Theory of Computation

Nondeterminism

Lecture 8a - Manuel Mazzara

Nondeterministic models (1)

- Usually one thinks of an algorithm as a **determined sequence of operations**
  - In a certain **state** with a certain **input** there is no doubt on the next step
- Example: let us compare

```
if x>y then max:=x else max:=y
```

with

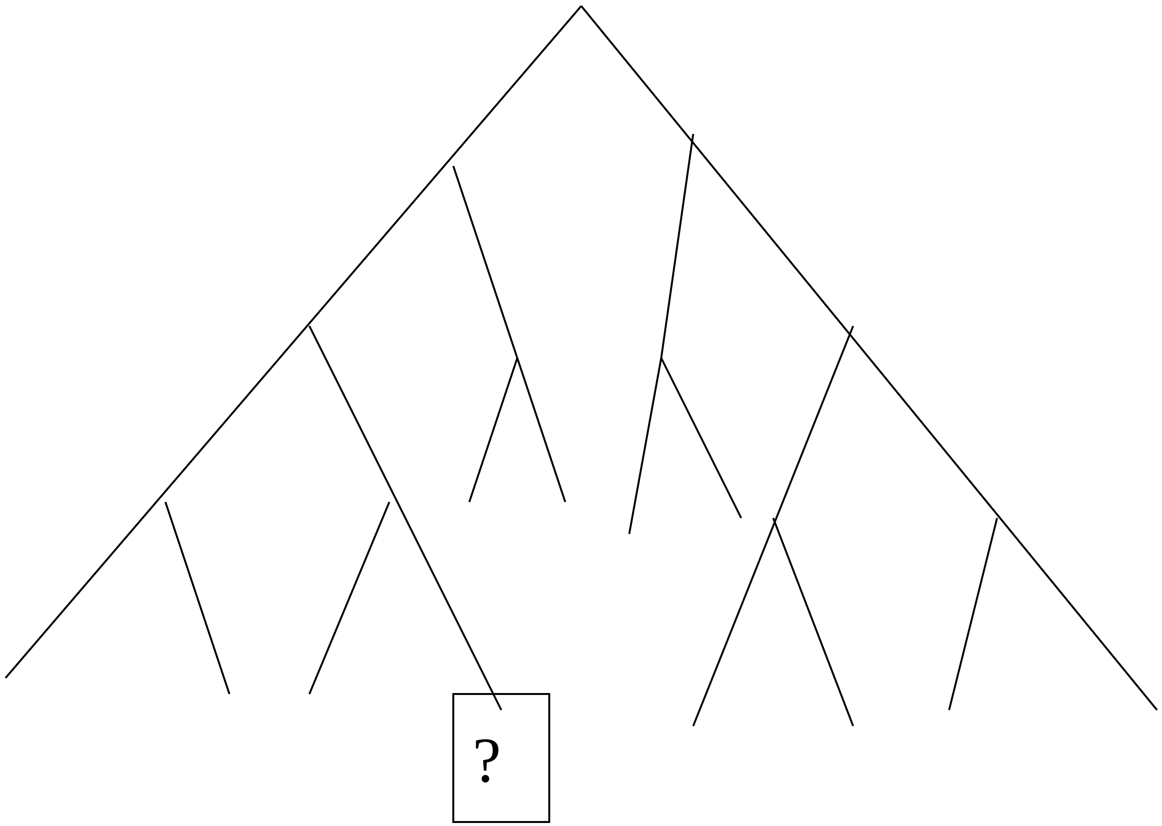
```
if
        x>=y then max:=x
        x<=y then max:=y
fi
```

Nondeterministic models (1)

- Is it only a matter of elegance?
- Let us consider the **`case`** construct of Pascal: why not having something like the following?

```
case
      x=y     then S1
      z>y+3   then S2
      …       then
endcase
```

Blind search



Another form of nondeterminsm that is usually “hidden”

Search algorithms

- Search algorithms are a “simulation” of basically nondeterministic algorithms
  - Is the element searched for in the root?

