5 9
7	4 8	5
8	4 6	5 7 9
9	8 6	5

	r	b	b
S	2	1	5
1		3	3
	4	5	7
		7	9

rbb reaches 9

T. func of NFA

$\delta(q, a)$ is a set of states

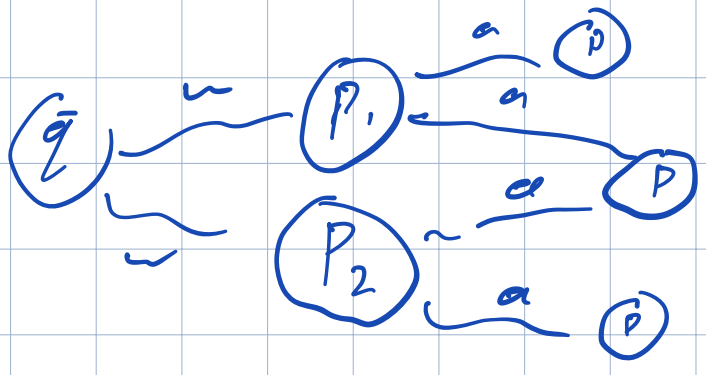
Extend to strings as follows

Basis (base case) for empty string ϵ

$\delta(q, \epsilon) = \{q\}$

Induction

$\delta(q, wa)$ is the union over all states p in $\delta(q, w)$ of $\delta(q, a)$



Follow all paths out of final transition A

DFA, NFA equivalence:

If $\delta_D(q, a) = p$ and the NFA have $\delta_N(q, a) = \{p\}$

Then the NFA is always in a set containing exactly one state - the state the DFA is in after reading the same input.

A DFA is just an NFA w/out any non-determinism

∴ A DFA as an NFA simply has single element set transitions

Subset Construction

2^Q is the powerset of Q

NFA:		DFA
States	Q	States 2^Q (set of subsets of Q)
Alphabet	Σ ⟶	Same
T-func	δ_N	T-func: δ_D
Start	q_0	Start: $\{q_0\}$ set w/ s. of NFA
final states	F	Final: All F

The powerset is the set of all subsets of Q

Important:

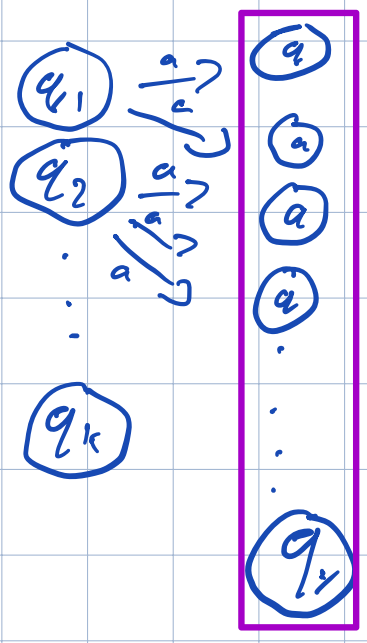
- The DFA states have names that are sets of NFA states
- But as a DFA, an expression like $\{p, q\}$ must be understood to be a single symbol not as a set.
- Analogy: A class of objs whose values are sets of objects from some other class.

T-func of DFA δ_D is defined by

$\delta_D(\{q_1, q_2, \ldots q_k\}, a)$ is the union over all $i = 1, \ldots, k$

of $\delta_N(q_i, a)$

Chessboard T-func:



Union of these states

NFA Table

S	a	b
1	2, 4	5
2	4 6	1 3 5
3	2.6	5
4	7, 8	1 5 7
5	2 4 6 8	1 3 7 9
6	2 8	3 5 9
7	4 8	5
8	4 6	5 7 9
9	8 6	5

DFA Construction

S	a	b
{1}	{2.4}	{5}
{2.4}	{4,6,2,8}	{1,3,5,7}
{5}	{2.4.6.8}	{1,3,7,9}
{1,3,7,9}	"	{5}
{2.4.6.8}	{2.4.6.8}	{1,3,5,7,9}
{1,3,5,7}	"	"
{1,3,5,7,9}	"	"

This is lazy construction because we're only filling states when we're forced to
So we don't have to do each

Proof of equivalence: Subset construction

- Proof is almost a pun
- Show that by induction on $|w|$ that

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

Prove each language is contained in the other

One accepts a string w iff the other does.

$$\delta_D(\{q_1, q_2, \dots q_k\}, a)$$

the union over $i = 1, \dots, k$ of $\delta_N(q_i, a)$

Base case: $\delta_N(q_0, \epsilon) = \delta_D(\{q_0\}, \epsilon) = \{q_0\}$

$w = xa$

IH: inductive hypothesis

Induction:

Assume IH for strings shorter than w

Then prove w

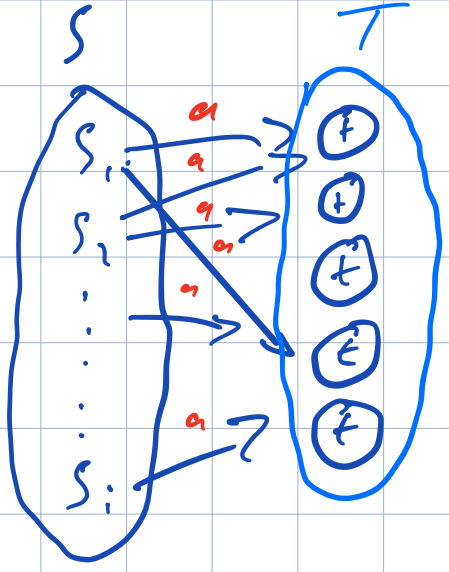
let $w = xa$; IH holds for x

let $\delta_N(q_0, x) = \delta_D(\{q_0\}, x) = S$

let T = The Union over all states p in S of $\delta_N(p, a)$

S is a set of states when δ_D is applied to input x

We apply δ_N to each state p in S. This gives us T



because $\delta_N(\delta_N(q_0, x), a) = \delta_N(q_0, w)$

And $\delta_N(q_0, x) = S$

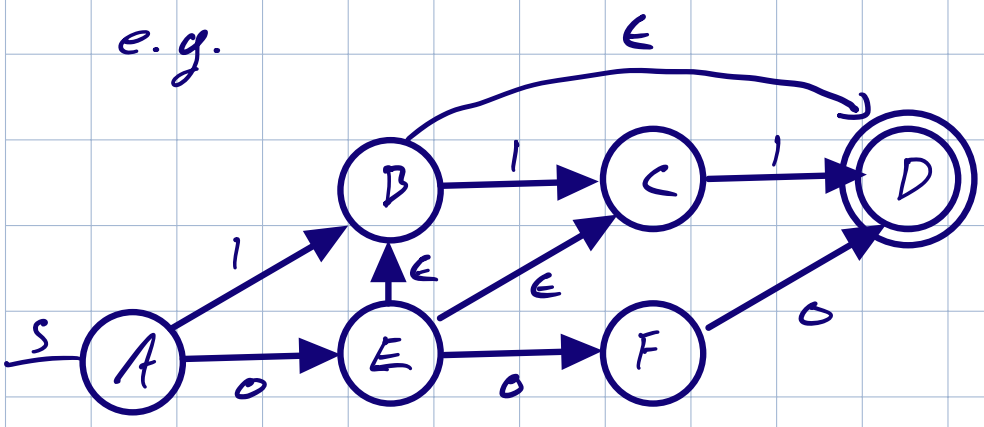
Then $\{\delta_N(\{S\}, a) \supseteq T$

Spontaneous Transitions of NFA on ϵ.

- NFA allows transitions on empty input ϵ
- Done spontaneously w/out looking at input strings
- Can be convenient but still only regular languages are accepted.

We call models that do this ϵ-NFAs

e.g.



T-diagram

		0	1	ϵ
→	A	{E}	{B}	∅
	B	∅	{C}	{D}
	C	∅	{D}	∅
*	D	∅	∅	∅
	E	{F}	∅	{D, C}
	F	{D}	∅	∅

ϵ has a place on the table but is not part of of the input alphabet

Closure of States: CL

- $CL(q)$ = the set of states that you can spontaneously transition to from q via ϵ arcs.

Example from above. Can always transition to same state?

$CL(A) = \{A\}$

$CL(E) = \{B, C, D, E\}$

The Closure of a set of states is the union of the closure of each state. $CL\{A, E\} = \{A, B, C, D, E\}$

If you're in a state q_i you can always stay in q_i.

Extended $\hat{\delta}(q, w)$ - for ϵ-NFA

Intuition:

$\hat{\delta}(q, w)$ is the set of states you can reach from q following path w.

Basis: $\hat{\delta}(q, \epsilon) = CL(q)$

Induction

Let $w = xa$, IH is true for x

$\hat{\delta}(q_0, x) = S$

So ex-della of start state q_0 and string x = a set of states S

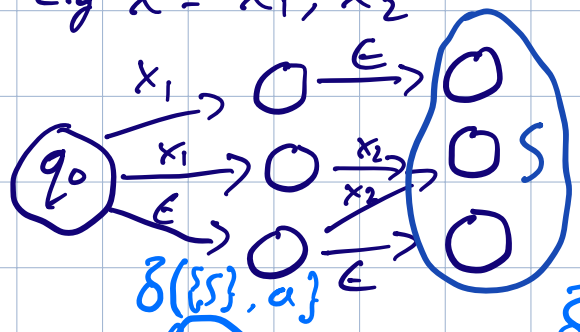
Follow input x and all ϵ-transitions along the way to find S

For ϵ-NFA $\delta \neq \hat{\delta}$

→ because δ, $\hat{\delta}$ are very different

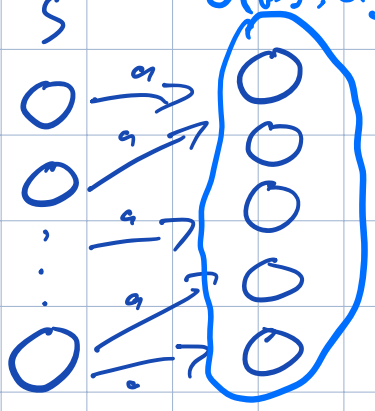
$\hat{\delta}$ includes all possible transitions including ϵ transitions

e.g $X = x_1, x_2$



Now we have S, find all a transitions. Only because a is a non-empty symbol

$\delta(\{S\}, a)$



$\delta(\{S\}, a)$ is the same as $\delta(p, a)$ for all p in S
And the union of $CL(\delta(p, a))$ for all p in S is: $\hat{\delta}(q, xa)$

Example from above:

1. $\hat{\delta}(A, \epsilon) = CL(A) = \{A\}$

2. $\hat{\delta}(A, 0) = CL(E) = \{B, C, D, E\}$

3. $\hat{\delta}(A, 01) = CL(\delta(CL(\delta(A,0)),1)$ $\quad \delta(A,0) = E$

$CL(\delta(CL(\delta(A,0)),1)$

$CL(\delta(CL(\ E\ ),1)$ $\quad CL(E) = \{B, C, D, E\} = S$

$CL(\delta(CL(\ E\ ),1)$

$CL(\delta(\ S\ ,1)$ $\quad \delta(p, 1)$ p in S

$= \{C, D\}$

$CL(\delta(S, 1))$

$CL(\{C, D\}) = CL(C) \cup CL(D)$
$= \{C, D\}$

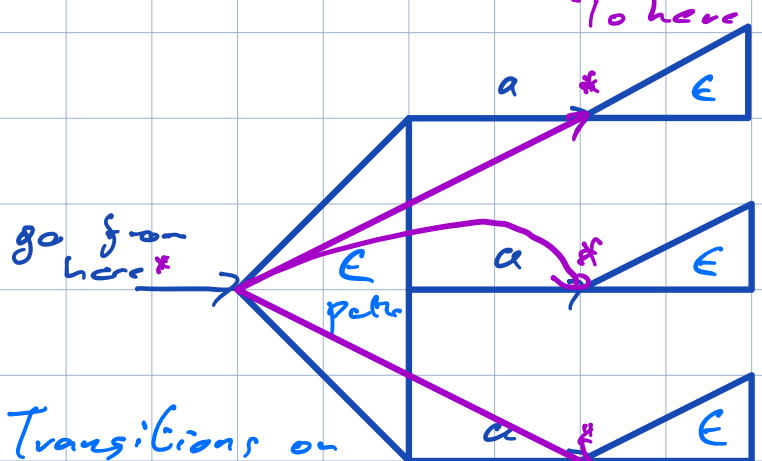
For ϵ-NFA follow the input string w as many ϵ thrown in as possible and you'll find your resulting path.

Language of ϵ-NFA is the same as other FA it's any string w that gets to a final state.

Equivalence of NFA, ϵ-NFA

- Easy for NFA → ϵ-NFA an NFA is simply an ϵ-NFA with no ϵ-transitions.
- Converse needs to construct NFA from ϵ-NFA that accepts the same language.
- Do so by combining ϵ transitions with the next transition on a real input

How? We have to get rid of the ϵ-trans



In blue is the ϵ a path for ϵ-NFA i.e all possible ϵ-transitions followed by $\delta(S, a)$

In purple are equivalent

ϵ

NFA transitions.

Because NFA allows for multiple transitions on one symbol we can say

$$\delta(q_\epsilon, \epsilon a) = \delta(q_N, a)$$

Where q_ϵ = some state in Q of ϵ-NFA, q_N is same state in Q of NFA.

We may have to increase the no. of final states in our NFA as ϵ-T could lead to a final state.

Formal equivalence

ϵ-NFA has Q states, Σ inputs, ss q_0, F final t. func δ_ϵ

We construct an ordinary NFA w/ Q, Σ, q_0, F', δ_N

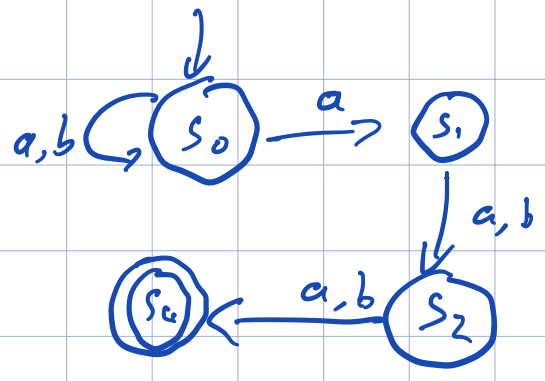
Compute $\delta_N(q, a)$ as follows:

1. Let $S = CL(q)$
2. $\delta_N(q, a)$ = union of all $\delta_\epsilon(p, a)$ for p in S

- F' = set of all states of q such that $CL(q)$ contains a state of F

aab

aaaa



abb

aba

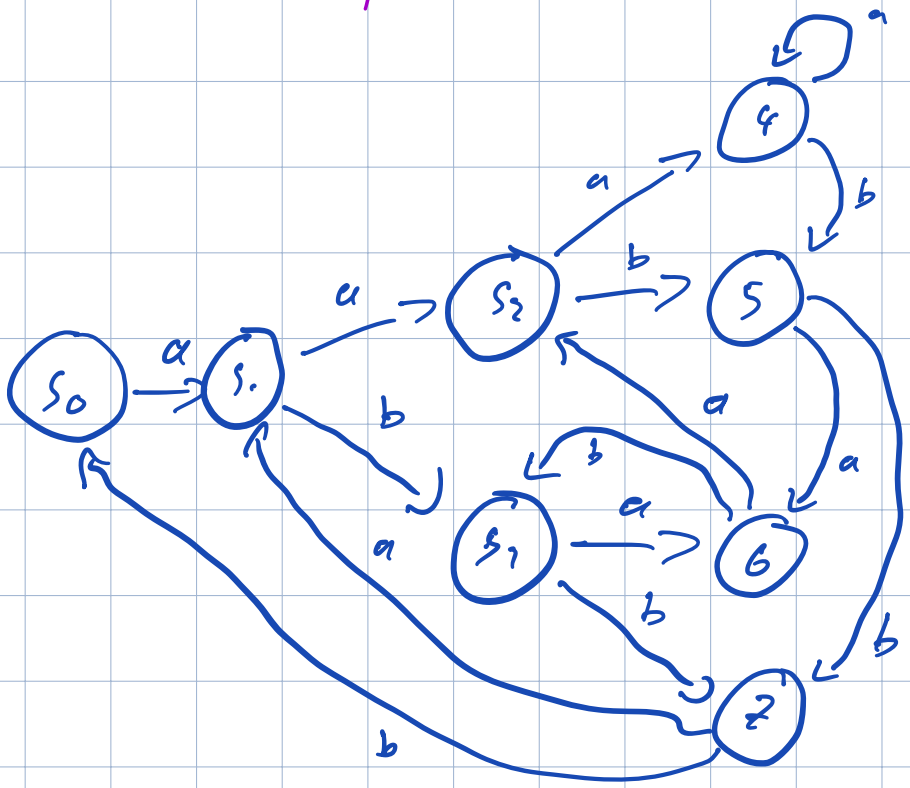
aaa

aab

$2^3 = 8$

Make path to all accepting states

also break path at end



Prove DFA for L_3 has $\geq 2^3$ states

DFA has $Q, F, \Sigma, \delta, q_0$

$Q \geq 2^3$ $Q = 8$

Prove by Q = 8 or by Q != 7

$L_j \cup \{\}$

A DFA D recognises L_n has less than 2^n states

Problem Set 1:

- Types
- Language an Automata accepts
- Sound of No hands clapping
  - Compute sum of 0 ints

Types:

- Types are important in AT
- Distinctions

Strings v Chars

String: 0 or more Chars "a"

Char: Symbol in Alphabet 'a'

- Oddity: in ϵ-NFA some arcs labelled with ϵ, some with chars like 0

Not a problem because we can think of Chars as strings of len 1

Mentally concat arcs into strings.

$0 \xrightarrow{\epsilon} 0 \xrightarrow{0} 0 \to 0 \xrightarrow{\epsilon} 0 \xrightarrow{0} 0 = 010$

Sets v elements

- Strings are elements
- But sets of strings are sets
  - ϵ is a string
  - $\emptyset$ is a set.
- Sets can have members
  elements never do
- Empty sets have no members
- Elements cannot have members
- States are always elements
- Subset construction appears to construct DFA states that are sets of NFA states.
  - But DFA states correspond to sets of NFA states but are elements like q
    - $\{p, q\}$ is the name of a DFA state

Language of Auto

- Auto accepts strings
  - These strings are labels of paths from start state to an accepting state.
- They also accept languages
  - Languages are exactly the set of strings an Aut accepts Σ^*
- Many strings, one language

- Automaton A accepts Language L.
  Means L is the one language of A
  A accepts all strings of L and no more.

No Ands Clapping: Applying an operation to 0 things
- What is the sum of 0 integers?
-

Sum of 0 ints:

$4+7+3 = 14$

$4+7 = 11$

$4 = 4$

→

$0 + 4+7+3 = 14$

$0 + 4+7 = 11$

$0+4 = 4$

$0 = 0$

0 is identity

We use 0 as identity in sum loops i.e:

```
Sum = 0
for(...)
    Sum += 1
```

0 is also the id for OR stmts

Also used in string concatenation

Is 0 odd or even
- Even because $0 \% 2 = 0$
- Empty string has an even no. of every symbol

Regex, equivalence to FA

- Algebraic notation for description of regular languages
  - Can convert any regex to NFA, vv
- Languages by algebra
- If E is an expression L(E) is the language
- Regex uses 3 ops:
  - Union - Union of sets
  - Concatenation - Denoted L/M i.e. L concat M
  - * - Concatenation of strings in language.

Union e.g. $\{01, 111, 10\} \cup \{00, 01\} = \{00, 01, 10, 111\}$

Concatenation: Every string wx such that w is in L, x is in M. Like a dot prod -

E.g $\{01, 111, 00\}\{01, 11\} = \{0101, 0111, 11101, 11111, 0001, 0011\}$

*: If L is a lang then L^* is the set of strings formed by concatenating ≥ 0 strings from L in any order.

e.g. $\{0,1\}^*$: Any binary string inc. ϵ

L^*: $\epsilon \cup L \cup LL \cup LLL \cup \ldots$

Any no. of strings from L.

RE's definition:

Basis 1: If a is any symbol, then a is a RE
and $L(a) = \{a\}$

- a is a language containing one string of len 1

Basis 2: ϵ is a RE and $L(\epsilon) = \{\epsilon\}$

Basis 3: $\emptyset$ is a RE and $L(\emptyset) = \emptyset$

Induction:

$+$: set union.

I1: If E_1 and E_2 are regex then $E_1 + E_2 = RE$
And $L(E_1 + E_2) = L(E_1) \cup L(E_2)$

I2: Concatenation: $E_1 E_2$ is a RE
and $L(E_1 E_2) = L(E_1) L(E_2)$

I3: $* \rightarrow E$ is RE then E^* is RE
and $L(E^*) = (L(E))^*$

Order of Operations:

Parenthesis used to group ops.

1st $*$ = exponent

2nd concat = multi

3rd Union = Add

E.g.

- $L(01) = \{01\}$
- $L(01+0) = \{01, 0\}$
- $L(0(1+0)) = L(01) \cup L(00) = \{01, 00\}$
- $L(0^*) = \{\epsilon, 0, 00, \ldots\}$
- $L((0+10)^*(\epsilon+1))$: all strings of 0's, 1's w/out consecutive 1's

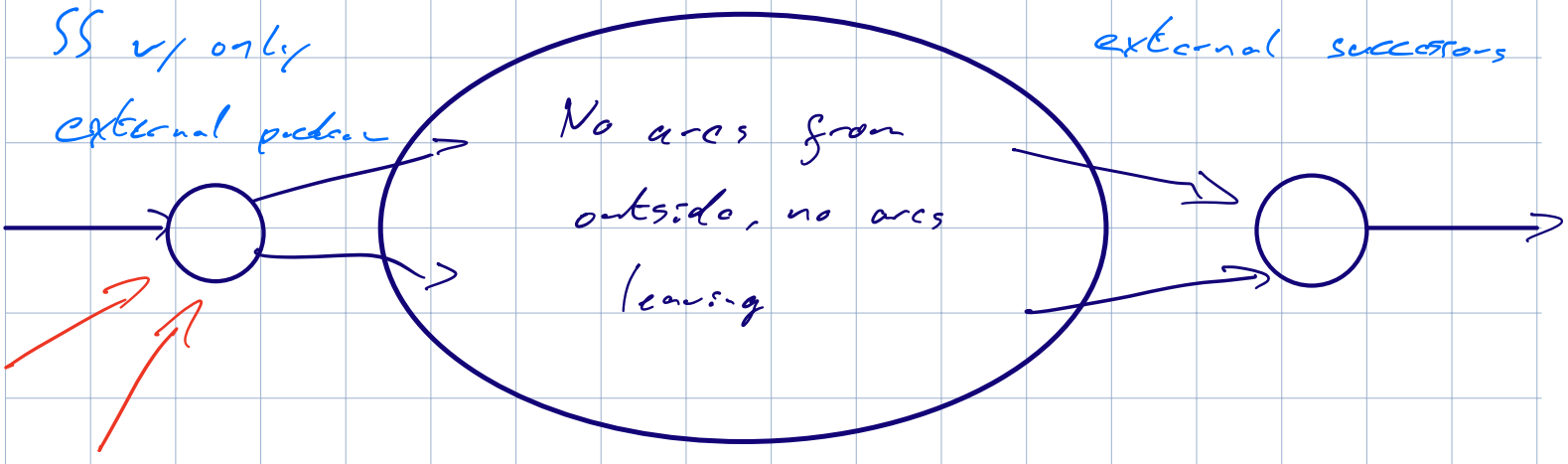
Equiv of RE's, FA's.

