Comp2011
Theory of Computation

a)Admin info

- Topic
  - Automata Theory
  - Computability theory
  - Complexity theory & formal systems

- Further reading
  - Automata & computability ← more excerises
  - Introduction to the theory of computation ← broader and less technical

- Assessment
  - 4 class tests - 30%
  - exam – 70%

b)Theory of computation

- Why?
  - Every idea covered in this module is older than 50 years – solid foundations?
  - Abstract ideas used to create programs & systems in the real world

c)Alphabets & Strings

- An alphabet is any finite set
  - {1,2,3}, {a,b,c}
- A string *s* over an alphabet is a finite sequence of elements
  - 12
  - abc
  - 111222111
- Σ^* is set of all strings from alphabet Σ

d)Strings

- Two strings are equal if they have the same elements in the same order
- Empty string = ε
- Length $\#:\Sigma^* \rightarrow \mathbf{N}$

e)Languages

- Definitions : A language L over an alphabet Σ is some set of string Σ
- Two language are equal when they contain the same string

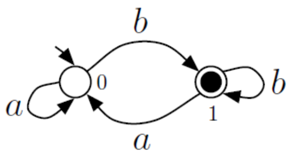
g)Finite Automata

- state – a description of a system at some point in time
- We will use mathematical models of systems with a finite number of states called finite automata
- 3 notations
  - deterministic
  - non-deterministic
  - non-deterministic with ε-moves

h)Deterministic finite state machines

- Expressed in diagrams and with formal definitions

- Change states on actions
- Diagram:



- the above is a picture of a DFA over $\{a, b\}$ with two states, 0 and 1.
- 0 is a start state
- 1 is a final state
- the labelled arrows represent transitions

- Formally:
  - *M =(Q,Σ,δ,s,F)*
    - *Q* is finite set of states
    - *Σ* is the alphabet
    - *δ* is the 'transition' function (.i.e. what decides what happens)
    - *example:* $q \rightarrow^{a} q'$ for $\delta(q,a) = q'$
    - *s* is the start state
    - F (subset) Q is the set of all final states

  - For the diagram above:
    - Q = {0,1}
    - Σ = {a,b}
    - δ(0, a) = 0, δ(0, b) = 1
    - δ(1, a) = 0, δ(1, b) = 1
    - s = 0
    - F = {1}

e)Acceptance and rejection

- An automaton accepts or rejects strings
- Claim: any $x \in \Sigma *$ determines a unique state q
- such that $s \rightarrow^{x} q$. M accepts x when $q \in F$.
  - For machine above: strings abbaabb and bbbb would be accepted but aaabaaa and abaa wouldnt be

- $\delta\text{^} : \Sigma \rightarrow Q$
  - $\delta\text{^}(\varepsilon) = s$,
  - $\delta\text{^}(ab) = \delta(\delta\text{^}(a),b)$

- *String x is accepted by automaton M if* $\delta\text{^}(x) \in F$

a)Deterministic
Means each symbol takes the machine to only one other state, no choices

b)Languages
Sets of strings

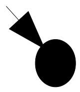
b)Regular languages
Definition:

There exists a machine M such that L = L(M)

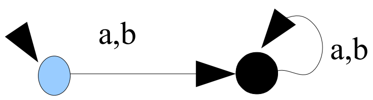
L(M) = {x ϵ Σ | δ^(x) ϵ F}*

In other words there is an automata that can be made which accepts every string in the language then the language is regular

c)Specials
Empty language, no strings to make transitions and start state is a final state



*Σ** (language is all strings): First transition takes machine to final state where it stays



Union:

if L1 and L2 are regular then L1 U L2 is regular

Intersection

if L1 and L2 are regular then L1 ::intersect:: L2 is regular

Concatenation:

Concatenating words from 2 regular languages forms a new regular language

Concatenating words from a language forms a new regular language

d)DFA Product
"Running two DFAs in parallel"