    If yes, ok

    Otherwise

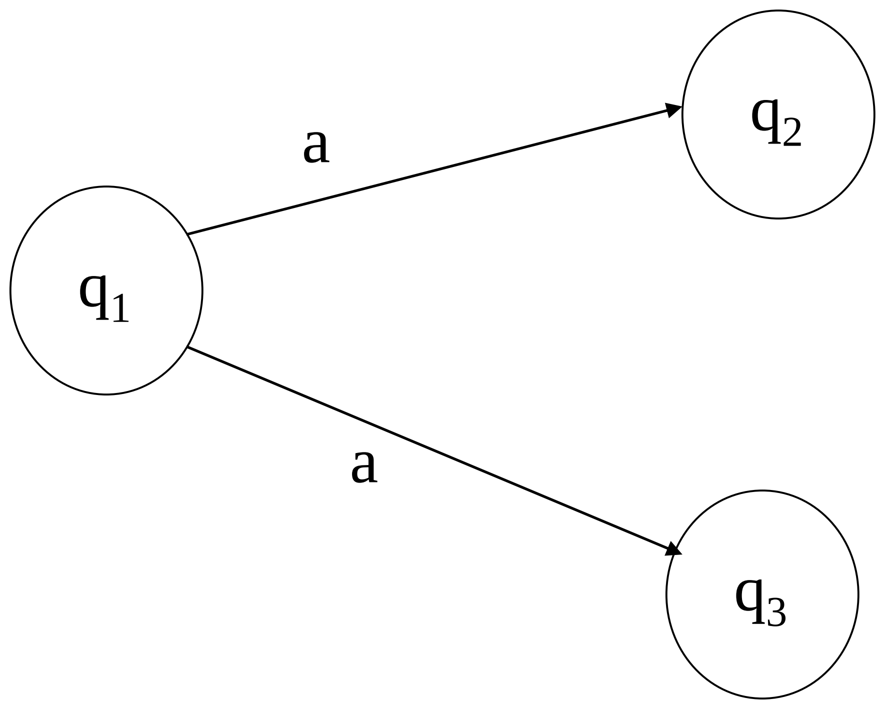
    Search the left subtree

    Search the right subtree
- Choice of priority among paths is often arbitrary

In conclusion

- Nondeterminism (ND) is a model of computation or at least a model of **parallel computing**
  - Ada and other concurrent languages exploit it
- It is a useful abstraction to describe search problems and algorithms
- It can be applied to various computational models
- Important: ND models must not be confused with stochastic models

Adding nondeterminsm



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

Nondeterministic FSA

- A nondeterministic FSA (NDFSA) is a tuple

$\langle Q, I, \delta, q_0, F \rangle$, where
  - Q, I, q_0, F are defined as in (D)FSAs
  - $\delta: Q \times I \rightarrow \mathcal{P}(Q)$
- What happens to δ^*?

Move sequence

- δ^* is inductively defined from δ

$$\delta^*(q, \varepsilon) = \{q\}$$

$$\delta^*(q, y.i) = \bigcup_{q' \in \delta^*(q,y)} \delta(q', i)$$

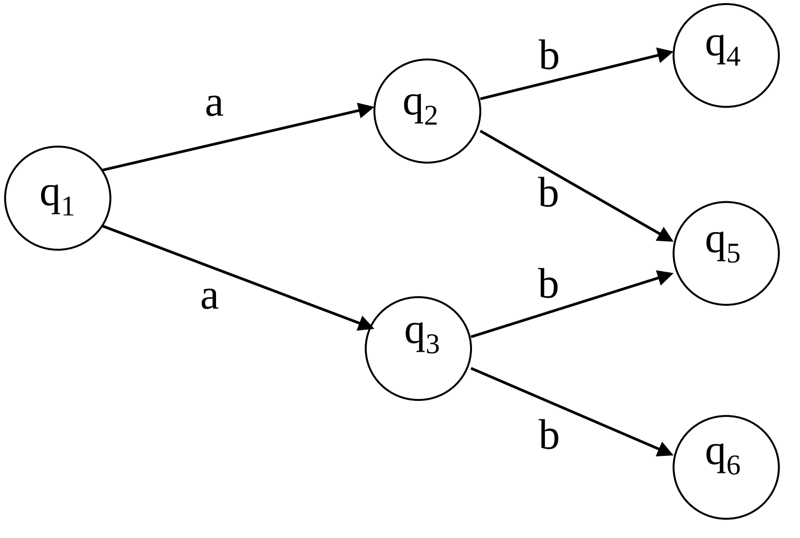
- Example:

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

  $\delta(q_2,b) = \{q_4, q_5\}$,

  $\delta(q_3,b) = \{q_6, q_5\}$

$\rightarrow \delta^*(q_1,ab) = \{q_4, q_5, q_6\}$



Acceptance condition

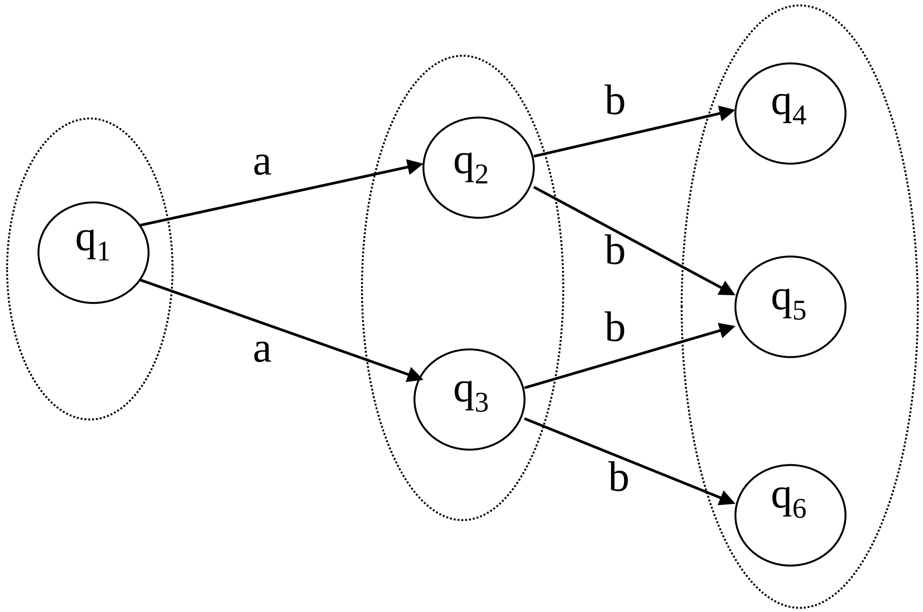
$$x \in L \Leftrightarrow \delta^*(q_0,x) \cap F \neq \emptyset$$

Among the various possible runs (with the same input) of the NDFSA, it is sufficient that **one of them succeeds** to accept the input string

$\rightarrow$ Existential nondeterminism

- There exists also a universal interpretation: $\delta^*(q_0,x) \subseteq F$

DFSA vs NDFSA



- Starting from q_1 and reading *ab* the automaton reaches a state that belongs to the set $\{q_4, q_5, q_6\}$
- Let us call again “state” the set of possible states in which the NDFSA can be during the run

Formally

- **NDFSA have the same power as DFSA**
- Given a NDFSA, an equivalent DFSA can be <u>automatically</u> computed as follows:

  If A_{ND} = <Q, I, δ, q_0, F> then

  A_D = <Q_D, I, δ_D, q_{0D}, F_D>, where

  – $Q_D = \mathcal{P}(Q)$

  – $\delta_D(q_D, i) = \bigcup_{q \in q_D} \delta(q, i)$

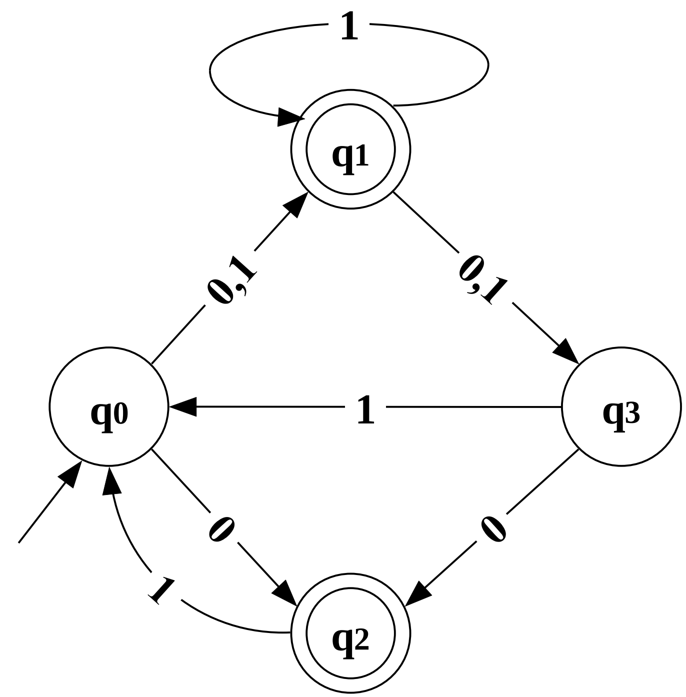
  – $q_{0D} = \{q_0\}$

  – $F_D = \{q_D \mid q_D \in Q_D \wedge q_D \cap F \neq \emptyset\}$

Example (1)