- Show for every RE there is a FA accepting the same L
  - Pick ϵ-NFA : most powerful atmtn
- Show for every FA there exists a RE.
  - Pick most restrictive FA : DFA

RE to ϵ-NFA

- Proof is induction on no. of operations. (*, cc, +)
- Must always construct atmtn of a special form.

Special form of ϵ-NFAs



We can add arcs to SS but not inside or to "F"

F here is symbolic.

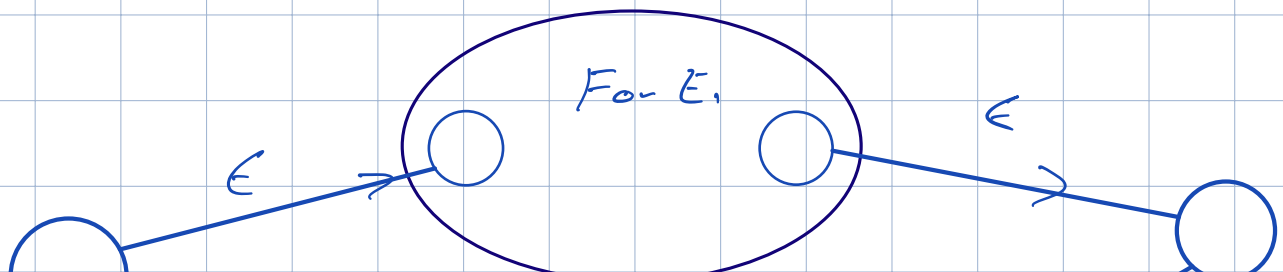
Basis: Symbol a:

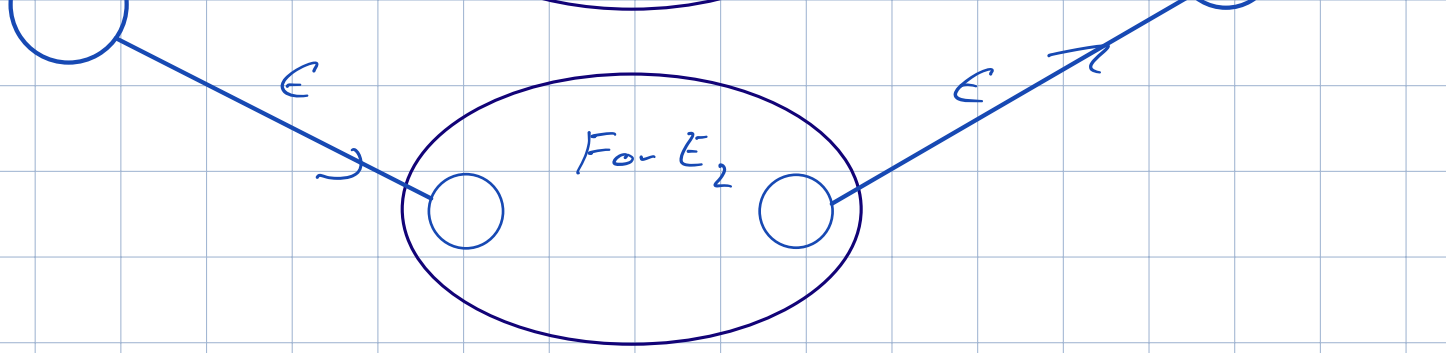
ϵ:

$\emptyset$

Induction for each operation:

Union: $E_1 + E_2$ we assume ϵ-NFAs for E_1, E_2

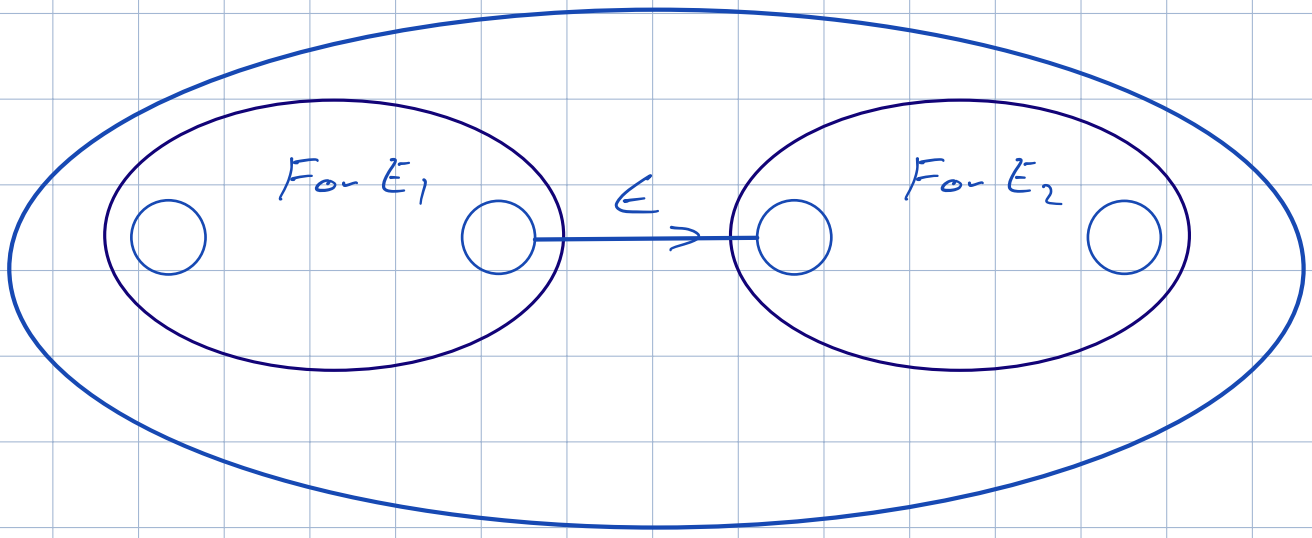




If we have valid ε-NFAs for both REs then we join these FA to new start, and states we can see that ε transitions can occur

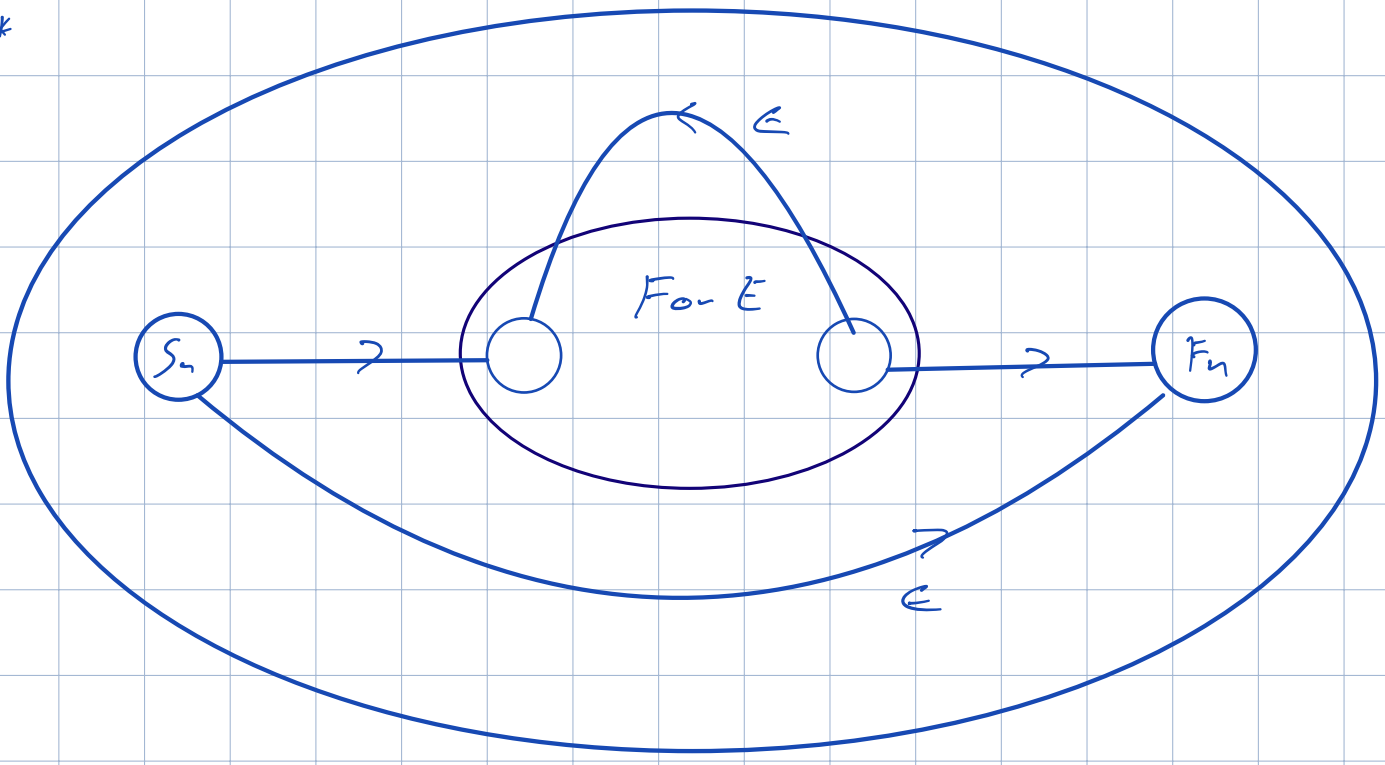
There is a path from s_1 to f_1 through either E_1, E_2

In E_2 add ε transition between E_1, E_2



For E_1 E_2

E^*



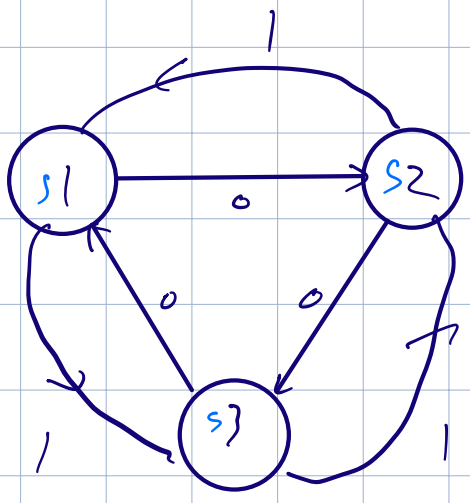
DFA to RE

- Strange induction
- States of DFA are 1, 2, .., n
- Induction is on k, max state no. we can traverse along a path

k-paths:

- k-paths are paths through the graph of a DFA that goes through no-state higher than k.
- Endpoints are not restricted, they can be any state.
- n-paths are unrestricted.
- RE is the union of RE's for n-paths from start to each final state.

K-paths example:



The 0-paths from S2 to S3 is the language of the RE: 0 because we can only follow an arc from S2 -> S3 and there is only one which has the label 0

0 is the only path that gets from S2 to S3 without going through any other state.

The 1-paths for S2 to S3 can go through S1 but not S2 or S3: RE for labels: 0, 11

2-paths for S2 to S3 = 0, 11, 1011, 100, 101011, 101011

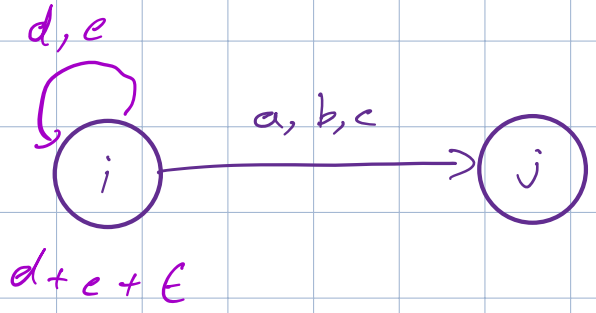
$= (10)^*0 + 1(01)^*1$

3 paths: complex

DFA-to-RE

- Basis: $k = 0$; only arcs or a node by itself

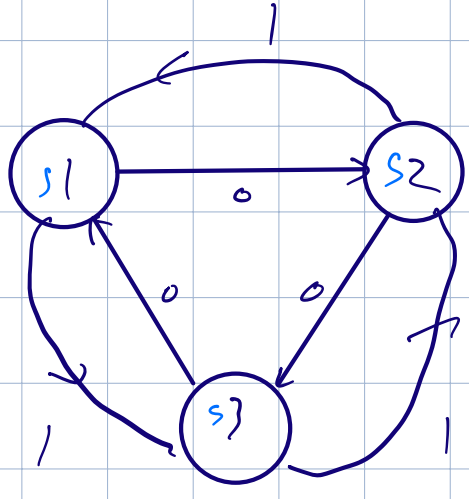
- Induction: Construct RE's for paths allowed to pass through state k from paths allowed up to $k-1$



k-path induction: Let R_{ij}^k be the regular expression for the set of labels of k-paths from state i to state j

basis: $k = 0$. R_{ij}^0 = sum of lbls of arc from i to j.

- $\varnothing$ if no such arc
- but add ϵ if $i = j$



$R_{12}^0 = 0$

$R_{11}^0 = \varnothing$ because no arc, and we add ϵ because $i = j$

$R_{11}^0 = \varnothing + \epsilon = \epsilon$

$R_{22}^0 = \varnothing + \epsilon = \epsilon$

$R_{33}^0 = \epsilon$

Inductive step:

Assume we've written all steps upto $k-1$

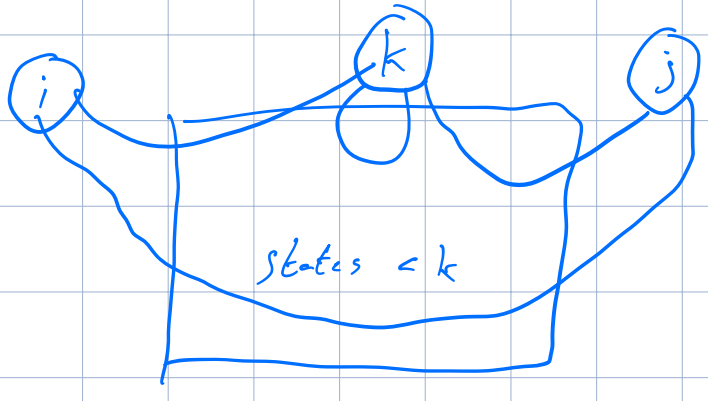
A k-path from i to j either

1. never goes through k at all
2. Goes through k one or more times.

$$R_{ij}^{k} = R_{ij}^{k-1} + R_{ik}^{k-1} \left(R_{kk}^{k-1}\right)^{*} R_{kj}^{k-1}$$

- R_{ij}^{k-1}: doesn't go through k
- R_{ik}^{k-1}: Goes through i to k the 1st time
- $\left(R_{kk}^{k-1}\right)^{*}$: 0 or more times from k to k.
- R_{kj}^{k-1}: Then from k to j

Illustration of Induction:



k path might go through k 0 times, Once or any no. of times.

Final step of induction.

- The RE w/ the same language as the DFA is the sum (union) of R_{ij}^n, where:
  1. n is no. of states; i.e paths are unconstrained.
  2. i is the start state
  3. j is one of the final states

Example:

$$R_{13}^3 = R_{13}^2 + R_{13}^2 (R_{33}^2)^* R_{33}^2 = R_{13}^2 (R_{33}^2)^*$$

- $R_{13}^2 = (10)^* 0 + 1(01)^* 1$
- $R_{11}^2 = \epsilon + 0(01)^*(1+00) + 1(10)^*(0+11)$

$$\therefore R_{13}^3 = [(10)^* 0 + 1(01)^* 1][\epsilon + (0(01)^*(1+00) + 1(10)^*(0+11))]^*$$

Summary

DFA, NFA, ϵ-NFA as well as regex describe exactly the same set of langs: regular languages

Algebraic laws for REs

- Union + concatenation are like addition, multiplication
  - + is commutative, associative.
  - CC is associative
  - CC distributes over +

- Exception: CC is not commutative.

Identities and Annihilators

- $\emptyset$ is id for Addition
- 1 is id $\times$

- Empty set $\emptyset$ is the id for $+$ union
  - $R + \emptyset = R$

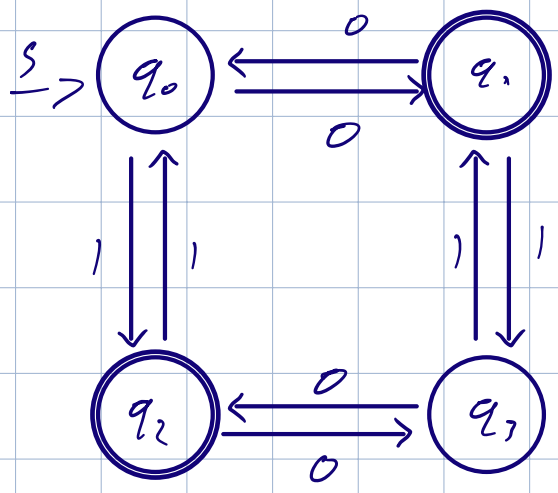
- ϵ is the id for concatenation
  - $\epsilon R = R\epsilon = R$

- $\emptyset$ is the annihilator for concatenation
  - $\emptyset R = R\emptyset = \emptyset$

DFA, Lang 5 things formally

Q: Set of states
Σ: set of input elements
q_0: Start state.
F: Set of final states
δ: Transition function

e.g.



$Q = \{q_0, q_1, q_2, q_3\}$

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

$q_0 = q_0$

$F = \{q_1, q_2\}$

δ = Transition table →

	0	1
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_2	q_3

Definitions:

- Computation of DFA M on input w
  = Sequence of states visited
  i.e. 0110 = q_0, q_1, q_3, q_1
  States can repeat here.
- A computation is accepting if last state in sequence of M on w is in the set of Final states.
- Language is set of w for which M on w is accepting
- A language is regular if it is the language of some DFA

- No. of computations of DFA is always

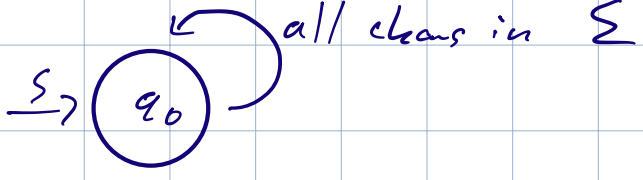
Regular languages examples:

- Σ^*: All strings over an alphabet Σ.

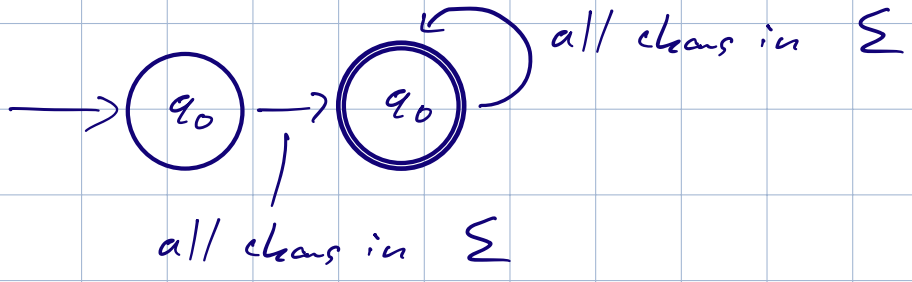
Regular

- $\emptyset$: Empty set

No final state for anything



- $\Sigma^* - \{\epsilon\}$:



Alphabet w/ out ϵ, must not accept in start state but does accept in one transition

languages are sets and accept set operations.

Set operations on (reg.) L's:

- Strings can be concatenated

Let C be a set of languages

Say is closed under operation $\oplus$ if applying $\oplus$ to languages in C results in a language already in C.

Examples of Closure: Integers are closed under addition, multiplication but not division

Regular Operations: let L_1, L_2 be langs

- Union : $L_1 \cup L_2 = \{x : x \in L_1 \text{ or } x \in L_2\}$
- Concat : $L_1 \cdot L_2 = \{xz : x \in L_1, z \in L_2\}$

  e.g $L_1 = \{0, 10\}$

  $L_2 = \{1, 101\}$

  $L_1 \cdot L_2 = \{01, 0101, 101, 10101\}$
- Star : Only defined for one language

$$L_1^* = \bigcup_{k \geq 0} L_1^k \rightarrow L_1^0 = \{\epsilon\}$$

$$L_1^k = L_1 \cdot L_1^{k-1}$$

Non-regular operation:

Complement: let L be a lang.

$\bar{L}$ is the complement of L

$\bar{L} = \Sigma^* - L$

$\bar{L}$ is all the strings over Σ not in L

so $\bar{L} \cap L = \Sigma^*$

We can say Regular languages are closed under complement.

Why is this true?

We have DFA: DFA

Show that for any Regular language if we take the compliment we get a regular language as output.

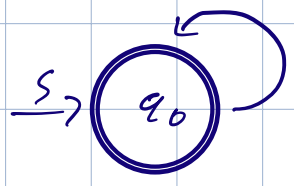
If we have a DFA we must have L_1 where

L_1 is a regular language

DFA M has some alphabet Σ_1 such that $L_1 \subseteq \Sigma_1^*$.

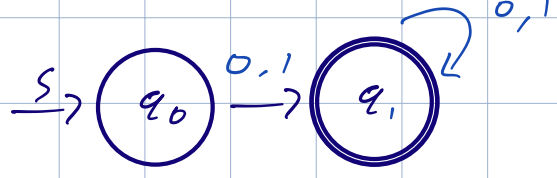
We can say that $\bar{L}_1$ is $\Sigma^* - L_1$, $L_1 \cup \bar{L}_1 = \Sigma^*$

And any Σ^* can be a dfa



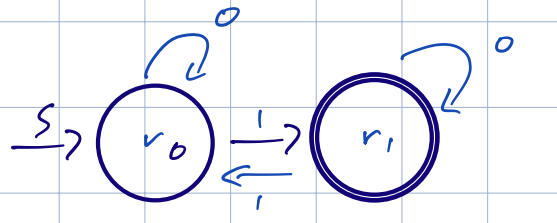
We can also flip the final and non-final states.