If M1 and M2 are DFAs then their product M3 = M1 X M2

Q3 = Q1 X Q2

s3 = (s1,s2)

F3 = F1 x F3

δ3((q1, q2), σ) = (δ1(q1, σ), δ2(q2, σ))

e)Parallel Languages
If L1 = L(M1) and L2 = L(M2) then L3 = L(M1 X M2) = L1 ::intersect:: L2
.i.e. the set of strings which are always accepted by L3 can be formed from the set of string accepted by L1 which are present in L2

<u>Deterministic v Non deterministic</u>

a)Deterministic v non-deterministic

Deterministic

- next state uniquely determined by current state & input

Nondeterministic

- Next state could be any of a set of states which are accumulated from previous possible transitions

b)Nondeterministic

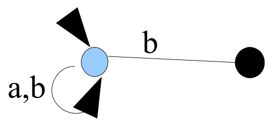
$M = (Q, \Sigma, \Delta, s, F)$

...

$\Delta : Q \times \Sigma \rightarrow P(Q)$ – delta is a function that takes a state and a character and returns a set of states

$q \rightarrow^{a} q'$ for $q' \epsilon \Delta(q,a)$

Essentially if you're on a state q, input a can take you to any of a set of states connected to q by transitions hence the non-determinism



M will accept a string *x* if there is a final state in the set of states which can be reached by making transitions ordered by the string

M accepts x if ::exists:: f . f ϵ F Λ f ϵ $\Delta^{\hat{}}(x)$

$\Delta^{\hat{}}(x)$ is a recursive function that returns the set of states which could be moved to by M using string x. See δ^(x) definition from DFA.

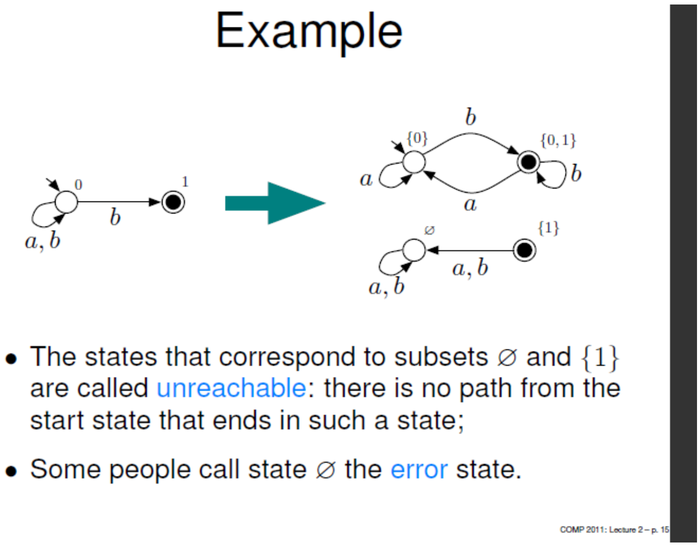
c)So whats more powerful?

DFAs are a subset of NFAs (non-deterministic finite-state automata) where the mapping function is total (i.e. each state has a transition for each letter in the alphabet)

NFAs would be more powerful than DFAs if there was a language that could be accepted by a NFA what wasn't by a DFA. However this isn't the case as a NFA can be modelled as a DFA. Instead NFAs advantage comes in reducing the number of transitions needed to reach a final state (state space?). This becomes apparent in complexity theory.

d)Subset construction
"Making a DFA out of an NFA"

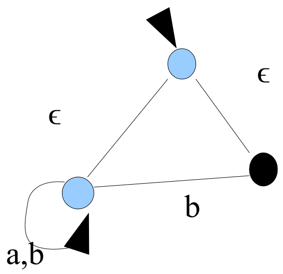
- Set of states is the powerset of states in the NFA
- Start state is {s} of the NFA
- Set of final states is a set containing all subsets of the NFA states which contain a final state of the NFA
  - F' = {X | ::exists:: f ϵ F ^ f ϵ X ^ X ϵ P(Q)}