Transform the following NDFSA into the equivalent DFSA

q0	0	q1
q0	0	q2
q0	1	q1
q1	0	q3
q1	1	q3
q1	1	q1
q2	0	⊥
q2	1	q0
q3	0	q2
q3	1	q0



Example (2)

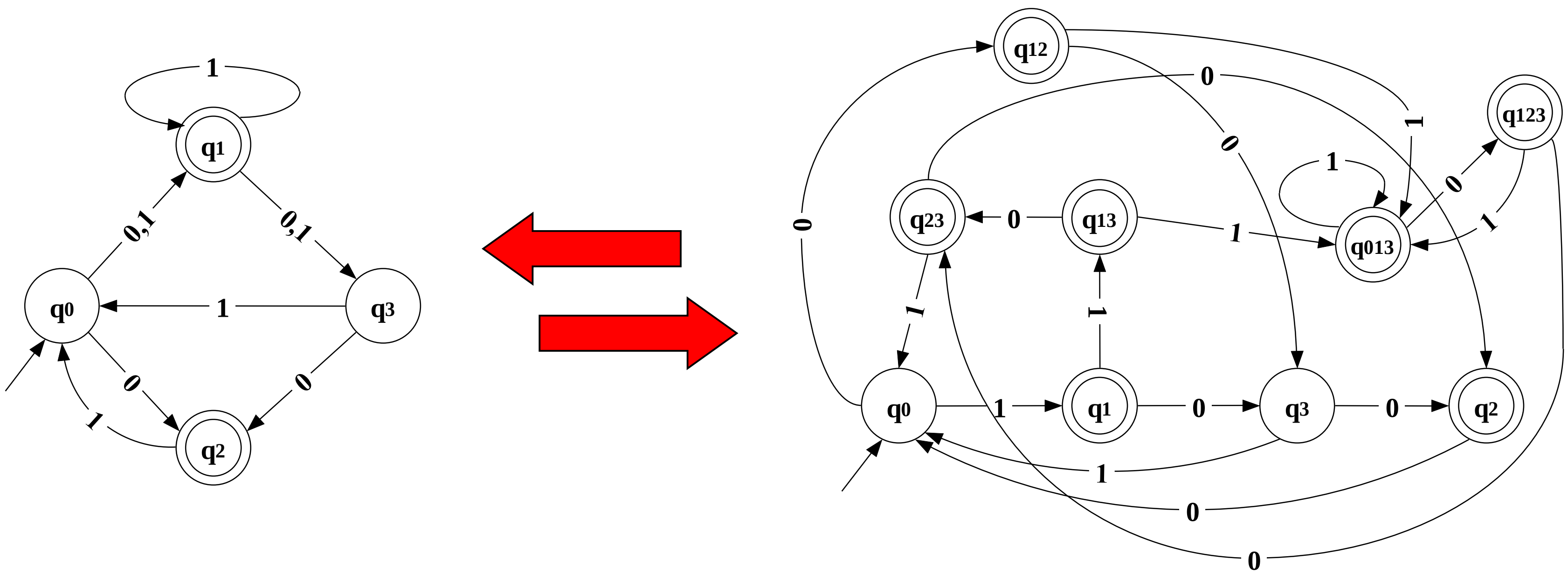
- Let us proceed recursively

q0	0	q1
q0	0	q2
q0	1	q1
q1	0	q3
q1	1	q3
q1	1	q1
q2	0	⊥
q2	1	q0
q3	0	q2
q3	1	q0

q0	0	q12
q0	1	q1
q1	0	q3
q1	1	q13
q2	0	⊥
q2	1	q0
q3	0	q2
q3	1	q0
q12	0	q3
q12	1	q013
q13	0	q23
q13	1	q013
q013	0	q123
q013	1	q013
q23	0	q2
q23	1	q0
q123	0	q23
q123	1	q013

Example (3)

Graphically



Why ND?