Union:

L_1: 

011

L_1: q_0, q_0, q_1, q_0

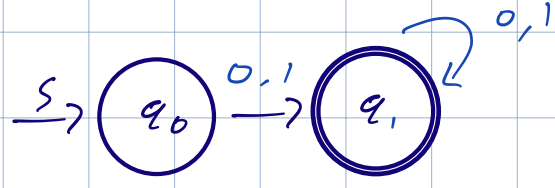
L_2: 

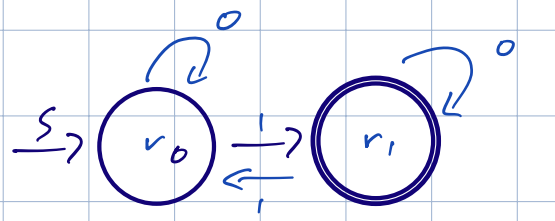
L_2: r_0, r_0, r_1, r_0

the idea of showing closure under union is to try simulate both machines at the same time

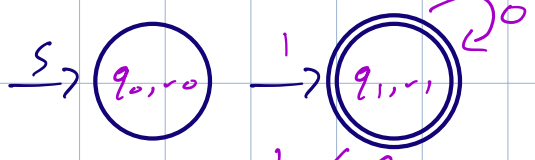
Theorem: reg langs are closed under union

Idea: "Simulate" both machines at same time.

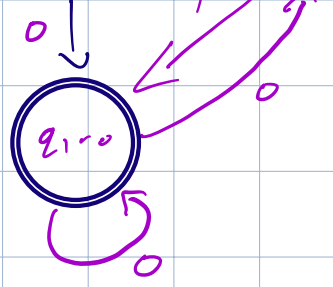
L_1: 

L_2: 

Create a start state for both:



Any state with q_1, r_1 is final



Product construction

- The cartesian product of state sets

Closed under Union means a machine that would accept strings from either L_1, L_2.

Intersection:

De Morgans law: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

because we know Reg langs are closed under complement and union we know that the intersection is also closed.

Concat: Are regular languages closed under concatenation?

$L_1 L_2 = \{ xz : x \in L_1, z \in L_2 \}$

Is this regular

let w : str: $\underbrace{w_1 w_2 \dots w_i}_{\in L_1?} \underbrace{w_{i+1} \dots w_n}_{\in L_2?}$

Where is the split?

have to instantly jump from one machine to the other at any state

look at generalistion of the problem.

We use NFA's.

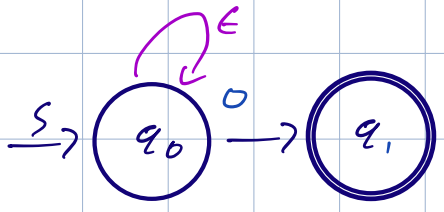
- Any no. of transitions from any state
- ϵ-transition = is taken, no char is read on that

Transition

Nondeterministic = if some way to make choices ends in final state we will accept.

Multiple choice paths but decision to go - final choice is taken. Always 1 choice at a time

no. of computations can be anything

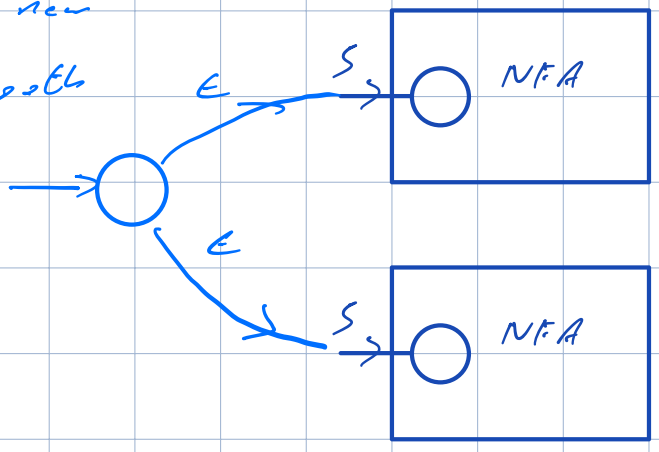


because we self loop on a there are infinite possible states of q_0 before proceeding to q_1.

NFA for regular ops.

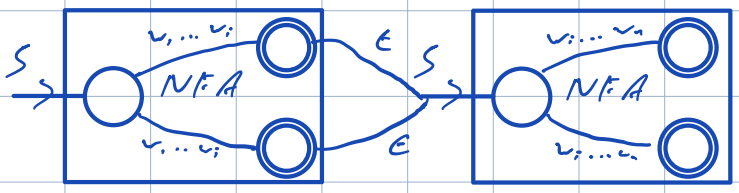
Union: We have 2 NFAs

We make a new q_0 before both w/ ε-t

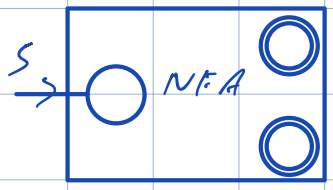


NFA concatenation:

We need to jump on ϵ-transitions only from final states from NFA_1 to NFA_2



NFA Star: With * we want to be able to pick a string from our input and return to the start repeatedly.



Theory of computation:

Closed under intersection.

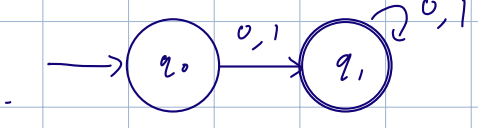
We can show that DFAs are closed under intersection using de morgans laws.

De Morgans: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

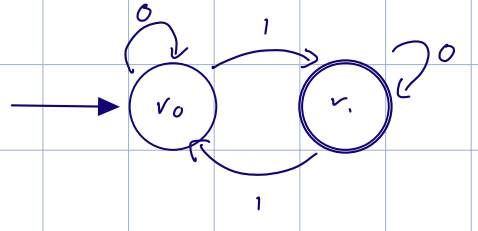
1. Because we know that DFAs are closed under complement and union they must be closed under intersection acc. to Demorgans laws.

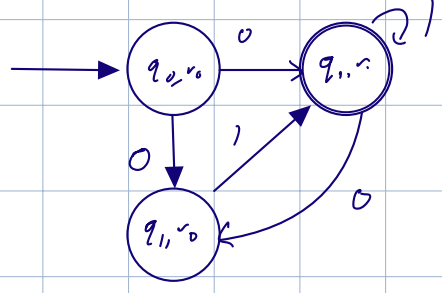
2. We can also use product construction and change the final states as necessary

e.g.



	0	1
r_0	r_0	r_1^*
q_0	q_1^*	q_1^*
r_1	r_1^*	r_0
q_1	q_1^*	q_1^*

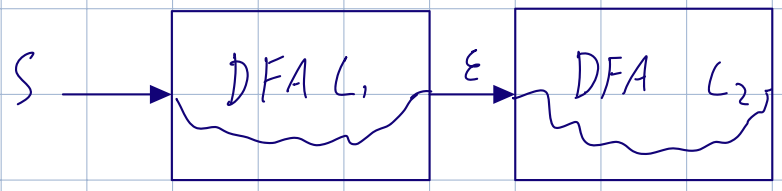


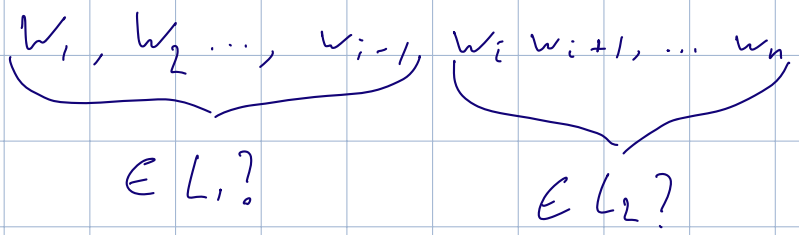


$r \cap q$

Intersection: all the pairs of F states have the property of a final states in intersection.

Concatenation: $L_1 L_2 = \{ xz : x \in L_1, z \in L_2 \}$





We generalise DFAs to allow them to be non deterministic
i.e for an input w, we don't know if

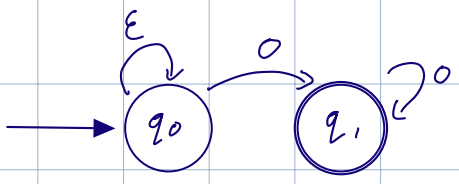
NFA: non deterministic finite automaton

- NFAs have any no. of transitions from any state
- They can have an epsilon transition
  an ε-transition allows for a transition between states w/ out any input
  - is taken, no character is read.
- If some way to make choices to end in a final state we will accept.
- If any possibility of making choices to end in a final state we will make those choices
- Always one choice at a time.

ε transitions are essentially jumps

The no. of computations could be anything

ex



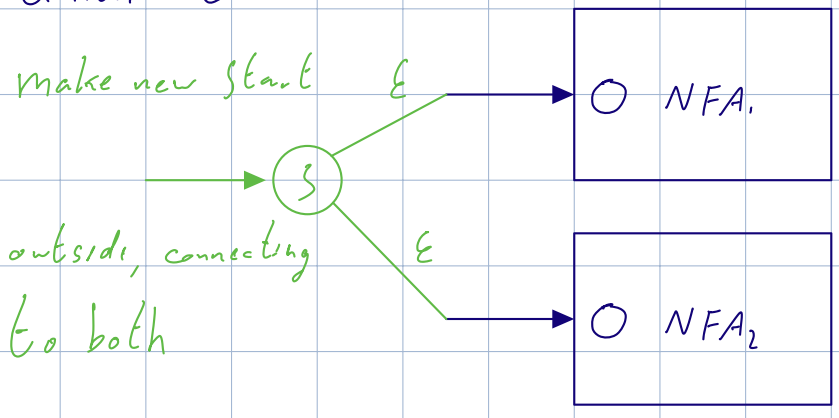
here there are infinitely many ε-transitions we can make before we go to q_1.

These inf. transitions makes it so we can't simply flip the final and non-final states to take the compliment.

It does allow for quick regular operations.

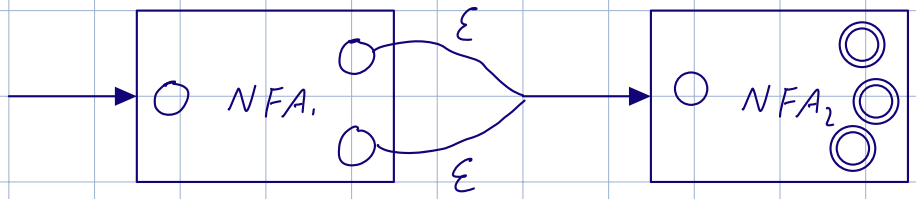
NFA's for reg ops: $\cup$, $\circ$, $*$

Union $\cup$



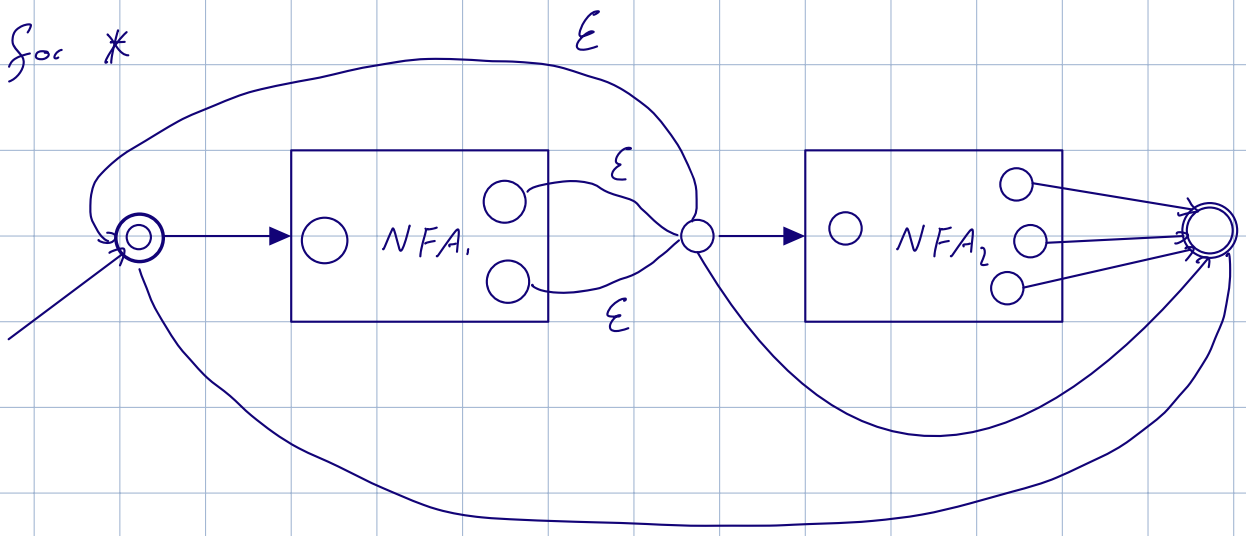
The ε transition will always take the correct path.

for concatenation of NFA's the order matters



ε moves from the formerly final state of NFA_1 to the s of NFA_2

for *



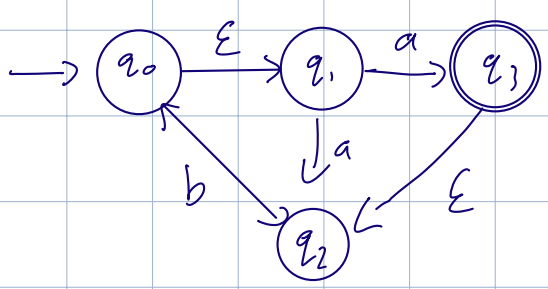
NFA will always make the right choice

The languages of nfas are closed under union, concatenation and * but it does not show that DFAs are

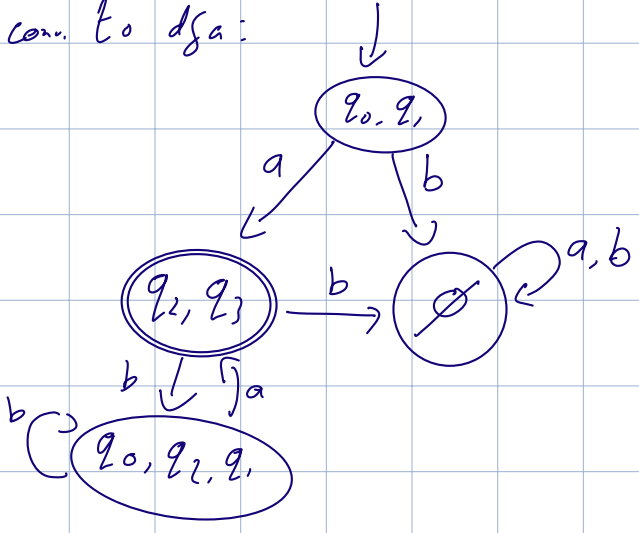
We have to show that the relationship between nfa's and dfa's

Relationship between nfa's and dfa's

- every dfa is an nfa.
- not every nfa is a dfa



conv. to dfa:



Power set construct →

There no valid transition for S on b.

We can say nfas are equivalent to dfa's
dfa's have 2^n possible states of an nfa
but not all are always accessible

equivalence of nfas and dfas mean they can comp the same things.
note: getting to a final state w/ no transitions part way through a string is invalid.

---

dfas / nfas accept a language.
We provide a string to them:
It computes by going between states, says acc. or not.
Does not give reason why string is acc. or not
If we say that
$L = \{ w \in \{0,1\}^* : \text{no. of 0's in } w = \text{no. of 1's in } w \}$
we want to be able to describe a language precisely:
This is the job of regular expressions.
Regex has 6 forms: Let Σ be an alphabet.
R is one of the following
1. $R = \emptyset$
4. $R = R_1 \cup R_2$

2. $R = \varepsilon$

3. $R = a \ (a \in \Sigma)$

5. $R = R_1 \cdot R_2$

6. $R = (R_1)^*$

ex: $(0 \cup 1)^* 101$

Any combo of 0's or 1's, ending in 101

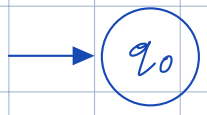
$(01)^*$ all strings of the form 0101... 01 or ε

example regex:

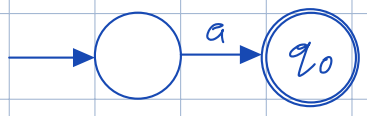
Theorem: nfas + regex are equivalent

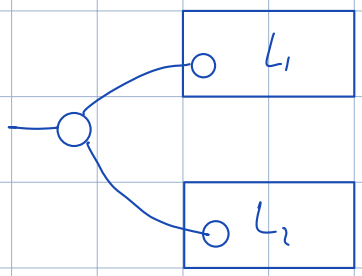
Lemma: (little result) every regex can be converted to an nfa

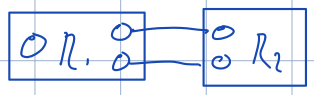
Regex to nfa:

1. $R = \varnothing$ 

2. $R = \varepsilon$ 

3. $R = a \{a \in \Sigma\}$ 

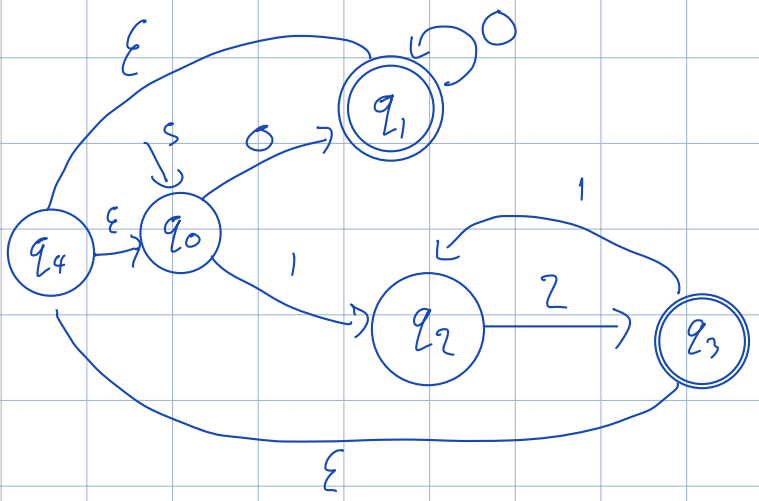
4. $R = L_1 \cup L_2$ 

5. $R = R_1 R_2$ ⟶ 

6. $R = (R_1)^*$

ex: $(0 \cup 12)^*$

guess



every regex can be converted into an nfa

Lemma 2: every NFA can be converted to an equivalent regex

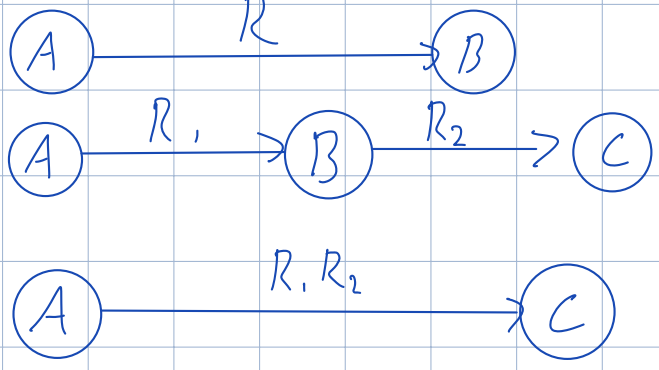
every nfa can, without loss of generality, have 1 final state

We can only add a start state w/ out any incoming transitions

We now have a single incoming and final state.

If we can generalise the inbetween parts of the nfa we can prove nfa equivalence for all regex

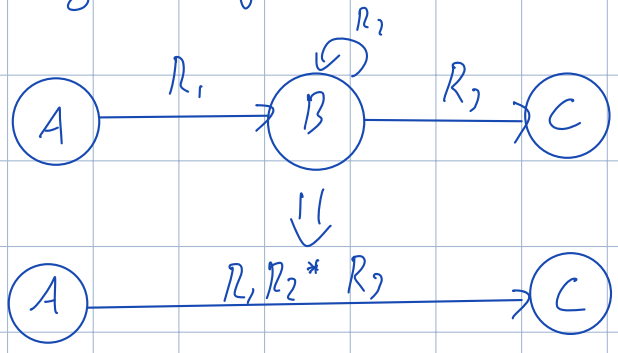
lets think about A goes to B on some regex R



what if we have A->B->C and want to remove B

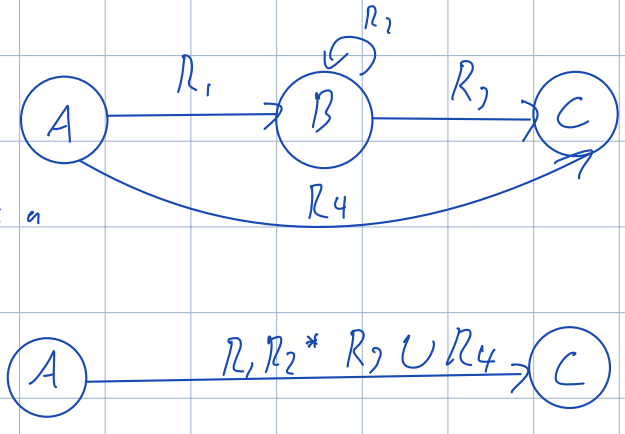
We can change it into a single transition.