Example

ϵMoves & Regular Expressions

a)Epsilon Moves
idea: you can always take an ϵ-labelled transition without consuming a symbol to take you to a new state



ϵNFA

b)Language
DFAs, NFAs and ϵNFAs all accept the same regular languages and so are all as powerful as each other.
Instead the ϵNFA and NFA take less computing 'space' to move to a final state and therefore have a complexity advantage in calculations

c)NFA to ϵNFA
- Alter ϵNFA so start state is a final state (so it accepts the string ϵ) and the ϵ transitions are replaced with alphabet transitions.

d)Regular Expressions
A regular expression will either match or not match a string

e)Base cases and operations
Every σϵΣ is a regular expression
L(σ) = { σ}

ϵ is a regex
L(ϵ) = {ϵ}

{} is a regex
L({}) = {}

if a and b are regx then...

L(a + b) = L(a) U L(b)
"strings accepted by a or b is the union of their languages"

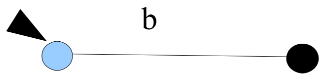
L(ab) = L(a)L(b)
"strings accepted by the concatenation of a and b is the concatenation of the languages accepted by a and the languages accepted by b"
Example: let a = a and b = b, so L(ab) will accept the string "ab" only.
Let a = a and b = a+b so L(a*(a+b)) accepts strings containing all a's or which end in a b*

L(a) = L(a)**
"strings accepted by a is the strings accepted by L(a) concatenated together many times"*
Example: Let a = a, L(a) will accept "a", "a....a" and the empty string*

e)Kleen's theorem
“If a is a regex then L(a) is a regular language”
“If L is a regular language then L=L(a) for some regex a”

This allows regular expressions to be modelled as finite automata

Examples: (modeled as ϵNFAs)



Accepts only
strings ending in b

Languages

a)Regular languages
Can be accepted by an *FA & regexp
Question: Are all languages regular?

b)Non-regular languages
Example: $\{a^n b^n | n \in N\}$ is a non-regular language
No *FA can be constructed for it as it is unbounded & there is no way to store how many a's or b's there are

c)Pumping lemma
See greek symbols on sheet!

Effectively if a part of a string can be repeated an infinite number of times and is still a member of the language then its regular

Pushdown automata – Automata with a stack!!

a)FA's
Can be thought of as a machine that scans a string & changes its internal state
Accepts a string if it ends its transitions in a final state

b)Pushdown automata
Automata with a stack
Works like memory
Stack has pre-set sizelimit
Allows for counting etc...

c)Formally
PDA is a 7-tuple:

$$M=(Q,\Sigma,T,\delta,s,I,F)$$

T is the stack alphabet
I is the initial stack symbol

Non-deterministic!! This gives it more power unlike an FA

$$\delta\subset((Q\,x\,(\Sigma\cup\epsilon\,x\,T))\,x\,(Q\,x\,T))$$

- allows epsilon moves
- Can push an empty string onto the stack

d)Navigation
Each move between states requires a certain symbol in the input string and a symbol on the top of the stack
You then pop that symbol off the stack and add symbols back on (this can be the empty string too)

Moves from state 0 to 1 when there is an A on the input string and a b on the stack, popping the b and replacing it with b

e)Configuration
"A complete description of a computation at a point in time"
- Current state
- Part of input still to be read
- Contents of the stack

$(q, sigmas, yg)\Rightarrow(q', s, hg)$ when $((q,\sigma,y),(q'h))\in\delta$

Note this is non-deterministic!

f)Acceptance

Go through configuration until state is final and input string has been consumed
OR
If input string has been consumed AND stack is empty!!

$(s, x, I) \Rightarrow (q, \epsilon, \epsilon)$

g)Languages accepted by a PA

Regular languages

- epsilon NFA's can be thought of as a PA that doesn't use its stack

Palindromes (aba, aaabaaa etc...)
Balanced parentheses
Context-free languages!