- NDFSAs are no more powerful than FSAs, but they are not useless
  - **It can be easier to design a NDFSA**
  - They can be exponentially smaller w.r.t. the number of states
- Example: a NDFSA with 5 states becomes in the worst case a FSA with 2^5 states

Nondeterministic TM

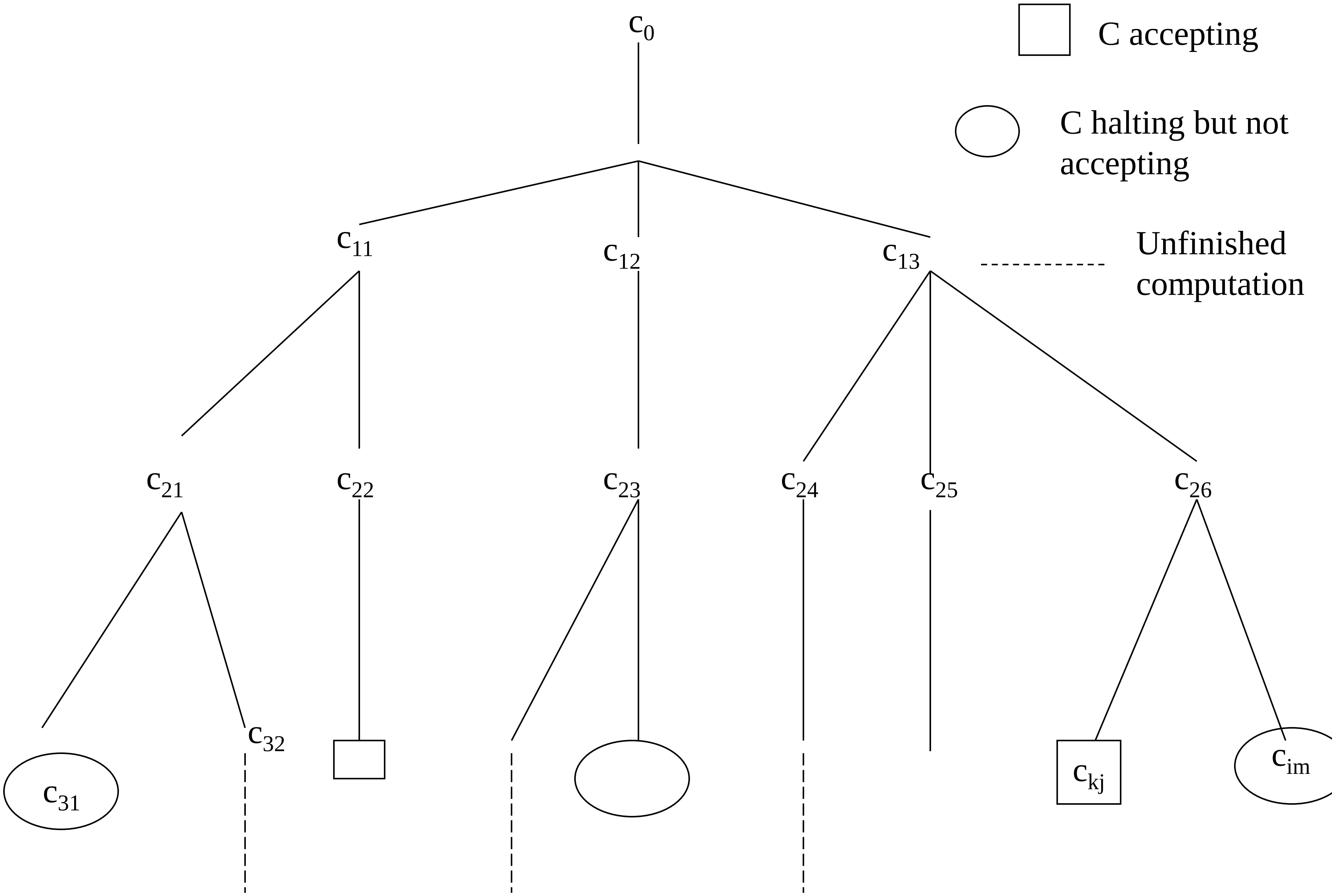
- To define a nondeterministic TM (NDTM), we need to change the **transition function** and the **translation mapping**
- All the other elements remain as in a (D)TM
- The transition function is

$$\delta: (Q\text{-}F) \times I \times \Gamma^k \to \mathcal{P}(Q \times \Gamma^k \times \{R,L,S\}^{k+1})$$

and the output mapping

$$\eta: (Q\text{-}F) \times I \times \Gamma^k \to \mathcal{P}(O \times \{R,S\})$$

Computation tree



Acceptance condition

- A string $x \in I^*$ is accepted by a NDTM if and only if <u>there exists a computation that terminates in an accepting state</u>
- It would seem that the problem of accepting a string can be reduced to a visit of a computation tree
  - How should we perform the visit?
  - What about the relationship between DTMs and NDTMs?