What abt following scenario:



---

generalised nfa, every transition is a regex



We can show that any operation an nfa performs can be described in regex syntax

generalised nfas have nothing going into the start state and nothing leaving the final state

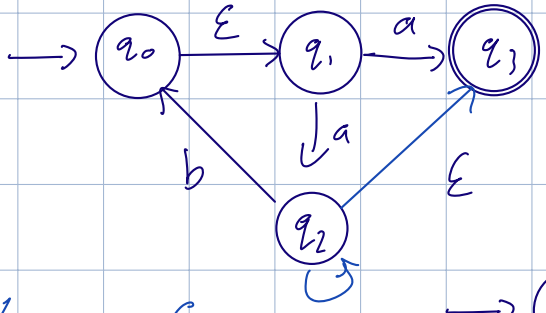
We can repeatedly rip/delete a state from our gen-nfa

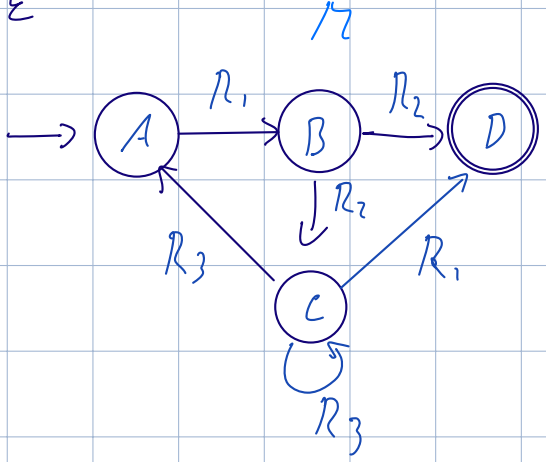
1 at a time until we have only the 2 states left w/ a regex syntax

At the end, the generalised nfa will have the form

R will be an equivalent regex of our nfa

example: convert this nfa into g-nfa form



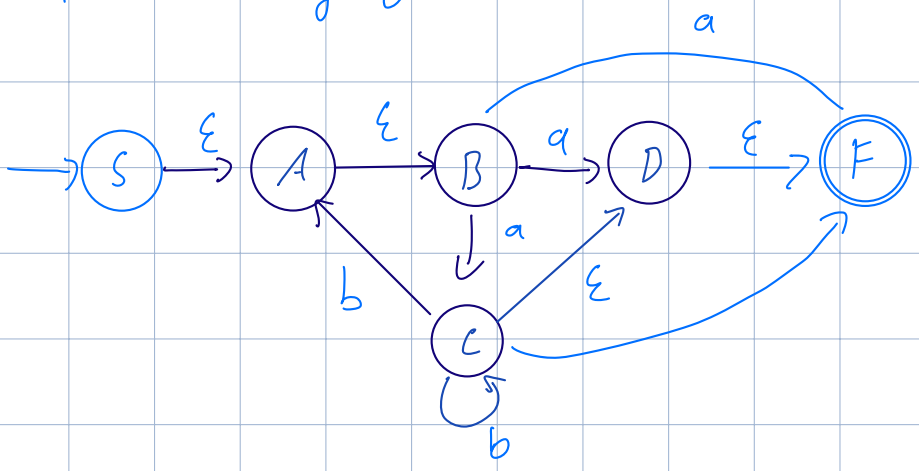


let $R_1 = \varepsilon$

$R_2 = a$

$R_3 = b$

Change M so that there is a new start state w/ no incoming δ

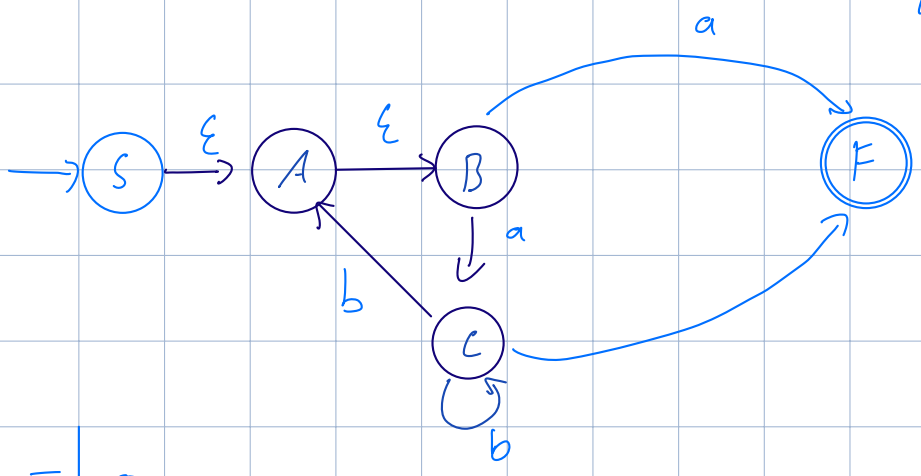


make an i.e. List

D: In	out
B, C	$\{\}$

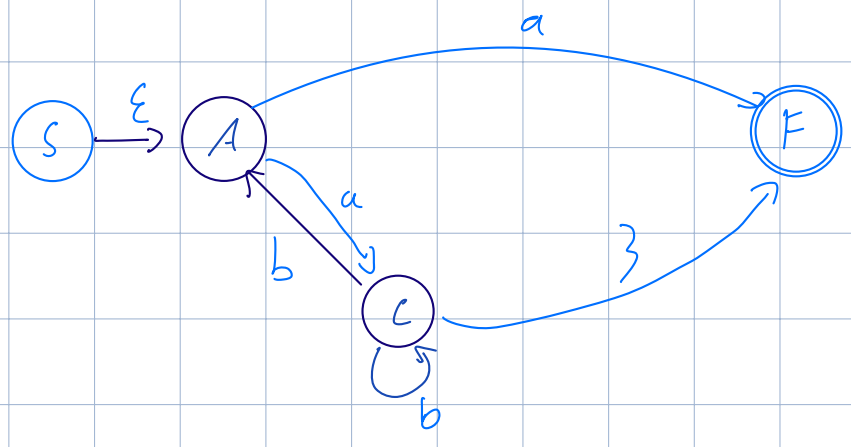
make a transition for: B to $\{\}$: R_2 (a)
C to $\{\}$: R_1 (ε)

we can now delete q_3 (B)
because we've replaced its transitions



B: I	O
A	C F

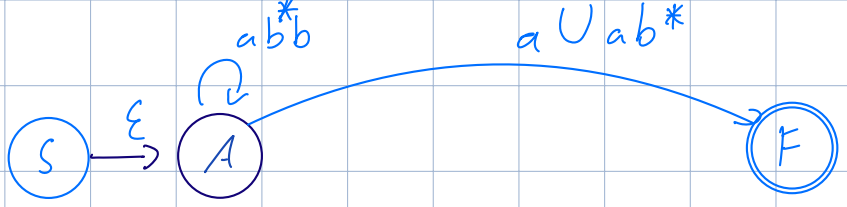
make transition for A -> C : a
leave out ε because A -> F : a
we not reading anything



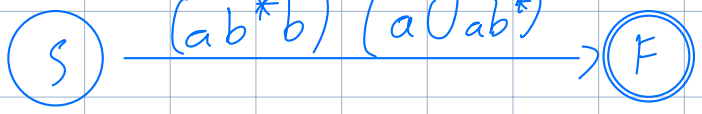
now rip C :

I	O
A, C	C, A, F

Trans for A -> A : ab
A -> F : ab^*
C -> C : b^*



rip A

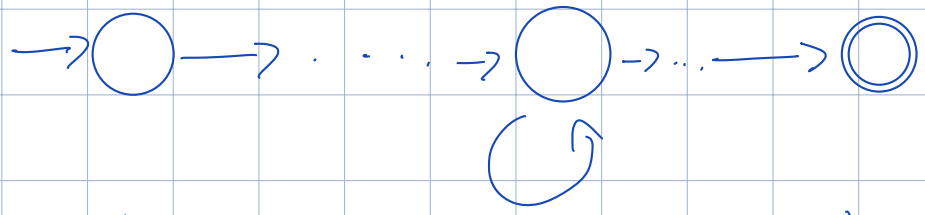


We can have many d.ff. regex as output but as long as we rip correctly it will be valid

Some languages are not regular:

suppose w is a string acc. by dfa M

What other strings are accepted?



If a loop exists (leading to a final F)

there are infinitely many other strings accepted

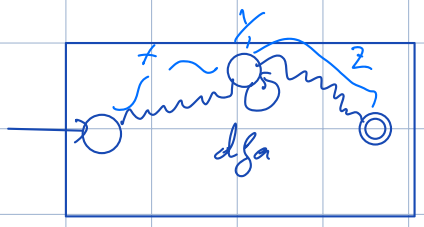
When can we guarantee a loop?

We can guarantee a loop if $len(w) \geq |Q|$ exists and is accepted $\exists w \in \Sigma^* (len(w) \geq |Q|)$ loop

? because we always start in the start state ∴ we always traverse $len(w)+1$

for any $w \in L$ with $|w| \geq |Q|$ (# of states in dfa)

We have a loop



$w = xyz$

$xz \in L$

$xy^2z \in L$

$xy^iz \in L$

note:

1. $xy^iz \in L$ for all $i \geq 0$
2. $|y| \geq 1$, $(y \neq \varepsilon)$
3. $|xy| \leq$ # of states in DFA.

We can prove the pumping lemma for regular languages

Pumping Lemma: A proof for irregularity of a language

If a language L is regular then any string in the language will have a certain property provided that it is long enough

Pre-Lemma note: We know that a language is regular if we can construct a dfa for it.

There are 2 pumping lemmas defined for
1. Regular languages
2. Context free Languages

For regular Languages:

For any regular Language L, there exist an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists $u, v, w \in \Sigma^*$, such that $x = uvw$ and:

a. $|uv| \leq n$

b. $|v| \geq 1$

c. For all $i \geq 0$: $uv^i w \in L$.

A string v is 'pumped' if v is inserted any no. of times, the resultant v still remains in L.

Pumping Lemma is used as a proof for irregularity of a Language.

If a language is regular it always satisfies PL

If there exist one string made from pumping not in L then L is not regular.

Pumping Lemma:

Let L be a regular Language

There is a dfa for L that has a certain no. of states in it

Then there exists a "pumping constant" or length p for L

p: no. of states in dfa

To get this looping behaviour to work we need to have strings acc. by the dfa / in the language and are long enough $|w| \geq p$

For all $w \in |L|$ with $|w| \geq p$

there exists x, y, z such that $w = xyz$

So that:

1. $xy^iz \in L$ for all $i \geq 0$ can loop as often as we want.
   $xy^iz \in L \ \forall i \{i \geq 0\}$
2. $|y| \geq 1$
3. $|xy| \leq p$

If a language is regular then it must have these properties: if one or more properties are not met then it's not regular

The pumping lemma is a proof of irregularity

If we find a string $\geq |p|$ that breaks the property then we prove irregularity

Prove $\{0^n 1^n : n \geq 0\}$ is not regular:

To apply P.L. assume Lang is reg. ∴

For any $w \in L$ $|w| \geq |p|$:

There exist xyz s.t

1. $xy^i z \in L$ for all $i \geq 0$
2. $|y| \geq 1$
3. $|xy| \leq p$

let $n=2$, $w = \overbrace{00}^{x}\overbrace{11}^{y}$, $n=3$ $w = 000111$

So. $n=2$, $x = 0$, $y = 1$, $z = 1$

$n=3$, $x = 000$, $y = 11$, $z = 1$

x of $n=3$! = x of $n=2$

Choose $w = 0^p 1^p$

look at all possible breakups of w into xyz

$x = 0^a$

$y = 0^b$ $b \geq 1$

$a = p - b = n - b$

z = rest

lets pick y^i $i=2$

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

The only way $0^{p+b}, 1^p$ is in L is if $p+b = p$

$\therefore$ b would be 0, which breaks 2. $|y| \geq 1$

This proves that it can't be regular.

ex: Language of perfect squares

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

Assume L is regular. for some $w \in L$ $|w| \geq p$

xyz s.t

1. $xy^iz \in L$ for all $i \geq 0$
2. $|y| \geq 1$
3. $|xy| \leq p$

Sub p in for n

$w = 0^{p^2}$ $|w| = p^2$

$x = 0^a$

$y = 0^b$ $b \geq 1$

$z = 0^c$

pick $i = 2$

$w = 0^a 0^b 0^b 0^z = 0^{p^2}$

$|x| = a$, $|y| = b$

$|p^2| = |xyz|$

$|xyyz| = |p^2| + |y| = |p^2 + b|$

Pump up: choose val $i \geq 1$

choose $i = 2$: $|xyyz| = |p^2 + b|$

Only way $p^2 + b \in L$ is if $p^2 + b$ is a perfect square

We know $b \geq 1 \Rightarrow p^2 < p^2 + b$

next perfect $\square = (p^2+1)^2 = p^2 + 2p + 1$

$p^2 < p^2 + b \leq p^2 + p \leq p^2 + 2p + 1$

$|xy| \leq p \Rightarrow |y| \leq p$

Context free Languages:

Overview:

Machine models: Computes whether to acc a string

Regex: formal description of a language

Grammars: How to create strings in a language

↳ Context free grammars

All grammars have:

1. Variables (non-terminals)
2. Terminals
3. Rules - what rules allowed tells us how strings are made
4. Start variables

The purpose of a grammar is to repeatedly apply rules until we get a string.

We say that:

- Languages are over alphabets and that's what terminals are. They are the strings that we want to make at the end of the day

Rules are analogous to transitions
We start @ the SV,

What a grammar looks like:

$\alpha \rightarrow \beta$ both 'α' and β are some mix of vars and terminals

example:

011 → 23 means we can replace any 011 with 23 in our computation

What is a CFG, context-free?
All rules are of the form
$A \rightarrow X$
A is a single variable,
X is any mix of symbols, variables

Consequence is that whenever a term. is created at any point it can never be replaced.
A terminal has no equivalents

ex:

multiple rules w/ same lhs variable

var

$S \to 0S1 \mid \varepsilon$

start var

Terminals

We can start with S and replace it with one of the 2 different rules either '0S1' or 'ε'

choose us

$S \Rightarrow 0S1$ - now we have a new string made up of 2 terminals and S

$0S1 \Rightarrow 00S11$ faced w/ the same choice to

$00S11 \Rightarrow 000S111$ cont. the string or finish

$000S111 \Rightarrow 000111$ here

Lets think abt the lang. of this grammar

easy to see that

$L = \{0^n 1^n : n \geq 0\}$ not regular

All regular languages have a cfg but not all cfg's are regular

Definitions: $S \Rightarrow X$

$\Rightarrow$: Applying the rules means "yields"

Derivation: following rules to a point of output, apply rules w/ start var to get only terminals

ex: let's say we have a derivation:

$S \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \ldots A \ldots B \ldots C \ldots$

$\Rightarrow \ldots X \ldots B \ldots C \ldots$

If A is the earliest acc. var and all other components before it are terminals we say that replacing A is a left most derivation.

Every derivation has a left most one

Dealing w/ CFGs

When applying a rule we're replacing one variable w/ stuff.

If we have a derivation we were able to apply a rule @ each point.

If we want a leftmost version we can simply focus on the leftmost var each time

$A \ldots B \ldots C$ each of the vars are indy

ex: balanced parenthesis

( ))((( ))( )

✓ X ✓

Order matters so does count

BAL = { all balanced parenthesis }

To create a grammar, keep in mind recursion
start w/ base cases 1st do inductive later

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

Let BSL = { all bal. p, square }

( ), [ ], ([ ])[ ]

$S \Rightarrow \varepsilon \mid (S) \mid [S] \mid SS$

Cont. free lang : any lang of a cfg
⇒ every regular lang is a cfl

Theorem cfls are closed under regular operations.
Union, concat, star.
If there a cfg for 2 diff languages is there a cfg for the union

L_1, L_2 are cfLs

$L_0 \{ S \Rightarrow L_1 \mid L_2 \}$

L_1 has grammar g_1, L_2 has grammar g_2

g_3 $g_1 \cup g_2$

$g_3 : S \rightarrow S_1 \mid S_2$ S is new SV

S_1, S_2 are SVs of g_1, g_2

assume vars are not overlapping

we can simply rename if needed.

Concatenation: $S \rightarrow S_1 S_2$

Kleene: L_1^* $S \rightarrow \varepsilon \mid S_1 S$

cfls are closed under reg ops

we can chain vars pointlessly

$A \rightarrow B$

$B \rightarrow C$

$C \rightarrow D$

$D \rightarrow \varepsilon$

} Useless chain

We can mitigate excesses of cfls by using CNF. Chomsky normal form.

Restriction on cfgs

Only 3 possible rules allowed

1. $S \to \varepsilon$ no other var can make the ε
2. $A \to a$ A var is a single terminal.
3. $A \to BC$ BC = vars $B, C \neq S$

S can be ε

A can be a char or 2 vars BC

$\{ w \in \{0,1\}^* : |w| \text{ is odd, same symbol at } i : 0, n, n/2 \}$

a|m|a|m|a 1 ✓ 11 ✗ 01 ✗ 101 ✗ 111 ✓

$S \Rightarrow 1 \mid 0 \mid 1A1 \mid 0B0 \mid \varepsilon$

$A = 1 \mid CAC$

$B = 0 \mid CBC$

$C = 1 \mid 0$

Every CFG can be conv. to CNF

good because in CNF all our vars are productive

In cnf we can deli. the exact no. of rules to apply to get a string.

The C/k algo tells us which rules to apply

Any cfg can be a cnf

eg:

$S \to \varepsilon \mid S.S$

$S_1 \to 1 \mid 0 \mid 1A1 \mid 0B0 \mid \varepsilon \mid A \mid B$

$1A \quad A \to \Delta 1 \mid \Delta 1$

$1 \downarrow 1 ? 1$

cnf:

$S \to \varepsilon \mid S_1 S$

$S_1 \to XA \mid YB \mid 1 \mid 0 \mid CC$

$X \to 1$

$Y \to 0$

$Z \to 0 \mid 1$

$\Delta X \Delta \to T$

$A \to XX \mid TX$

$B \to DY$

$C \to$

$T \to X \mid$

5 stage conversion:

1. Move S out of Rhs

Make new S_0

$S_0 \to S$

$S \to \varepsilon \mid S, S$

$S_1 \to 1 \mid 0 \mid 1A1 \mid 0B0 \mid \varepsilon \mid A \mid B$

2. elim ε-rules : list nullable vars : S_1, S can make ε

S_0 can make ε indirect.

Move rules around

$S_0 \to S \mid \varepsilon$

$S \to \cancel{\varepsilon} \mid S, S \mid S_1 \mid S$

$S_1 \to 1 \mid 0 \mid 1A1 \mid 0B0 \mid \cancel{\varepsilon} \mid A \mid B$

Step 3: elim unit rules "$A \to B$"

$S_0 \to S \mid \varepsilon$

$S \to S, S \mid \cancel{S_1}$

$S_1 \to 1 \mid 0 \mid 1A1 \mid 0B0 \mid \cancel{A} \mid \cancel{B} \mid 0A0 \mid 1B1 \mid 0A1 \mid 1A0 \mid 0B1 \mid 1B0$

$A \to CAC \mid 1$

$B \to BAB \mid 0$

$C \to 0 \mid 1$

$S_0 \to \varepsilon \mid S_1 S \mid 1 \mid 0 \mid 1A1 \mid 0B0 \mid 0A0 \mid 1B1 \mid 0A1 \mid 1A0 \mid 0B1 \mid 1B0$

$A \to CAC \mid 1$

$B \to BAB \mid 0$

$C \to 0 \mid 1$

Step ensure rhs is only vars or 1 terminal

✓ ✓ ✓ ✓ × × × × × × × ×

$S_0 \to \varepsilon \mid S_1 S \mid 1 \mid 0 \mid 1A1 \mid 0B0 \mid 0A0 \mid 1B1 \mid 0A1 \mid 1A0 \mid 0B1 \mid 1B0$

$A \to CAC \mid 1$

$B \to BAB \mid 0$

$U_0 \to 0$

$U_1 \to 1$

$S_0 \to \varepsilon \mid S_1 S \mid 1 \mid 0 \mid U_1 A U_1 \mid U_0 A U_0 \mid U_1 A U_0 \mid U_0 A U_1$
$\mid U_1 B U_1 \mid U_0 B U_0 \mid U_1 B U_0 \mid U_0 B U_1$

$A \to U_1 A U_1 \mid U_0 A U_0 \mid U_1 A U_0 \mid U_0 A U_1$

$B \to U_1 B U_1 \mid U_0 B U_0 \mid U_1 B U_0 \mid U_0 B U_1$

$U_1 \to 1$

$U_0 \to 0$

5. Break up long Rhs

$AU_1 \to A_1$, $AU_0 \to A_0$, $BU_1 \to B_1$, $BU_0 \to B_0$

$S_0 \to \varepsilon \mid S_1 S \mid 1 \mid 0 \mid U_1 A_1 \mid U_0 A_0 \mid U_1 A_0 \mid U_0 A_0 \mid U_1 A_1 \mid U_0 B_0 \mid U_1 B_0 \mid U_0 B_0$

$A \to U_1 A_1 \mid U_0 A_0 \mid U_1 A_0 \mid U_0 A_1$

$B \to U_1 B_1 \mid U_0 B_0 \mid U_1 B_0 \mid U_0 B_1$

$U_1 \to 1$

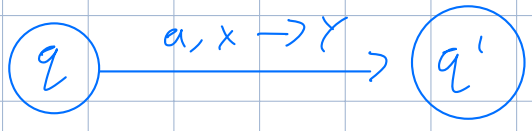
$U_0 \to 0$

Machine model for cgls: Machine model for context free gram

Pushdown automata:

↳ same as nfa, allowing for a stack and operations on a stack.

Push onto stack or pop off stack



a is char read, X is item popped

Y is item pushed

note: a, X or Y may be empty

* Popping empty stack = end of computation. (hangs)

Pop has to happen 1st

a could be an ε-transition

x could be nothing

y could be nothing

A pda transition is possible if we can read a and pop x

accept if: 1. In final state

2. Read all input

3.

Stack doesn't have to be empty, but does start empty

definition:

Q set of states

Σ input alphabet

Γ : set of pushdown symbols

q_0 : initial state

Z : init pd symbol

F : set of final states

δ : Transition function

ex: $\{0^n 1^n : n \geq 0\}$ not regular, have to use stack

If 0 put onto stack, match 1 on to 0

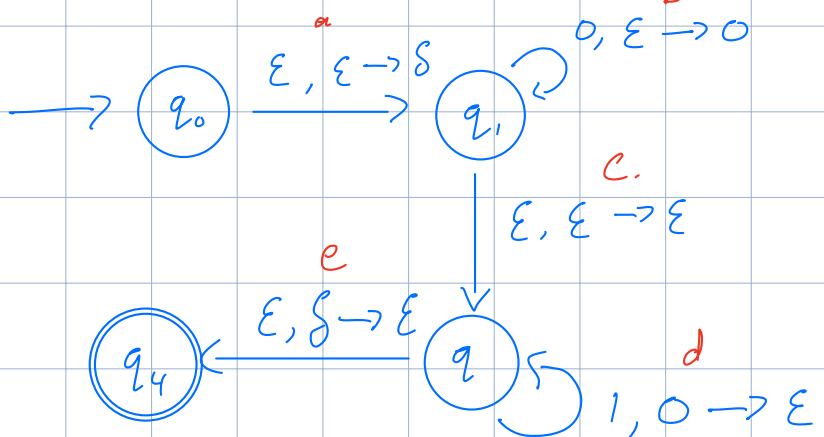
a. read ε, pop ε, push token symbol

b. read 0, pop ε, push 0

c. Triple epsilon t, so we can acc. empty string

d. Read 1, pop corresponding 0, leave empty

e Reached btm of stack : balance of 0's, 1's means only δ is left and we move to q_4, if at q_4 there is still input then it wont be accepted



Theorem : every CFG has an equivalent PDA.