Visiting the computation tree

- We know different kinds of visits:
  - Depth-first visit
  - Breadth-first visit
- A depth-first visit cannot work in our problem because **the computation tree could have infinite paths** and the algorithm might “get stuck” in it
- We should adopt a breadth-first visit algorithm

DTM vs NDTM

Can we build a DTM that visits a tree level by level?

It is a long (and boring) exercise, but YES

$\rightarrow$We can build a DTM that establishes whether a NDTM recognizes a string x

$\rightarrow$Given a NDTM, we can build an equivalent DTM

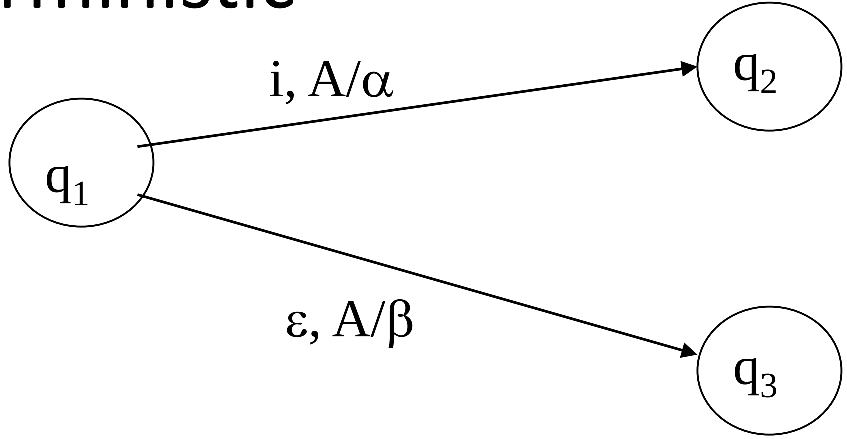
ND does not add power to TMs

ε Moves and PDAs

- ε-moves came with the following constraint:

  If $\delta(q,\varepsilon,A)\neq\perp$, then $\delta(q,i,A)=\perp \ \ \forall i\in I$

- Without this constraint the presence of ε-moves would make PDAs intrinsically nondeterministic



Adding nondeterminism to PDAs

- **Removing the constraint already makes the PDA nondeterministic**
- Additionally, we can have nondeterminism by changing the transition function of a PDA and consequently:
  – transitions among configurations
  – acceptance condition

Definition

A nondeterministic PDA (NDPDA) is a tuple

$<Q, I, \Gamma, \delta, q_0, Z_0, F>$

- $Q, I, \Gamma, q_0, Z_0, F$ as in (D)PDA
- δ is the transition function defined as

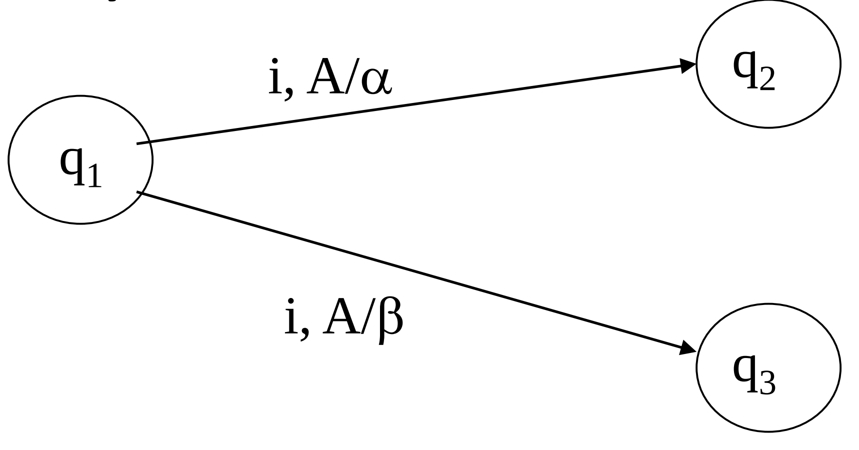
$$\delta: Q \times (I \cup \{\varepsilon\}) \times \Gamma \rightarrow \mathcal{P}_{\mathcal{F}}(Q \times \Gamma^*)$$

- What is the $\mathcal{F}$ in $\mathcal{P}_{\mathcal{F}}$?
- Why $\mathcal{F}$?

Transition function

$$\delta: Q\times(I\cup\{\varepsilon\})\times\Gamma\rightarrow\mathcal{P}_{\mathcal{F}}(Q\times\Gamma^{*})$$

- $\mathcal{P}_{\mathcal{F}}$ indicates the *finite* subsets of $Q\times\Gamma^{*}$
  - Why did we not specify it for NDTM?
- Graphically:

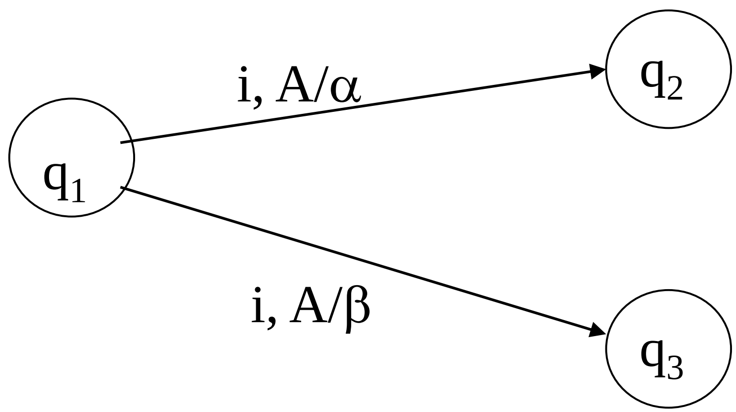


Effects of nondeterminism

- ND **does not add expressive power** to
  - TMs
  - FSAs
- Does ND add expressive power to DPDAs?

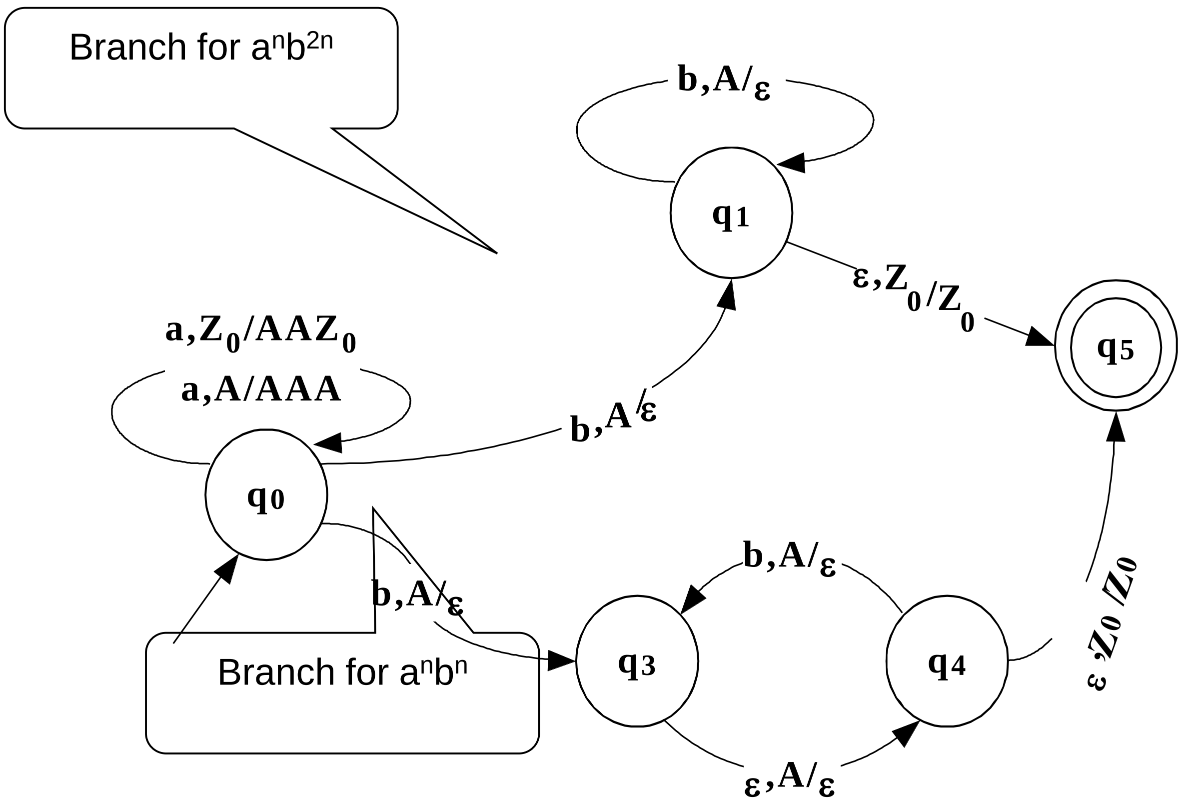
NDPDAs vs DPDAs (1)