Idea: Simulate derivation on a stack

we have :

$x_1, x_2, \dots x_n$

x_i can be var or terminal

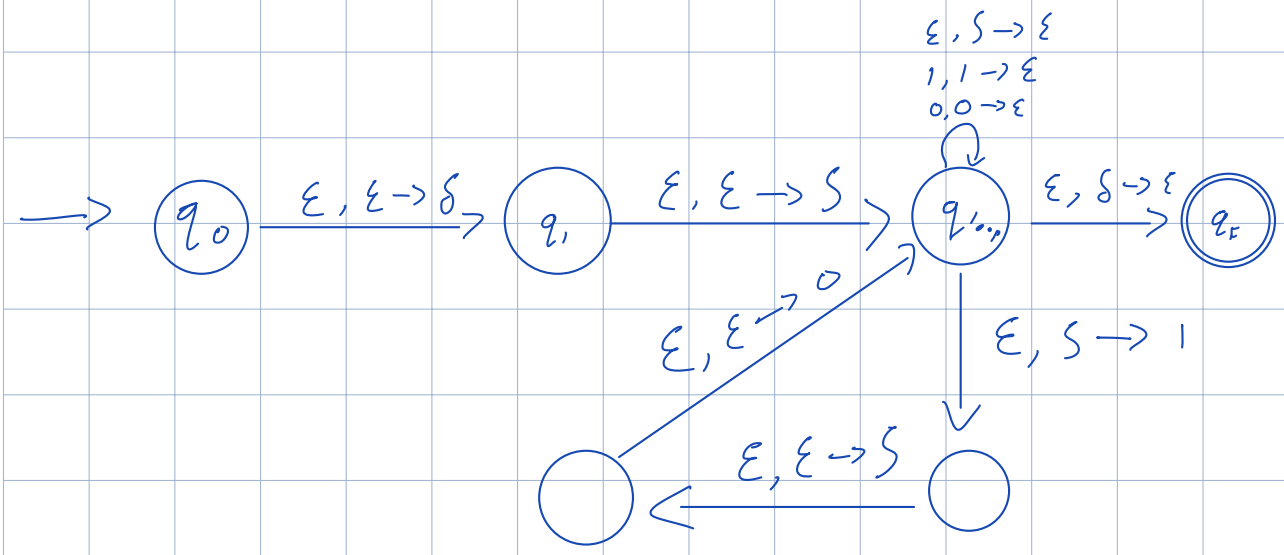
x_1
:
x_i
:
x_n

We can think of the stack as a leftmost set of rules and each x_i can be replaced with a grammar rule

replace X_i w/ the content of a rule or a terminal

ex: $S \to 0S1 \mid \varepsilon$ make a PDA

create a base state



If we see an $ at the top of the stack either apply $\varepsilon, S \to \varepsilon$ or move through the rule of $0S1$, then popping terminals as needed

CFGs can be converted to a CFG using loops

Pda to cfg:

Step 1: ensure Pda has one final transition.

2: ensure every transition pushes or pops but not both.

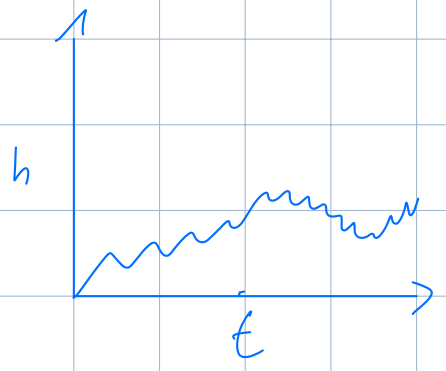
3: all stacks to be empty

Then: Proof based on analysis of stack height "over time"

every transition changes the stack height in some way

either inc. or dec.

Changes by one on every transition.



Stack changes by one each time.

There will be matching transitions:

Cfg: has $A_{p,q}$ $p, q \in Q$ (states of Pda)

$A_{p,q} \equiv$ generates all strings going from $p \to q$ with the same stack height

Base Case: $p = q$ $\therefore A_{p,p}$

Think abt the strings we can make for $A_{p,p}$

$A_{p,p} \to \varepsilon$,

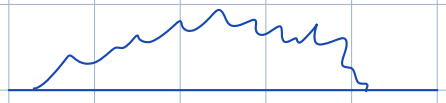
ex: $S \to SS \mid (S) \mid \varepsilon$

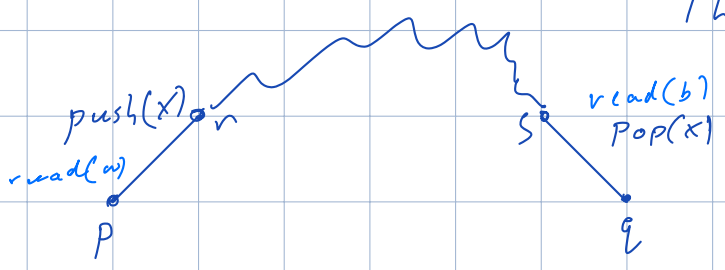
Inductive case 1:

$A_{p,q} \to A_{p,r} A_{r,q}$

If we can go from $p \to r$, $r \to q$
then we can go from $p \to q$

Inductive case 2:





They are inversed transitions
The first transition starts by pushing a var/term onto stack
the final finishes by popping it

let $p \to q$ read a, $s \to q$ read b

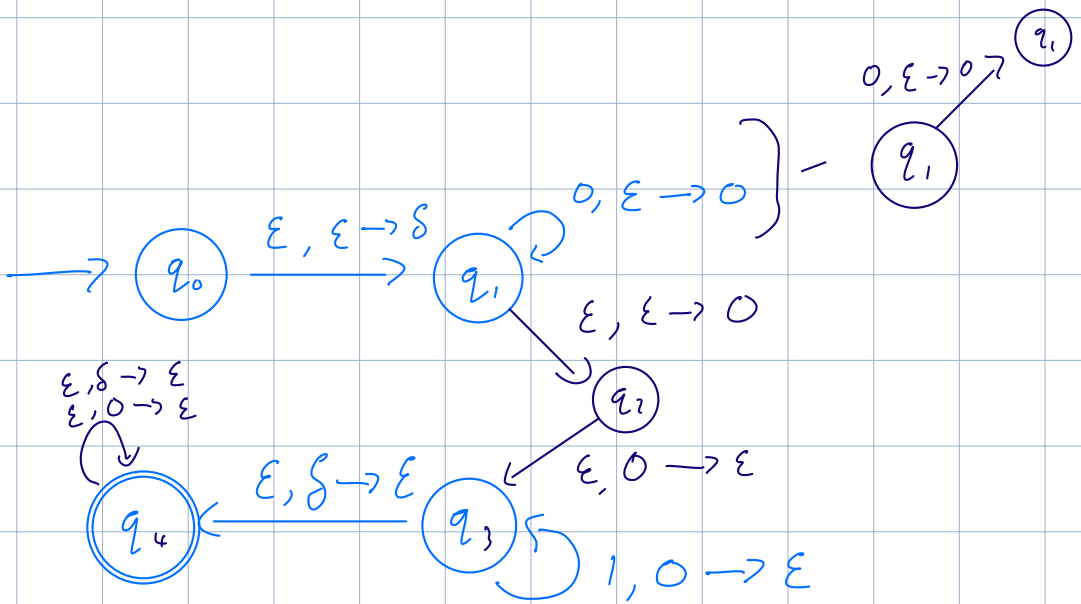
to go from $p \to q$ we read ab
a read in the pda is a production in the cfg

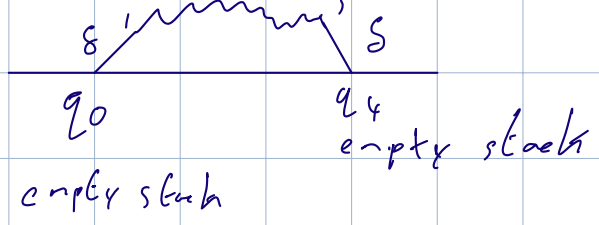
$A_{p,q} \to a A_{r,s} b$

$A_{r,s} \to x_i A_{q_i, q_j} x_j$

repeat this induction until mountain is traversed

ex





$A_{0,0} \to \varepsilon$

$A_{1,1} \to \varepsilon$

$\vdots$

$A_{4,4} \to \varepsilon$

$A_{2,3} \to A_{2,0}\, A_{0,3}$

$A_{q_1,q_1} \to \varepsilon$

$A_{q_0 q_4} \to \delta\, A_{q_1 q_2}\, \delta \mid \varepsilon$

$A_{q_1 q_3} \to$

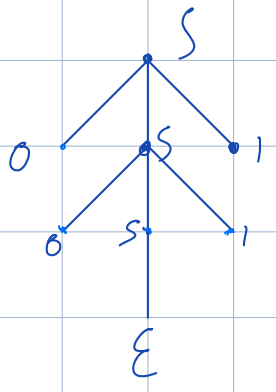
Pumping lemma for cfl's

Q: given $w \in L$ what of the $w' \in L$

let g be a cfg in CNF for L
A pase tree = left most derivation

$S \to 0S1 \mid \varepsilon$

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011$



We then read the leaves from left to right.

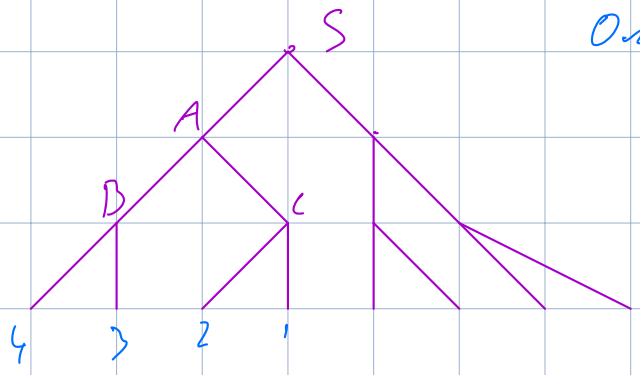
Parse trees, left most derivations are equivalent to each other.

Parse trees allow for the following argument

If a grammar is in cnf, what do the parse trees look like?

The pause tree is a binary tree

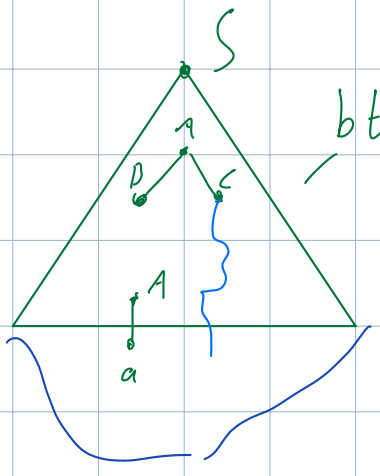
ex:



One Var A has 2 children

BC or a terminal

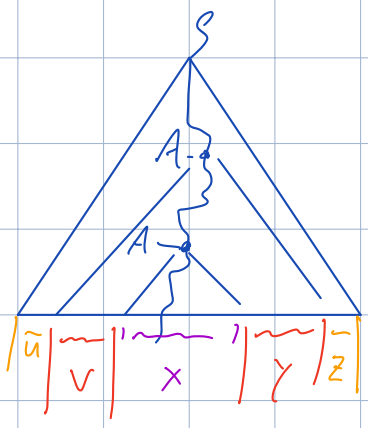
All internal nodes are variables



bt

If the tree is very tall, as long as the grammar doesn't change then: Variables will be repeated

if we make w longer the tree must grow taller



$w = uvxyz$

A is both 'x' and u, v

$A \to x$

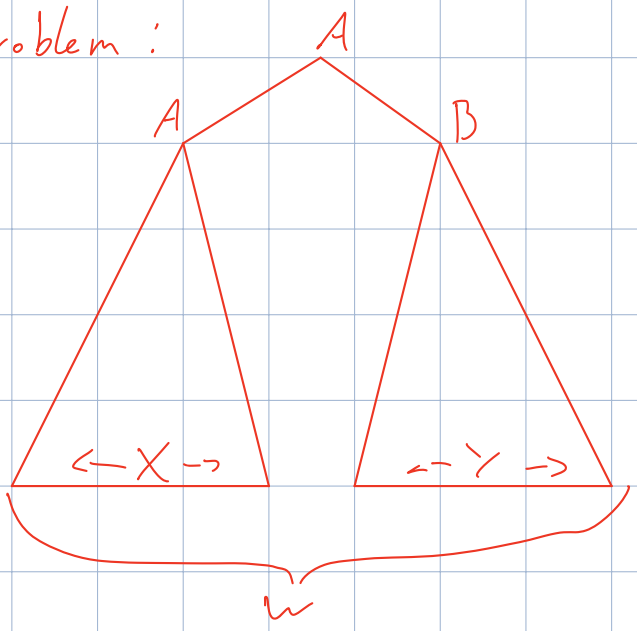
$A \to vxy \equiv vAy$

$A \overset{*}{\Rightarrow} v^i x y^i$

for any $i \geq 0$

$\therefore S \Rightarrow uv^i xy^i w \Rightarrow uvixyiw \in L$ for all $i \geq 0$

Problem:



Pumping Lemma for context free Languages:

let L be a cfL:

then there exists a pumping const p for L

for all $w \in L$ with $|w| \geq p$

then there exists a decomposition of w uvxyz

such that:

1. $uv^ixy^iz \in L$ for all $i \geq 0$
2. $|vy| \geq 1$
3. $|vxy| \leq p$

Thm: $L = \{0^n 1^n 2^n : n \geq 0\}$

suppose L = cfL

$\Rightarrow$ exists pumping const P for L

Choose $w = 0^P 1^P 2^P$

look @ all decomps into uvxyz

Case 1: v, y are only in 0's

pump $i = 2$ then no. of 0's > no. of 1's

Case 2, 3: v, y are only in 1's or 2's

pump $i = 2$ then no. of 0's < 1's or 0's < 2's

Case 4: v, y have 0's, 1's

pump $i = 2$ → 2's < 1's or 2's < 0's

Case 5: v, y's have 1's, 2's

Case 6: v, y's have 0's 1's, 2's is impossible

Show that cfl's not closed under $\cap$, complement

If $A \cap B$ is not cfl, proven

let A, B be cfl

$A = \{ a^n b^n c^k : n, k \geq 0 \}$

$B = \{ a^k b^n c^n : n, k \geq 0 \}$

$C = A \cap B = \{ a^n b^n c^n : n \geq 0 \}$

make cfg for A, B, C

A:

$S_0 \to SC$

$S \to aSb \mid \varepsilon$

$C \to cC \mid \varepsilon$

B:

$S_0 \to AS$

$S \to bSc \mid \varepsilon$

$A \to aA \mid \varepsilon$

Proving Complement:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

if cfls were closed under $\cup$ then $\overline{L_1}$ would be cf, $\overline{L_2}$ would be cf

$\cup$ is cf $\therefore$

$\overline{L_1} \cup \overline{L_2}$ would be cf

but $L_1 \cap L_2 \equiv \overline{\overline{L_1} \cup \overline{L_2}}$

which isn't cf

Constructive proof: L is c.f. but $\overline{L}$ is not.

$L = \{ 0^n 1^n 2^n : n \geq 0 \}$

in the language L we have all strings that have

$: i, j, k \geq 0$ $i \neq j$ and k

$w \in L$ if:

- w contains ba, cb, ca
- w is $a^i b^j c^k$ $i \neq$ both j and k

Take $i \neq j$

either $\overbrace{i < j}^{A}$ or $\overbrace{i > j}^{B}$

$S \to AC \mid BC$

$C \to cC \mid \varepsilon$

$A \to aAb \mid bX$

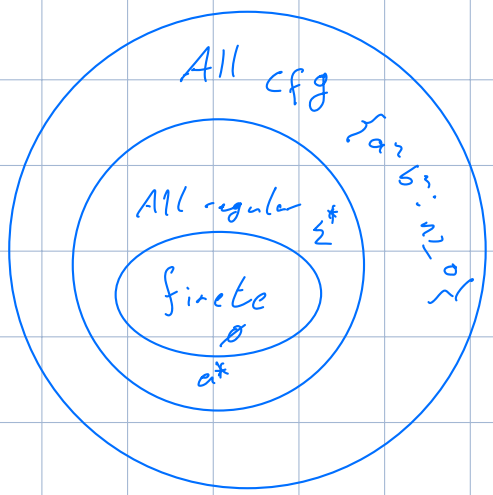
$X \to bX \mid \varepsilon$

$B \to aBb \mid Ya$

$Y \to Ya \mid \varepsilon$

} This is context free

Summary



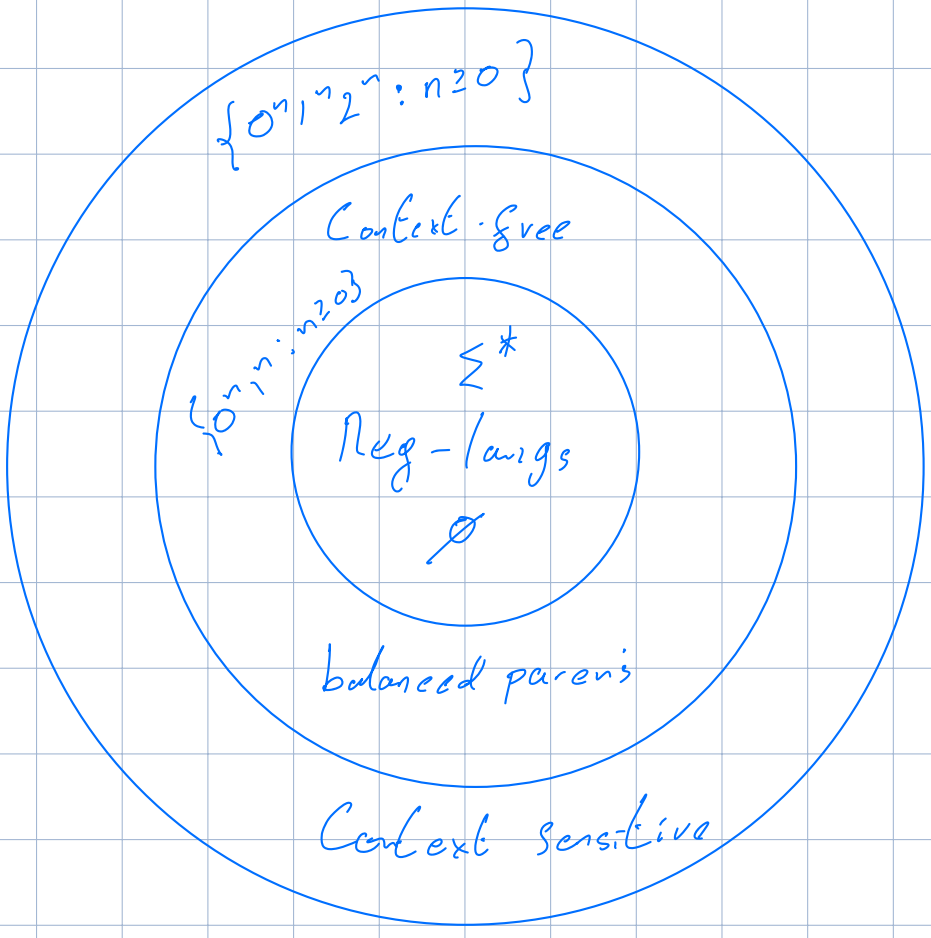
4 proofs of non-context free languages

Steps:

1. Assume L is a cfl
2. exists a pumping const. p for L
3. Pick some $w \in L$ $|w| \geq p$
4. Look at all decompositions of w into $uvxyz$ s.t.
   (a) $|vy| \geq 1$
   (b) $|vxy| \leq p$
5. find one i s.t. $uv^ixy^iz \notin L$

Decidability

- Turing machines
- Turing machine variants
- high level problems
- Church-Turing thesis
  - Importance
  - Truth
- Encodings of machines
- Decidability problems



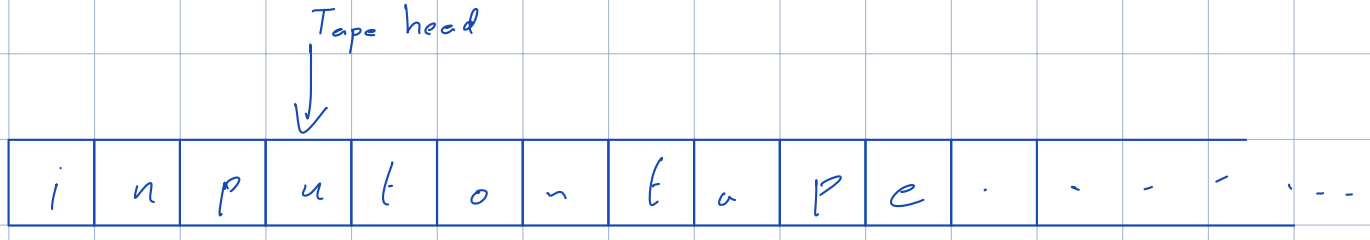
Can we find a "true model for CSL "cont. sensitive" languages for modern computers

Yes: if some language L is not possible in modern computation models it is not

deidable / we can't solve it.

The model is the Turing mach.

- TMs are state/transition models
- Deterministic
- Uses a tape: 1-way infinite e.g left hand end



Unlike stack we can look at any pos on tape that we want.

Tape head is magic pointer

It can:

- Change observed cell
- Move L/R one cell
- Transition:

  $q \to q'$ is a change or a move

Suppose TM at i where i is last input.

we can add a cell i+1 to tape as tape is unbounded even if input is finite.

Allocating new cell results in a blank symbol

Cells are never deleted.

TM formal definition:

- Q = finite set of states.
- Σ = input alphabet
- Γ = tape alphabet

$\Sigma \subseteq \Gamma$

- $q_0 \in Q$: start state
- $F \subseteq Q$ set of states that accept in Q
- Special States:
  - q_a - accecpt state $\quad q_a \neq q_r$
  - q_r - reject state.
- δ - transition function

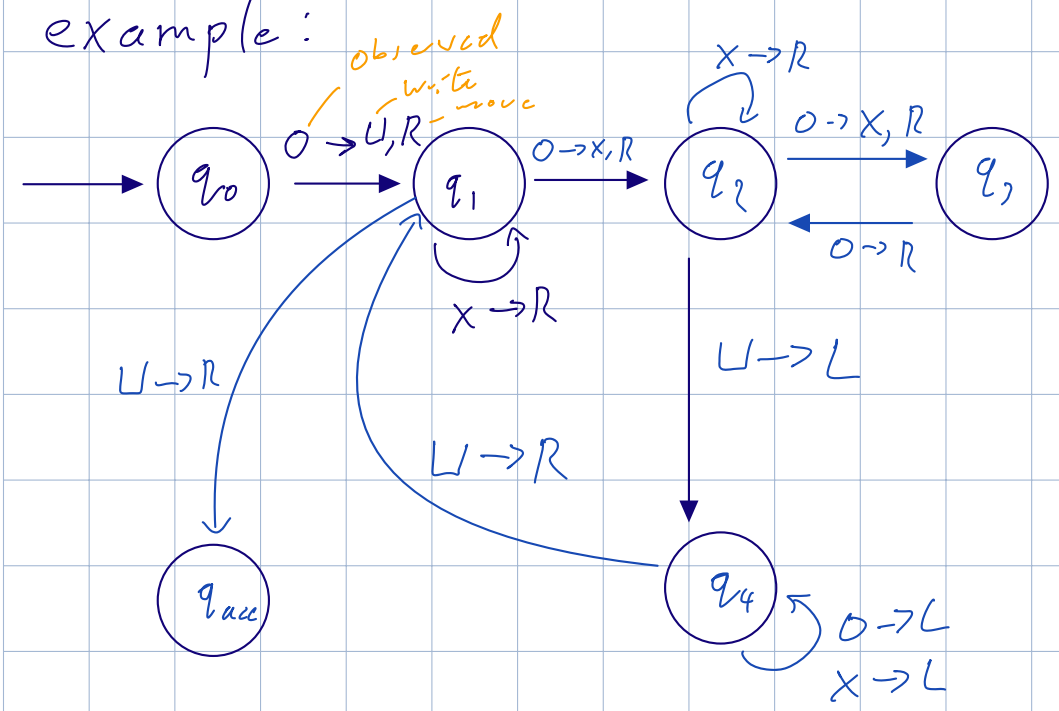
$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

direction of TH

- Q: old state
- Γ: what is in cell
- Q: after state
- Γ: what written

δ must be a total function, all $Q \times \Gamma$ have a transition.

example:



All unwritten δ transitions go to q_{rej}

$\Sigma = \{0\}$

$\Gamma = \{0, X, \sqcup\}$

Try compute a string

Input

$0\ 0 \to \sqcup\ 0 \to \sqcup\ X\ \sqcup$

$q_0 \quad\quad q_1 \quad\quad q_2$

$\to \sqcup\ X\ \sqcup \to \sqcup\ X\ \sqcup \to \sqcup\ X\ \sqcup$

$q_4 \quad\quad q_4 \quad\quad q_1$

$\to \sqcup\ X\ \sqcup \to \sqcup\ X\ \sqcup\ \sqcup$

$q_1 \quad\quad q_{acc.}$

Configuration.

Configurations track:
- Cell contents
- Current state
- Position on tape

$w \in \Gamma^* Q \Gamma^*$

w is symbols · current state · symbols

A seq of configurations is called a computation.
each config yields another until end of comp.

C_i is a config.
C_1 computation, $= C_0 C_1, \ldots C_m$
C_0 is start config.
C_i yields C_{i+1} for all i

let C be a computation.
C is accepting if C_m contains the accept state
C is rejecting if C cont. the reject state
C is halting if its accepting or rejecting

Language of machine M

$L(M) = \{ w \in \Sigma^* : M$ has an accepting comp on $w \}$

A Language L is recognisable if it is the language of some Turing machine.

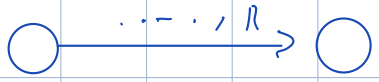
A language L is decidable if it is the language of some Turing machine M and M halts on all inputs e.g. it either rejects or accepts

If L is decidable it must also be recognisable

TM variant:

Stay put Turing machine.
Tape head stays in the same cell

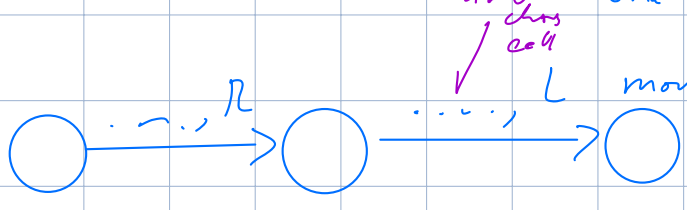
STM can simulate a normal TM



can simulate nominally

let S be stay put

break into 2 transitions one moves left, the other moves right



so R is OK because R is always possible

ex: Left reset Turing machine

L, R transitions as normal

Reset → moves to left most cell

special transition

LRTM can simulate a normal TM because it does not have reset for its transition.

Can a TM simulate an LRTM?

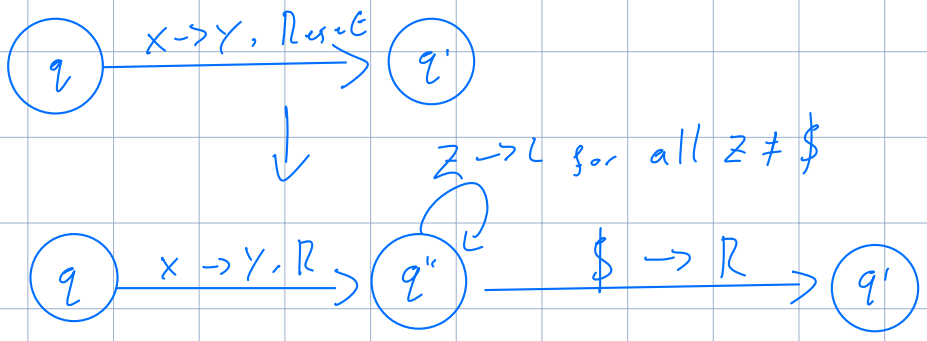
↳ Yes

↓ move contents of pos over one

↑

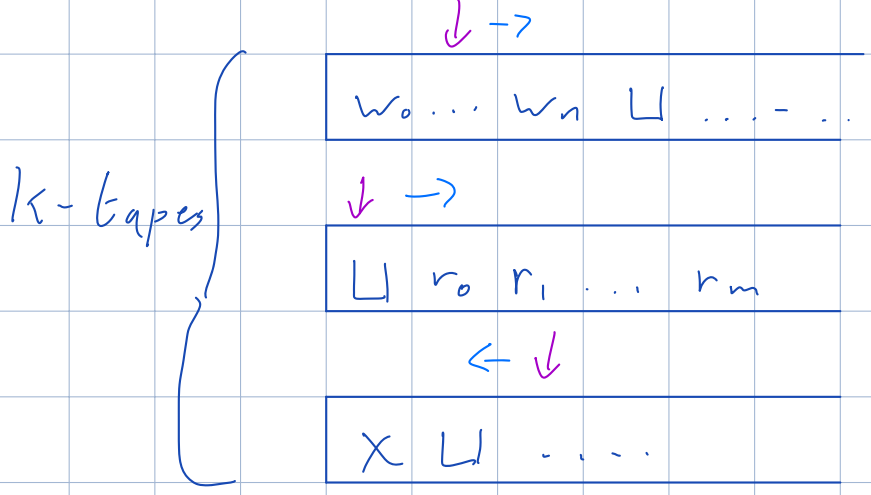
This is only used once in the entire input

We can make a seq. of transitions left until we read $ ∴ making that seq a LRTM in effect.



Variants:

Multi-tape



$$\delta: Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

Multi-tape TM can sim a TM

Use only 1 tape to do work

k-1 tapes just move R.

Can a single tape sim a multi-tape?

Yes

Use # as delimiters between tapes diff work.

#<tape 1> #<tape 2>#...<tape k># ⊔ ...

Problems:

1. Only 1 tape head
2. What if 1 tape allocates new cell?
3. What if 1 tape tries to move left off track

Add new symbols to tape alphablet.

0 1 2 # 0 0 # 1 1 2 }

$\Gamma' = \Gamma \cup \{\dot{w} : w \in \Gamma\}$ e.g. $\dot{1}, \dot{2}, \dot{3}$ etc.

The dot will track the respective simulated tape heads

sim head 1 ↓ sim head 2 ↓ sim head 3 ↓

0 $\dot{1}$ 2 # 0 $\dot{0}$ # 1 $\dot{1}$ 2 }

To carry out a transition δ To-do

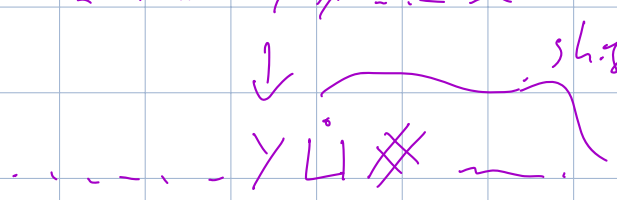
1. look at k cells
2. Which k items to write
3. which directions to move for each dot/tape

4. Whether to allocate new cells
5. Dot the corresponding new cells

How-to:

1. Go from beginning to end in 1 pass

2/3. Embedded in δ

4. 

Non-deterministic Turing Machine

Assume single tape

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

accept w iff some seq of choices to get to $q_{acc.}$

A NTM is a decider iff all possible $w \in L$ either accept or reject. i.e eventually halt

Decider: machine for decidable Language
halts for any string

Recogniser: machine for recognisable language
has to halt only on strings in language

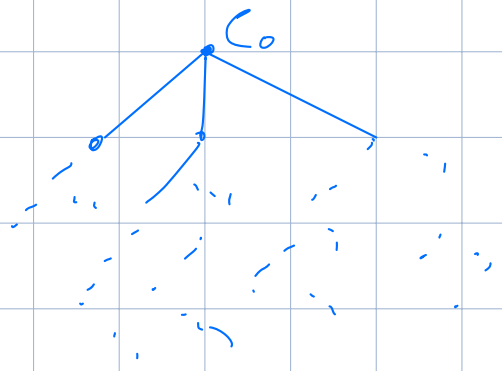
NTM's don't have to make non-det. choices
∴ can sim a norm TM

So: Can a normal TM sim a NTM?

- Yes

look at a computation tree

nodes are configurations



If q_{accept} is in the comp-Tree then we know there is the potential for the machine to stop

Doing a depth first search, trying every path we may spend a long time try to find the right path.

A breadth first search instead can tell us potentially a lot quicker if there an acc. state

Steps: look at L_1 for q_{acc}: if found good

else: look at L_2

Alg: BFS one level at a time

If q_{acc} found accept

If no new nodes found reject.

Are TM's the almighty?

They can do addition...

↳ proof with add by 1

$w = 0\ 1\ 1\ 0\ 1\ 1$

$\downarrow +1$

$0\ 1\ 1\ 1\ 1\ 0$

Algo: start at end, move backwards

While char is 1 change to 0, move L

If char is 0, change to 1

TM for $x+y$

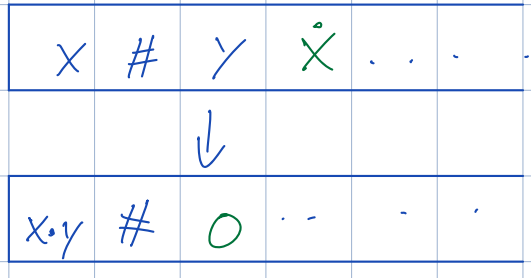
Tape | x # y . . .

$\downarrow$

x+y # 0 . . .

To do this we continually add 1 to x, subtract 1 from y until $y=0$

for $x * y$



Repeated addition: dec y: place the val of x into ẋ dot, add 1 to x dec. ẋ until 0, repeat until $y=0$

x^y is simply repeated multiplication

$\max(x, y)$: compare char for char against $x + y$

If we have have any math. operation on a comp.
then a Turing machine can do it.

Church-Turing Thesis:
Our intuitive notions of algos and TMs are equivalent.

- We can make any algo run in a TM, recreate any TM in a modern pc.

Not provable but intuitive

Problems for Turing Machines

"high-level" algos – avoid all unnecessary details
↳ pseudo code

encodings – taking a machine or a grammar or smth
↓ conv. into
string that represents it

e.g. DFA: $Q, \Sigma, \delta, q_0, F$
↓
$\# q_0 q_1 \dots q_m \# s_0 \dots s_k \# q_0, a\, q_3, q_0, b, q_6, \dots \# q_2 \# q_0 \#$

states in q | input chars | δ | finals | start

As long as we know the order of input we can decode the dfa
details are irrelevant.

We have a dfa / nfa / cfg / TM etc. M

M is the obj we are interested in
$\langle M \rangle$ is the encoding string

Decidable problems

↓ acc. problem

$A_{dfa} = \{ \langle M, w \rangle, \ M \text{ is a dfa and } w \in L(M) \}$

↳ is decidable

M accepts w

e.g: $\# \langle M \rangle \# w_1 \ldots w_n \#$

"high-level" desc.

"on input $\langle M, w \rangle$ where M is a dfa and w is a string in Σ^*

1. Repeatedly track what state M is in after reading each char of w
2. If last state is $\in F$ of M, accept else: reject

Show that each step takes finite amount of time
↳ shows the whole comp takes finite amt of time

Since this machine runs in a finite amnt of time it is a decider for A_{dfa}

ex

$A_{nfa} = \{\langle M, w\rangle : M \text{ is an nfa and } w \in L(M)\}$

runs in single time

1. Conv. M into an equivalent dfa D Using Powerset construction

also runs in $\mathcal{T}$.

2. Run decider for A_{dfa} on $\langle D, w\rangle$; output same answer.

January Exam 2

1. Regular Languages

Understanding what computers can do
- does power matter
- Are problems solvable or unsolvable.

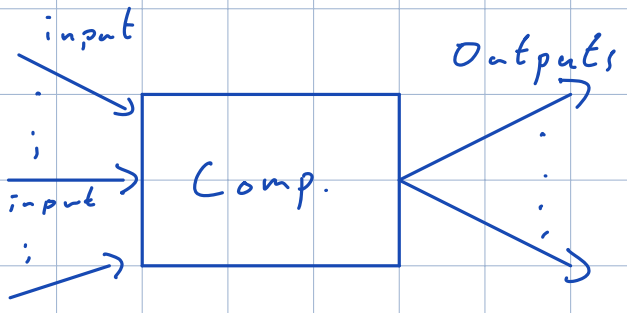
We can show that there are in fact unsolvable problems.
There are only countably many algos
each algo solves only 1 problem
There are uncountably many problems
∴ no. of algos < no. of problems

Can we find actually find an unsolvable problem?
And even if we find a problem that is solvable are there problems such that a computer with limitations is unable to solve it
Limitation examples
- Power
- Memory
- Instruction set (no "add")

We need a formal def. of both a computer and what computing something is.

What is a computer?

A device that converts inputs to outputs



The computer reads inputs does things based on inputs and produces outputs.

We can actually restrict the no. of inputs to be 1 without affecting the output.

Why?

Let input count = 3

i1 : 0 1 1 0 1 0 . . . .

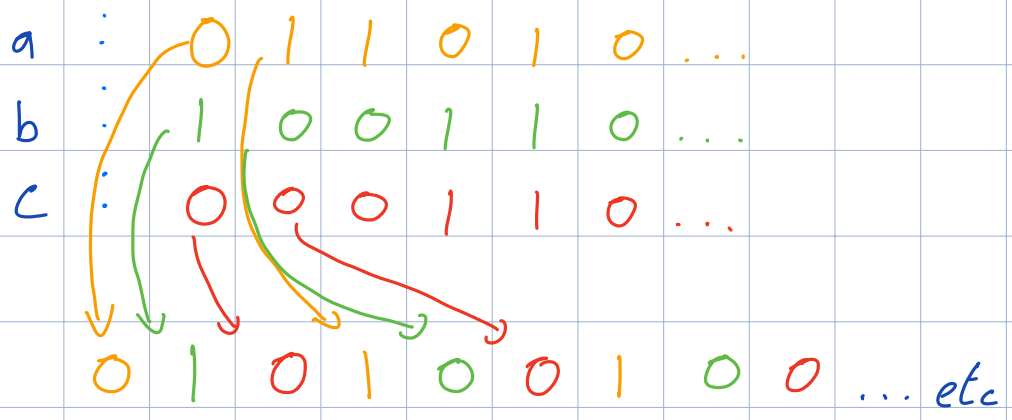
i2 : 1 0 0 1 1 0 . . . .

i3 : 0 0 0 1 1 0 . . . .

We can multiplex these into a single input stream

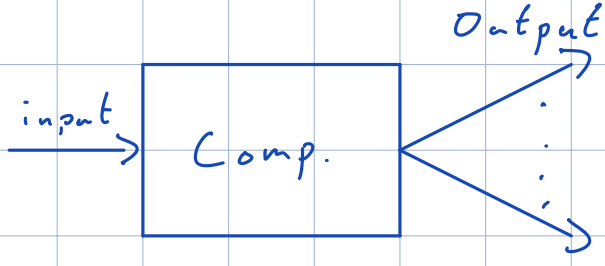
Multiplexing:

look at the first bits of all the inputs and create a single stream from each 1.



Then take the second bits in order of each stream, then 3rd etc.

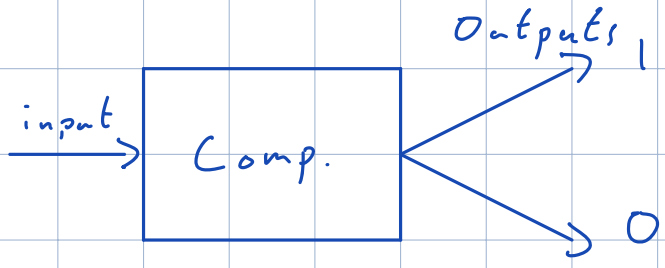
We do not lose generality by multiplexing and the result is a model:



Why stop there? We can also multiplex the output sequence!

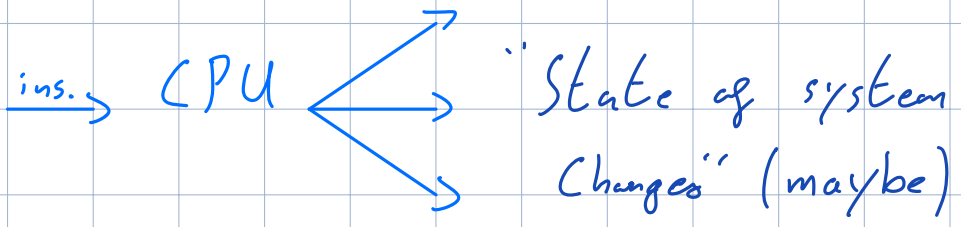
Giving us one io stream.

Not only can we reduce our output to 1 stream we can actually use a single symbol 1 or a 0, either accepting the input or rejecting it.



So what can we do with this comp. model?
Lets think about what is happening in our computation part.

We have our cpu and an instruction is coming into the cpu. which is executed.



Once executed a no. of things can happen.
but what we know for sure is that the state of the system will change.
* The system could transition to the same state its currently in.

Machine State:

We can think about the comp. being in a certain state:

- The contents of the register, the memory.
- Whats on the storage drive etc.

Ultimately its a certain state of the model at a given moment and when we proc. an instruction then the state will be updated to a new state or stay at the same one.

We then can see the program then is just a bunch of possible states and we move from one to another when we read an inst.

Lets think about the states and their transitions. How can we model the journey from one state to another

This is the core of Theory of Comp the theory of state transitions.

When analysing state transitions we make 3. assumptions:

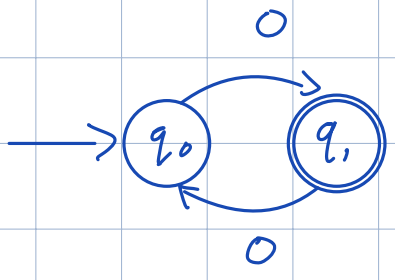
1. fixed internal memory
2. Once done input, the machine stops.
3. Machine starts in some state.

We need assumption 1 because w/out a fixed memory length the model of computation changes.

Assumption 2 exists because we can assume w/out loss of generality that we read

left to right w/out backtracking.
- It always moves forward.

example:
- 2 states q_0, q_1



A single circle is a non-accepting '0' state while a double circle is an accepting '1' state.
The arrows represent transitions.
Very simple machine w/ 2 memory states

Circles are states
'q_0', 'q_1' is a state name
↝ = Transitions
The 1st arrow from nowhere is a start arrow pointing to a single start state

We can look at an example input: '000'

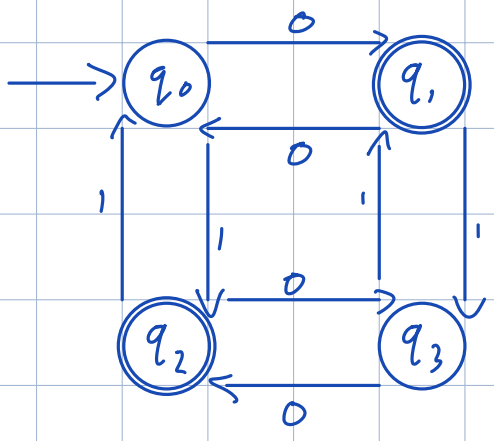
In	state
S1	q_0
0	q_1
0	q_0
0	q_1 Accepting

After reading the input of '000' we have completed 3 transitions and finished in an accepting state q_1

Example 2: Our input alphabet now includes 1 and 0

each state has a trans for all possible inputs

- This means the machine is completely deterministic.



lets trial some inputs for testing the machine

'0 0 0'

In	St
→	q_0
0	q_1
0	q_0
0	q_1 ✓

'0 1 1 0'

In	St
→	q_0
0	q_1
1	q_3
1	q_1
0	q_0 X

'1 0 1 1 1'

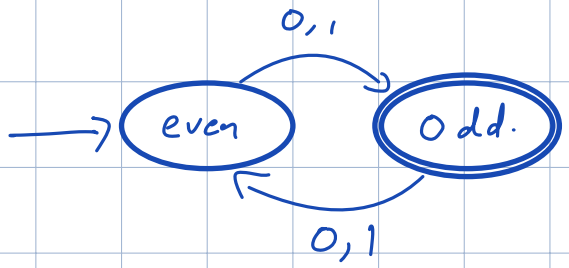
In	St
→	q_0
1	q_2
0	q_3
1	q_1
1	q_3
1	q_1 ✓

This machine acc. any odd length string.

Any even length input will result in either q_0 or q_2

- Machines solve only one problem but many diff. machines can solve the same problem.

The above machine can be simplified into a 2-state sol:



We call any string that ends up in an acc. a accepting string.

So which states acc.?

- '0' ✓
- '1' ✓
- '10' X
- '001' ✓

} looking @ some acc. strings + the machine we start to get a pic. of the machine acc. odd len strings

Introducing the empty string:

We denote the empty st. w/ 'ε' and it is used to denote a 'none' input

In the machine above ε is not accepted because we do not execute any transition at all

Some machines will accept ε-transitions

Defining machines formally:

The 1st formal def of a comp model to look @ is a Deterministic finite State machine which is a 5-tuple: M

M has:

1. Q - A finite set of states
2. Σ - Input alphabet - finite
3. $q_0 \in Q$ - the start state in Q
4. $F \subseteq Q$ - A set of acc. states in Q
5. δ - State transitions / transition function.

If the states weren't finite then we would have a diff model of computation.

How to represent δ as a function:

We have to know:

- Which state in Q we are in
- What to read

$$\delta : Q \times \Sigma \longrightarrow Q$$

Q: state to read; Σ: symbol to read

This is a total func., no matter what input and current symbol we are on we know which trans

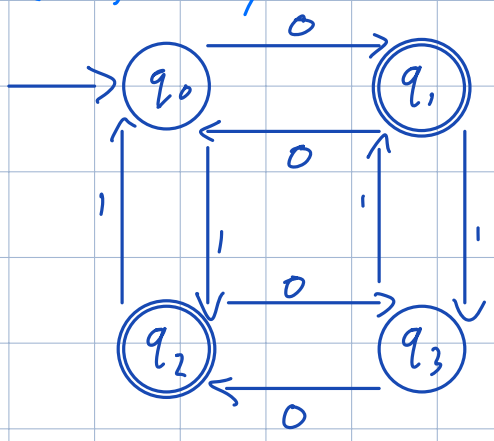
to make

Total function: every possible state / input pair is defined

Does not mean that all states are reachable

States can be transitioned to by multiple transitions.

example: define formally:



Q: $\{q_0, q_1, q_2, q_3\}$

Σ: $\{0, 1\}$

q_0: q_0

F: $\{q_1, q_2\}$

δ: $q_0 \times 1 \to q_1$

$q_0 \times 0 \to q_2$

$q_1 \times 1 \to q_3$

$q_1 \times 0 \to q_0$

$q_2 \times 1 \to q_0$

$q_2 \times 0 \to q_3$

$q_3 \times 1 \to q_2$

Transition table

Σ	0	1
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_2	q_1

↗ Simpler layout

q_3 x 0 → q_2

filled tbl = total func.

Further DFA definitions

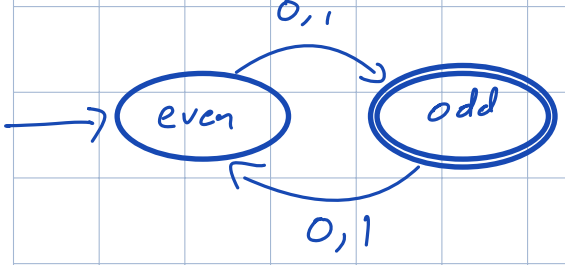
The computation of a dfa M on input w
= the sequence of states visited
- States can repeat

e.g. above machine on w = '010'
= q_0, q_1, q_3, q_2

Computations are accepting if the final state in the computation seq is $\in F$
The comp is acc if the final state is acc.

A language is the set of strings
A language is regular if it is accepted by some dfa.
All strings in the language are accepted by M.
The set of strings w/ an accepting competation.

example



The language of this dfa. is the set of all strings of odd length

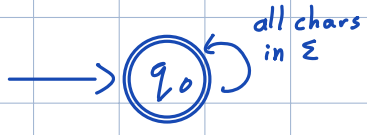
$L = \{$ all strings over Σ where $w \% 2 = 1\}$

We can say that L is regular

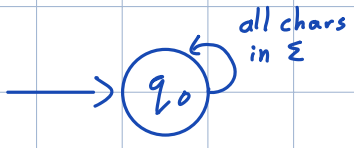
In a dfa the number of computations is always
1. because it is deterministic.
The same string will always produce the same path
therefore the same final state of acceptance.

examples of regular languages:

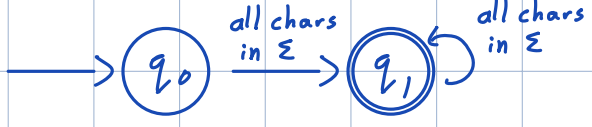
1. Σ^* - the set of all strings over Σ
Claim Σ^* is reg, proof: dfa.



2. $\emptyset$ - the empty set
claim $\emptyset$ is reg, proof: dfa



3. $\Sigma^* - \{\varepsilon\}$ - All strings except ε only
which is all non-empty strings



Operations on languages:

We know that languages are sets and we can therefore apply set operations to them but when will these operations maintain the properties of reg L's? If we do an operation on one or more regular languages is the result still regular.

Set operations on regular languages

Union: We can union 2 sets together $A \cup B$

let C be a set of languages

we say that C is closed under an operation $\oplus$ if applying a given op $\oplus$ to languages in set C results in a language that is a member of set C.

Simple closure op:

Integers are closed under add., multiplication but not division.

e.g $i + i = i$, $i \cdot i = i$, $\frac{2}{3} \neq i$

Regular operations:

Operations that are standardised for many langs.

Union: $\cup$

Concatenation: $+$

Star: $*$

We want to prove these ops

Union: $L_1 \cup L_2 = \{x : x \in L_1 \text{ or } x \in L_2\}$

The union combines both sets and elims duplicates.

Concate: $L_1 \cdot L_2 = \{xy : x \in L_1, y \in L_2\}$

New set is strings that start w/ string from L_1 and end w/ str from L_2.

e.g. $L_1 = \{0, 00\}$

$L_2 = \{1, 11\}$

$L_1 L_2 = \{01, 011, 001, 0011\}$ but not '10'

Star: $L_1^* = \{x_0 \ldots x_n : x \in L_1\}$

e.g

$L_1 = \{0\}$

$L_1^* = \{0, 00, 000, 00 \ldots 0_n$

L_1^* is also written as $\bigcup_{k \geq 0} L_1^k$

$L_1^0 = \{\varepsilon\}$

$L_1^k = L_1 L_1^{k-1}$

A non-reg operation:

The complement of a lang L is $\overline{L}$

$\overline{L_1} = \Sigma^* - L_1$

thats all strings in the star of the alphabet sigma Σ minus those in L_1

Every string not in L

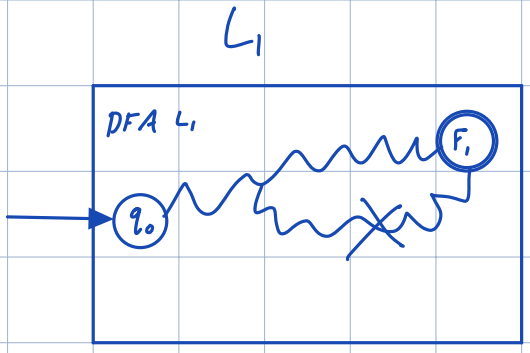
We know then that regular languages are closed under compliment.

Why is this true?

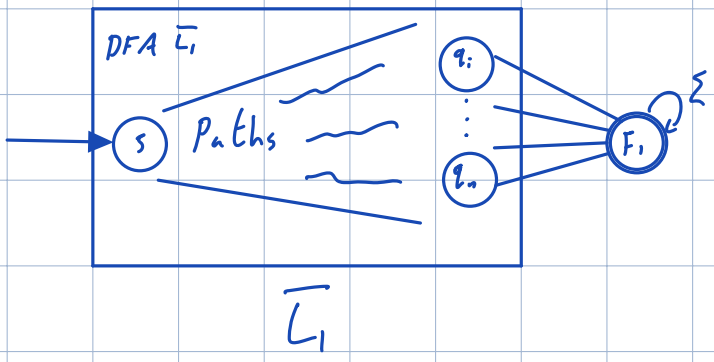
Because we can easily 'negate' any dfa to create a compliment dfa

Start w/ the knowledge that any reg lang by definition must have a dfa.

Then show that that dfa can be converted into a dfa of it's complement.

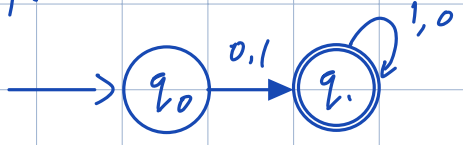


In a dfa there is only one possible path to the same state. So given that if we simply change all states b4 their accepting state to accepting and add additional acc. state that loops on ξ after it is the complement.

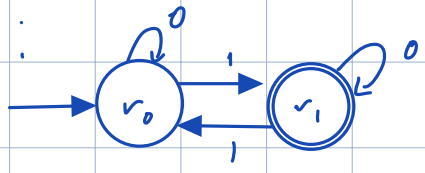


Union: The union of 2 dfa's

L_1:



L_2:



lets look at $w = 011$

trace it for both machines

L_1: $q_0 \to q_1 \to q_1 \to q_1$ acc.

L_2: $r_0 \to r_0 \to r_1 \to r_1$

So we know that the union of the 2 dfas should accept this string 011 because it is acc. in L_1

The idea for showing closure under union is to try to simulate both machines at the same time.

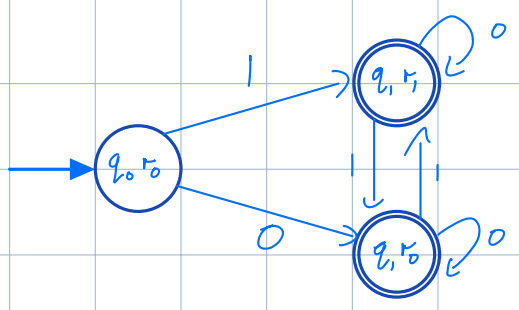
Theorem to prove: RLs are closed under union

Simulate both machines at the same time.

If we try to sim both at same time then we are in start state of both $q_0 r_0$

lets use a transition tbl to sim

	1	0
$q_0 r_0$	$q_1 r_1$	$q_1 r_0$
$q_1 r_1$	$q_1 r_0$	$q_1 r_1$
$q_1 r_0$	$q_1 r_1$	$q_1 r_0$



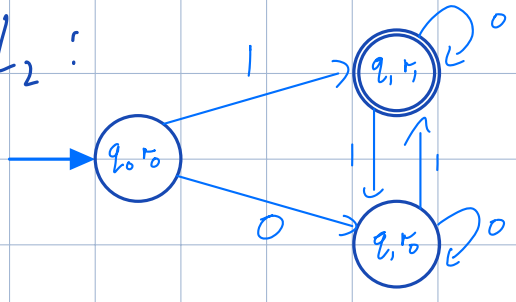
The final states of the union machine is a state in which either of the component machines are final.

The above proc. is called the product construction which is the cartesian product of sets

Taking the product of state sets will not give the sum of both sets because not all states are reachable e.g $q_0 r_1$

Intersection of 2 regular languages will have the same dfa layout as the union except only states that are accepting in both L's will be final

$L_1 \cap L_2$:



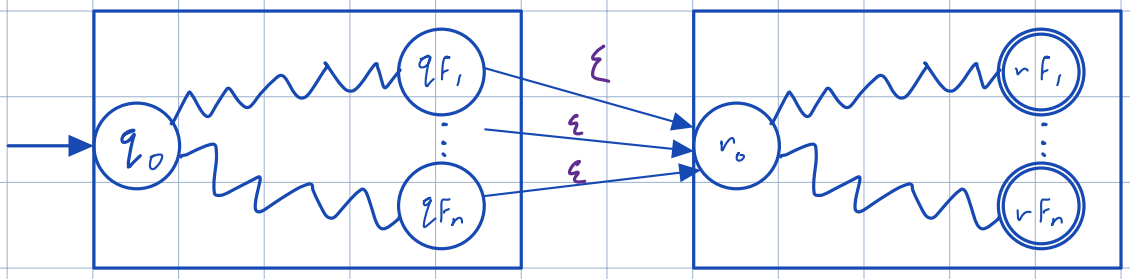
Note here $q_1 r_0$ is no longer accepting.

We could also use DeMorgans laws which states that:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Because reg. langs are closed under complement and union it must be closed under $\cap$

Concatenation: $L_1 \cdot L_2$ $\{xy : x \in L_1, y \in L_2\}$



All formerly final states of L_1 qF_i are now transitioned to the start state of L_2 through ε-transitions, thus concatenating the start of L_2 onto the end of any L_1 x

$$\underbrace{w_1 w_2 \dots w_{i-1} w_i}_{\in L_1} \underbrace{w_{i+1} \dots w_n}_{\in L_2}$$

We instantly jump from L_1 to L_2

But! My sol above jumped the gun because now $L_1 \cdot L_2$ is no longer deterministic because we can take any accepting path in L_1

The sol would be non-deterministic creating a Nondeterministic finite Automaton.

NFA:

nfa's have any no. of transitions from any state.
so instead of having 1 for each char. we could have multiple.

Above I used an ε-transition in my sd. this, however is only actually allowed by nfas, they do not exist in dfa's.

If an ε-transition is taken, no char is read/consumed.

Non-det. means that if some way to make choices to end in a final state, we will accept.
The NFA will choose an acc. path if possible
- The NFA will always make one choice at a time.

NFA caveats:

The no. of computations could be anything incl. the no. of accepting computations:

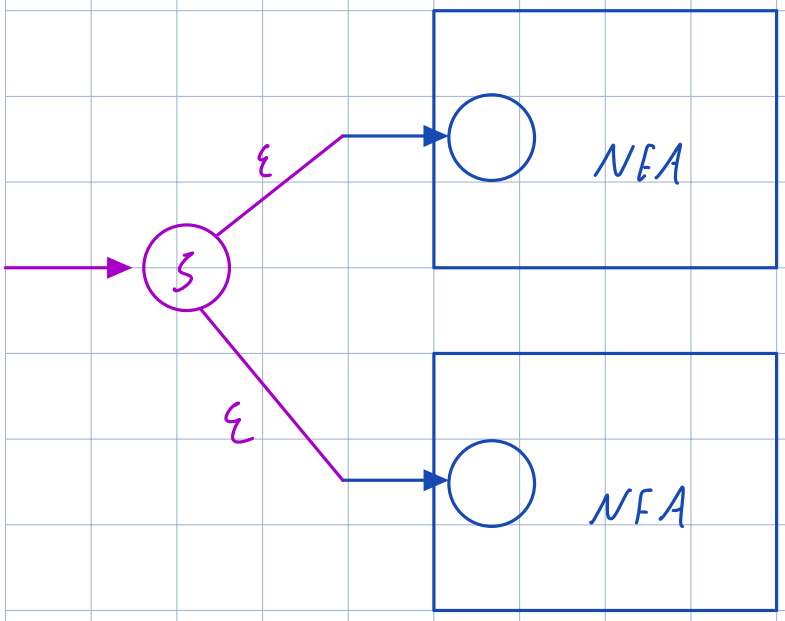
ex:

We could take ∞ many ε-transitions before moving to q_1

The infin trans prop. make it impossible to take the complement as before by switching the acc or reject states.

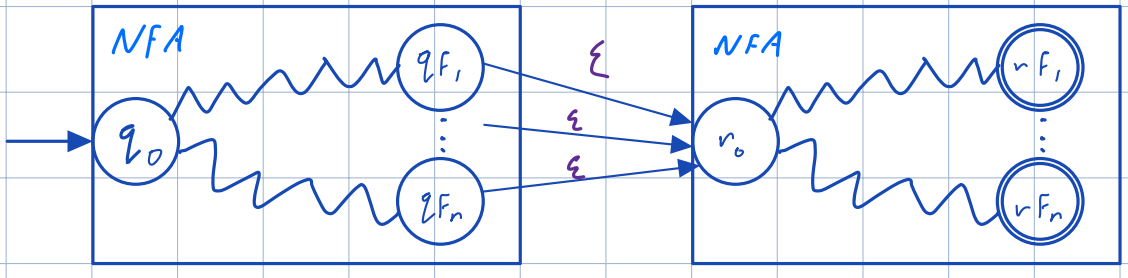
We can however do the reg ops easily:

Union:



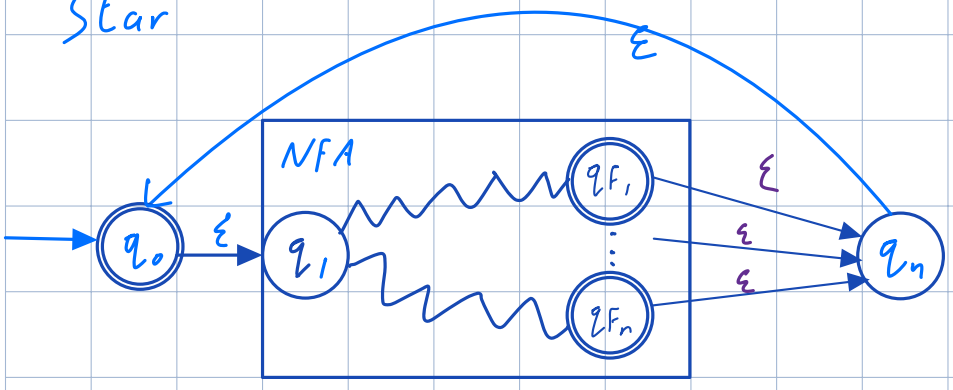
here we had 2 nfas, each w/ their own start state
We create a new SS by our nfa's with ε-trans to the prev unconnected states

Concatenation:



here all formerly final states in the 1st NFA now have ε-trans to the former ss of NFA 2.

Star



We simply allow the machine to ε-trans back to the new SS and allow for the empty string to be accepted.

Think of nfa as an arm or body part, if we want to pick something up the arm will transition through states to do the correct thing without us knowing the exact path it took.

Connection between NFAs, DFAs

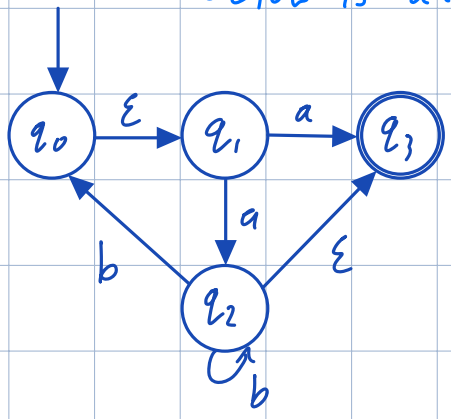
What is the rel. between these 2 types of Automatons.

Every DFA is an NFA already (sort of)

Does every NFA have a DFA equivalent?

Example:

below is an NFA



to make a dfa we need to elim the ε-transitions

Need to have a, b transitions from q_0

We need to simulate the machine of the nfa w/ out ε-trs

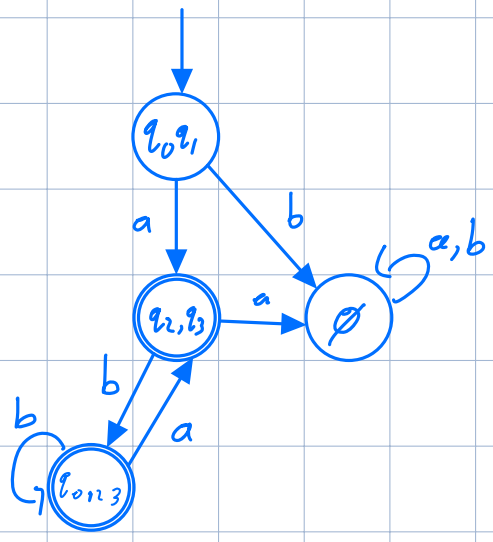
Because in our nfa q_0 has ε-t to q_1 we start in a state q_0, q_1 to rep the nfa possibilities.

If we read an 'a' on our S q_0, q_1 we could move to either q_2, q_3 so that becomes our next state

have reading a 'b' does not bring us to any state at all.:

we make a new state, the empty set.

The $\emptyset$ then self loops



This is called the pow. set construction because any 1 of the states in the new dfa. could be any possible subset of the new states

We esp. performed a breadth 1st search of the NFA to construct a DFA.

We can convert an DFA from a NFAs

NFAs : are equivalent to DFAs

DFAs / NFAs accept a language:

If we provide a string to an automaton it is either accepted, in the language or rejected, not in L.

Does not give a real reason why a string was acc. or not.

Describing an L using English is imprecise

We want to be able to describe a lang L precisely

Regular Expressions : Regex.

Let Σ be an alphabet, Regex could be 1 of 6 forms.

R can be:

- $R = \emptyset$ — empty set
- $R = \varepsilon$ — epsilon
- $R = 'a'$ — A char
- $R = R_1 \cup R_2$ — The union of 2 other Regex
- $R = R_1 \circ R_2$ — The concat of 2 Regex
- $R = (R_1)^*$ — The * of 1 R

ex: $(0 \cup 1)^* 101$

= Any string of 0s, 1s ending in 101

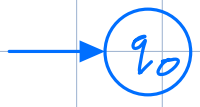
$(01)^*$: All str with the pattern 01 repeated 0 or more times

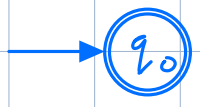
Theorem: NFAs and Regex's are equivalent.

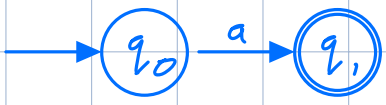
Proof using Thompson Construction

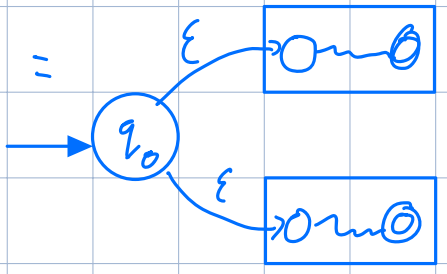
Lemma: Every Regex can be conv. to an nfa.

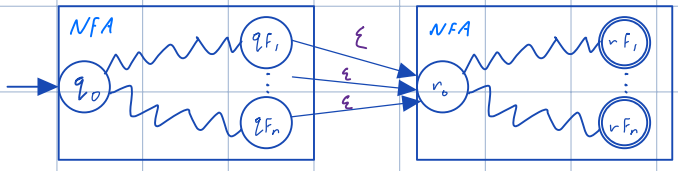
Regex has 6 types, find equiv nfa

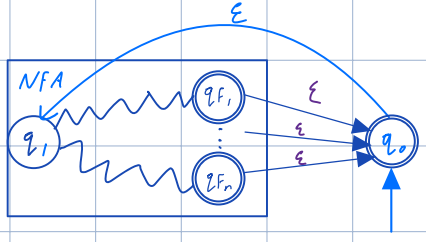
1. $R = \emptyset$ $M =$ 

2. $R = \varepsilon$ $M =$ 

3. $R = 'a'$ $M =$ 

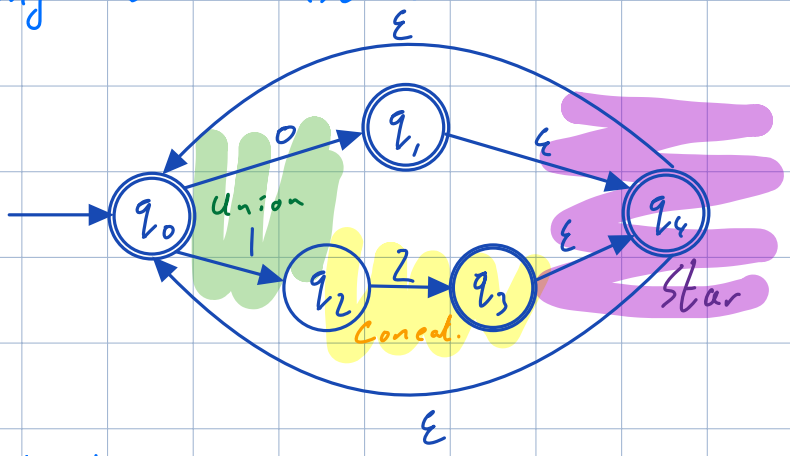
4. $R = R_1 \cup R_1$ $M =$ 

5. $R = R_1 R_2$ $M:$ 

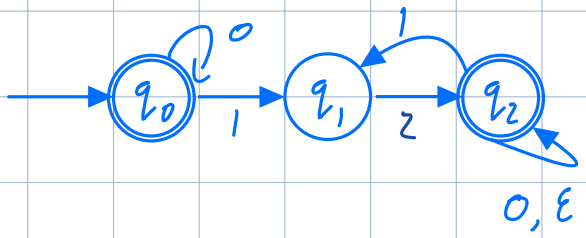
6. $R = (R_1)^*$ $M:$ 

Example: $(0 \cup 12)^*$

Using above constructions



Actual NFA

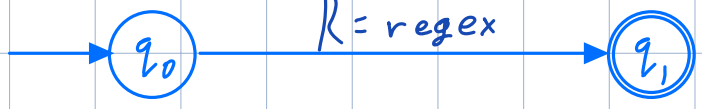


Lemma:

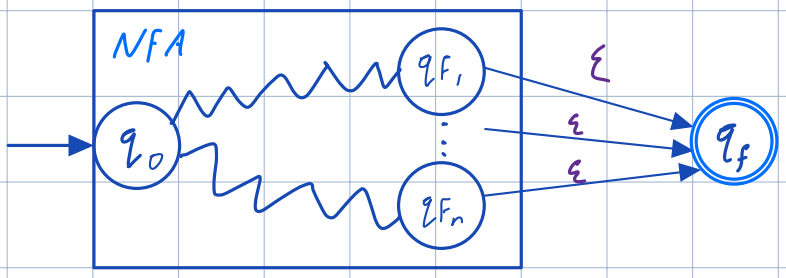
How do we show every nfa can be conv. into an equivalent regex

We can't perform a simple extraction because an nfa might have a complex construction

We need to abstract a potentially complex nfa down to



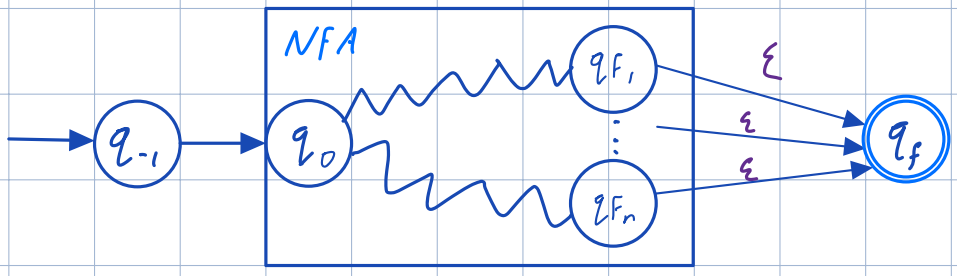
Start by showing that w/ out loss of generality an nfa can have exactly one final state:



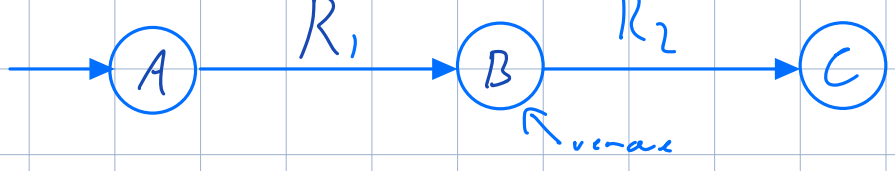
- new final state

We make all former final states non-final and make a new final that the former ε-trans to.

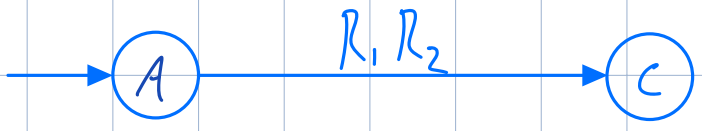
We also show that wlog we have a start state w/ no incoming transitions.



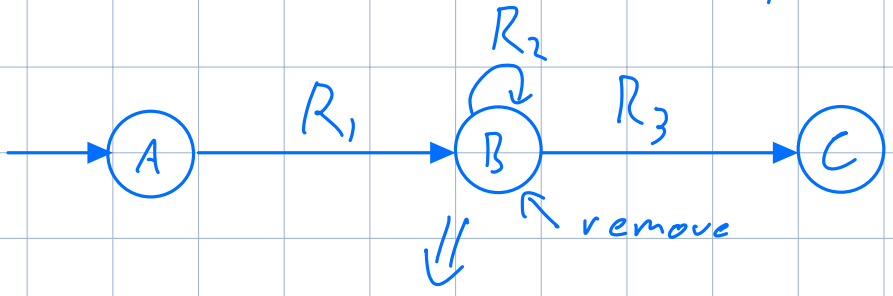
Now we have to simplify the nfas internal states let say we had A goes to B on some Regex R

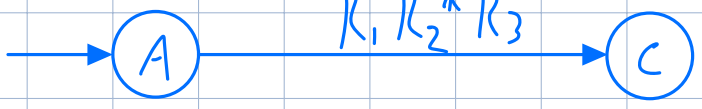


If we wanted to remove B from the above path we could concat. $R_1 \cdot R_2$ into a single trans.



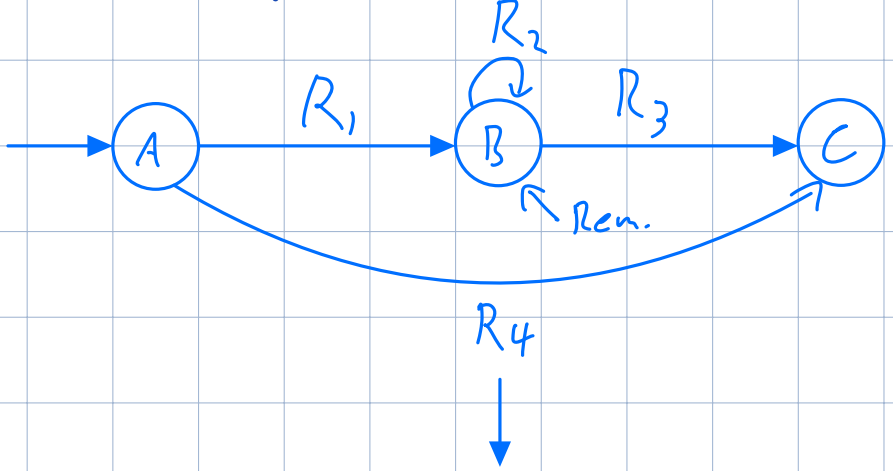
What if we had a self loop on B





We can use the * to conv. into 1-trans.

A tran. from A to C

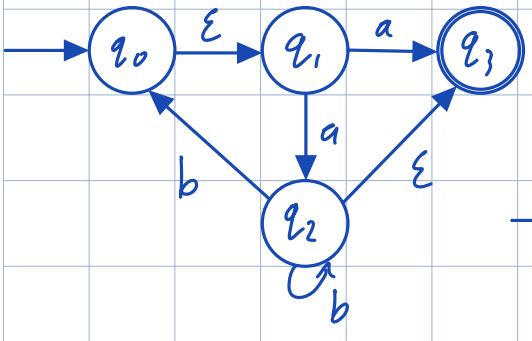


If we cont. this proc. we will inevitably be left with some R between q_0, q_n that will represent the nfa
Called a generalised NFA

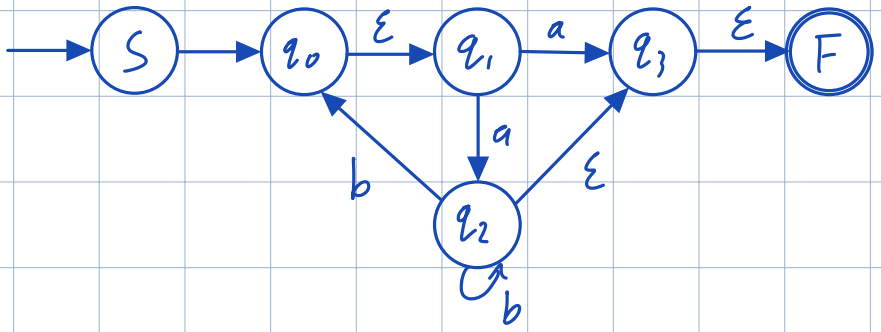
Generalised NFA:
Every transition is a regex, nothing going into start, nothing leaving final

NFA to regex is repeatedly ripping states and adding regex transitions as necessary until we reach

Example:



1st: new start, final states



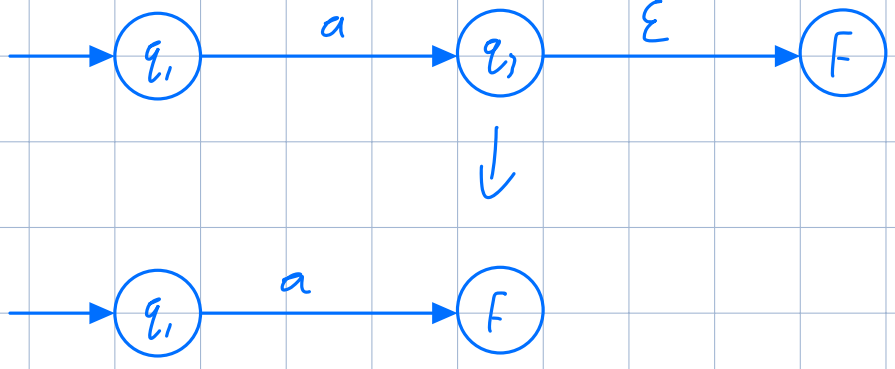
2. Rip states:

Start w/ least trans no. state: q_3

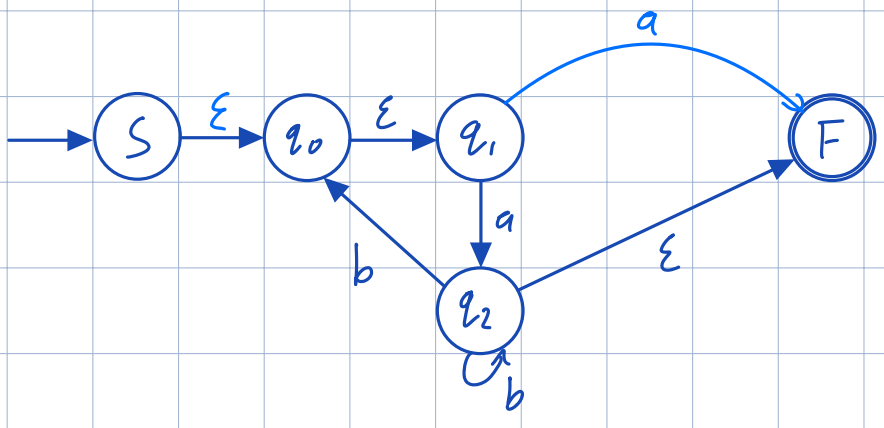
Use io list

In	Out
q_1, q_2	F

Choose 1 from in list, 1 from out and make a trans for them, e.g. q_1, F



for $q_2 \rightarrow F$: ε

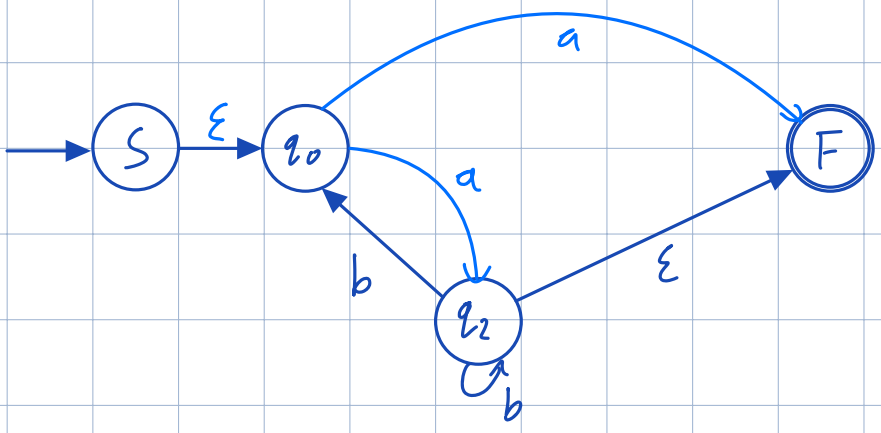


Now rip: q_1 :

In	Out
q_0	q_2, F

$q_0 \rightarrow F = a$

$q_0 \rightarrow q_2 = a$



rip q_2 :

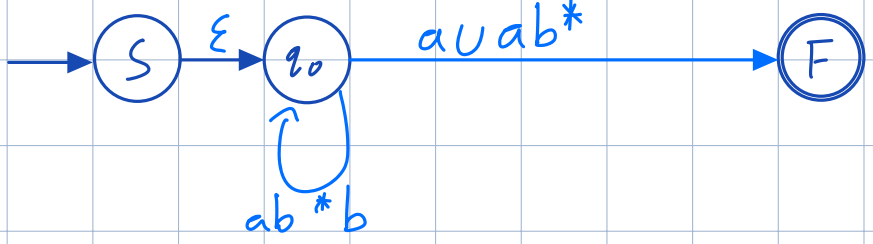
In	Out
q_0	q_0, q_2, F ($q_2 \to b^*$)

$$q_0 \xrightarrow{a} q_2 \xrightarrow{b^*} q_2 \xrightarrow{\varepsilon} F = ab^*$$

$$q_0 \xrightarrow{a} q_2 \xrightarrow{b^*} q_0 = ab^*b$$

We have 2 $q_0 \to F$

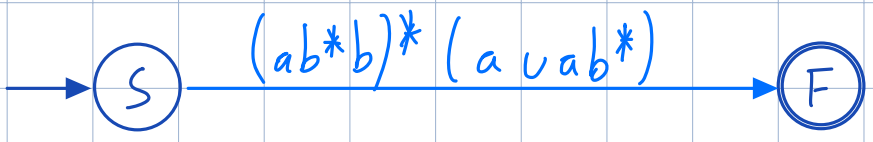
lets union $a \cup ab^*$



now rip q_0

In	Out
S	q_0, F

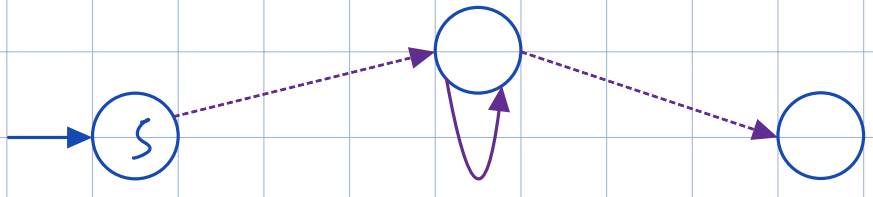
$$S \to F = (ab^*b)^*(a \cup ab^*) = R$$



What other strings are accepted.

Suppose w is a str. acc. by DFA M
what else is accepted?

Say we're reading w and we have a loop



We could acc. any string with or more internal loops in side.

If the loop exists we acc.

$w_1, \ldots w_{i-1}, (w_i, w_{i+1}, \ldots w_k)^*, w_{k+1}, \ldots w_n$

∴ There are infinitely many words

When can we guarantee such a loop exists?

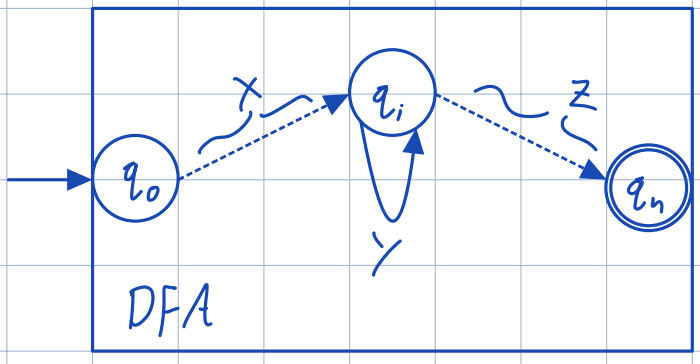
If a word of length greater than the no. of states exists then we know that the dfa conts a loop

Accept $\geq n$ length string means we have seen $n+1$ states because we always start in q_0

If we see $n+1$ states then a loop must exist.

Formally: for any $w \in L$ w/ $|w| \geq$ # of states in dfa, we get

$w = xy^*z$



Then:

$xz \in L$

$xyz \in L$

$xy^2z \in L$

$xy^iz \in L$

Note:

1. $xy^iz \in L$ for all $i \geq 0$
2. $|y| \geq 1$, $(y \neq \varepsilon)$
3. $|xy| \leq$ # of states in dfa

Proof of pumping lemma for reg-langs:

Let L be a reg-Lang

there exists a pumping constant p for L

for all $w \in L$ w/ $|w| \geq p$ there exists xyz such that $w = xyz$ so that

1. $xy^iz \in L$ for all $i \geq 0$
2. $|y| \geq 1$
3. $|xy| \leq p$

To show a lang is not regular show it breaking 1 of the pumping lemma.

Break PL = not reg.

Demonstrate by breaking an accepted w into all possible pieces not satisfying the PL

example: $\{0^n 1^n : n \geq 0\}$

Assume L is reg:

There exists some p for L where P = no. of states

Choose $w = 0^p 1^p$

$w = xyz = \underbrace{0 \ldots 0}_{p} \underbrace{1 \ldots 1}_{p}$

look at all possible break ups of w into xyz acc. to rules

we know $|xy| \leq p$ $\therefore$ $x = 0^a$

$y = 0^b \quad b \geq 1$

$z =$ rest of string

Try to find a choice of i for xyiz so that we leave the reg. lang

Pick $i = 2$, we have $xyyz$

$y = b \therefore |xyy| = p + b = 0^{p+b} 1^p$

$xyyz$ $0^{p+b}1^p$ can only be in language is if
$p+b = p$ but b cannot be $= 0$
$\therefore 0^{p+b}1^p$ is not regular

Proof that perfect squares are not regular

$Perf S = \{0^{n^2} : n \geq 0\}$

PL: 1. $xy^iz \in L$ for all $i \geq 0$
2. $|y| \geq 1$
3. $|xy| \leq p$

There exists some const. p for L such that
p = no. of states
for all $w \in L$ with $|w| \geq p$
there exists some $xyiz$ that $= w$

$w = \{0^{n^2} : n \geq 0\}$
Choose $w = 0^{p^2}$ if $|w| = p^2$
$|xy| \leq p$
$xy = 0^a0^b : a+b \leq p, b \geq 1$
z = rest of string $0^{p^2-(a+b)}$

Pump up: choose val of $i > 1$

$i = 2, \quad xy^2z = xyyz$

$|xyyz| = 0^a 0^b 0^b 0^z = p^2 + b$

The only way $0^{p^2+b} \in L$ is if $p^2 + b$ is a perfect square

We know that $b \geq 1 \rightarrow p^2 < p^2 + b$

show it can't be a perfect square

If we increment w.s n-val. by 1 we get $p+1$

the next perfect square then is

$(p+1)^2 = p^2 + 2p + 1$

$p^2 < p^2 + b$

$|xy| \leq p$

$\therefore |y| \leq p$

$\therefore b \leq p \rightarrow p^2 + b \leq p^2 + p$

but the next □ is $p^2 + 2p + 1$ which is $> p^2 + b$

$\therefore$ Perf S is not regular

Context free languages

Overview:

Machine Models: Compute whether to acc. a string.

Regex: Formal description of language

Grammars: How to create strings in a language

Grammars all have:

- terminals
- Variables (non-terminals)
- Rules
- Start variables

The purpose of grammars is to generate strings by repeatedly applying rules until a string is made.

We say that Languages are over alphabets and terminals are strings of the language that correspond to a grammar.

Rules: Rules are analogous to transitions

A Grammar example

$A \rightarrow B$	both A, B are some mix of vars and terminals

e.g.

011 → 23	This rule means if we see a '011' in our input we can replace it with 23

What makes a grammar context free?
We have all rules in the following form:

$A \rightarrow X$	A is a variable (single) X is any mix of vars, terminals

As a result once a terminal symbol is generated it cannot be replaced as Rules only apply to vars. Terminals have no equivalents

Example

$S \rightarrow 0S1 \mid \varepsilon$

S = start var
| = OR divider
'0', '1' = terminals

Possible strings:

ε

01

0011

$0 \ldots 0_n 1 \ldots 1_n$

It's easy to see that this grammar is the language $L = \{0^n 1^n : n \geq 0\}$

All regular languages have a cfg but not all cfg.s have a regular lang.

Definitions:

$S \Rightarrow X$: The arrow $\Rightarrow$ means yields, it apply's a rule

Derivation: Following rules to a point of output

Left most derivation

Applying rules to S yields a set of vars

$S \Rightarrow \_ \Rightarrow A \ldots B \ldots C$

$S \Rightarrow \_ \Rightarrow X \ldots B \ldots C$

we replace A w/ X

If A is the furthest fw var then we say we made

a left most derivation.

Every derivation has a left most version
Simply follow the rules the left most var at a time.

Example: balanced paren:

$L = \{(), (())(), \ldots \text{etc}\}$
$L \neq \{)(, (()), \ldots \text{etc.}\}$

Both order and count are important.

Grammar

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

A context free lang is any language for a CFG
All reg langs is a cfl
N.t all cfl's are reg.

Theorem: CFL's are closed under reg ops
Union, Concatenation and star.

Show these ops proving the ops

Let L_1, L_2 be cfgs which have grammars G_1, G_2

$L_3 = L_1 \cup L_2$
$G_3 = G_1 \cup G_2$

G_1 has some start var
S_1
G_2 has some start var
S_2

Let G_3 have start var

$S_3 \rightarrow S_1 | S_2 \quad \therefore G_3 = G_1 \cup G_2$

Concactenation:

$L_3 = L_1 L_2 \quad \therefore G_3 = G_1 G_2$

G_1 has start var
S_1
G_2 has S var
S_2

Let $G_3 = S_3 = S_1 S_2$

Star:

$L' = L_1^*$, L_1 has grammar G_1 with S_1

let $G' = S' \to \varepsilon \mid S_1 S'$

CFL's are closed under reg ops

Useless chains are possible under CFL's but we can mitigate these by reducing excess using the Chomsky normal form of a CFL

$A \to B$
$B \to C$
$C \to A$
} example of recurring chain.

Chomsky Normal Form:

We use a restrictive set of rules

1. $S \to \varepsilon$	Only S can be ε
2. $A \to a$	Vars can be terminals
3. $A \to BC$	Vars can be 2 vars

All Context free grammars have equivalent Chomsky Norm. form.

This is good because all vars in CNF are productive

whereas it's not guaranteed in a non-CNF CFG.

The CYK algorithm can determine the exact no. of rules to apply.

Example

$S \to \varepsilon \mid S,S$

$S_1 \to 1 \mid 0 \mid 1A1 \mid 0B0 \mid \varepsilon \mid A \mid B$

$A \to 1 \mid CAC$

$B \to 0 \mid CBC$

$C \to 0 \mid 0$

↓

1.

$S_0 \to S \mid \varepsilon$ (2)

$S \to \varepsilon$ (2) $\mid S,S \mid S_1 \mid S$

$S_1 \to 1 \mid 0 \mid 1A1 \mid 0B0 \mid \varepsilon$ (2) $\mid A \mid B$

$A \to 1 \mid CAC$

$B \to 0 \mid CBC$

$C \to 0 \mid 0$

Use 5 stage conversion

1. No rhs start var
2. Only $S_0 \to \varepsilon$
3. No unit rules $A \to B$
4. Only 2 vars or 1 terminal

3.

$S_0 \to S \mid \varepsilon$

$S \to S,S \mid S_1$

$S_1 \to 1 \mid 0 \mid 1A1 \mid 0B0 \mid CAC \mid CBC$

$S_0 \to \varepsilon \mid S_1 S_1 \mid 1 \mid 0 \mid cAc \mid cBc$

$A \to cAc \mid 1$

$B \to cBc \mid 0$

$C \to 1 \mid 0$

5.

$S_0 \to \varepsilon \mid S_1 S_1 \mid 1 \mid 0 \mid cAc \mid cBc$

$A \to C_1 A C_1 \mid C_1 A C_2 \mid C_2 A C_2 \mid C_2 A C_1 \mid 1$

$B \to C_1 B C_1 \mid C_1 B C_2 \mid C_2 B C_2 \mid C_2 B C_1 \mid 0$

$C_1 \to 1$

$C_2 \to 0$

Pushdown Automatas:

The machine model of Context free grammars

PDA's are the equivalent of nfa's in design but allow for a stacks and stack operations.
Push on, or pop off stacks