- Obviously **a NDPDA can recognize all the languages recognizable by DPDAs**
- ND allows

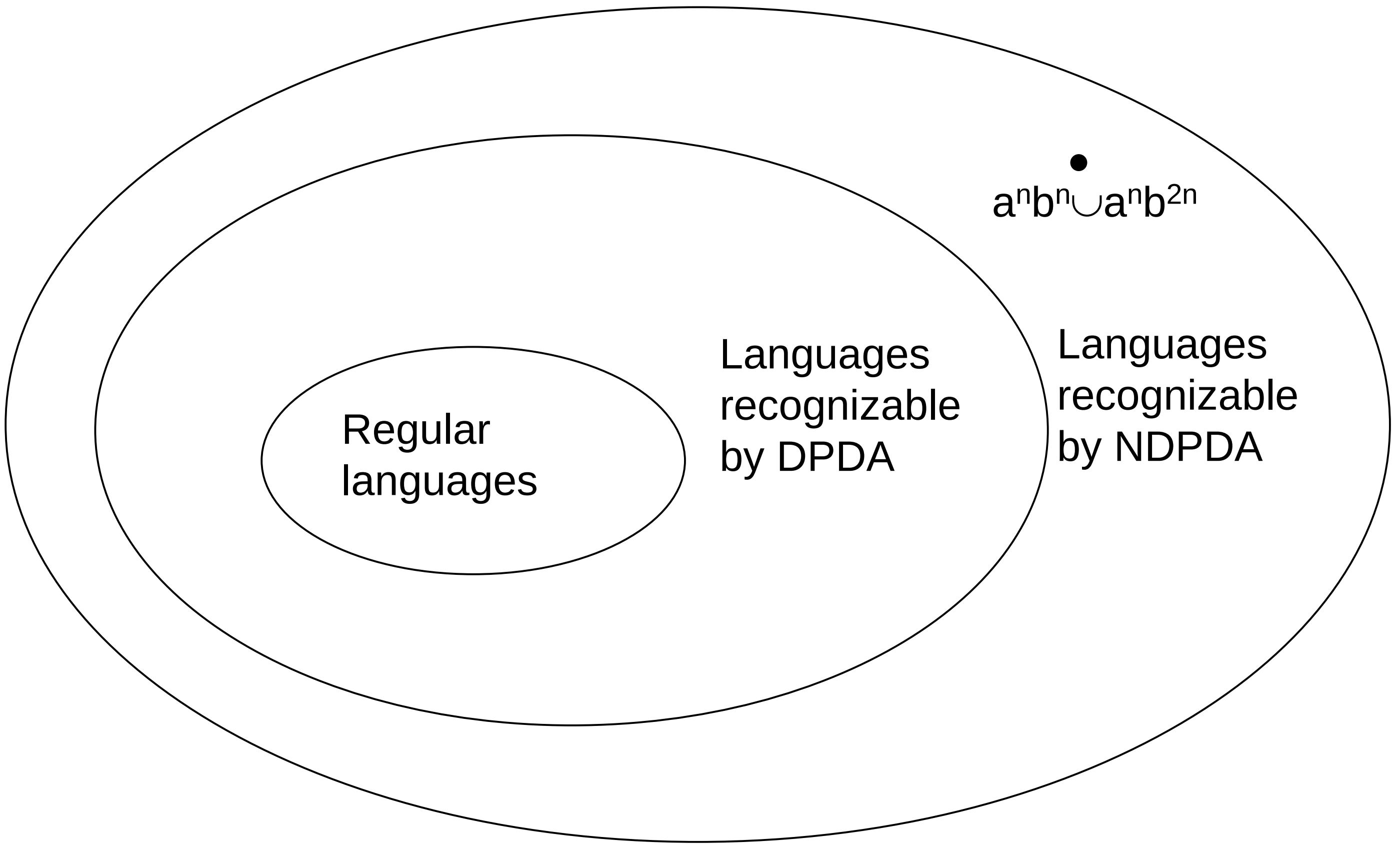


so NPDAs can recognize $\{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$

$\{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$

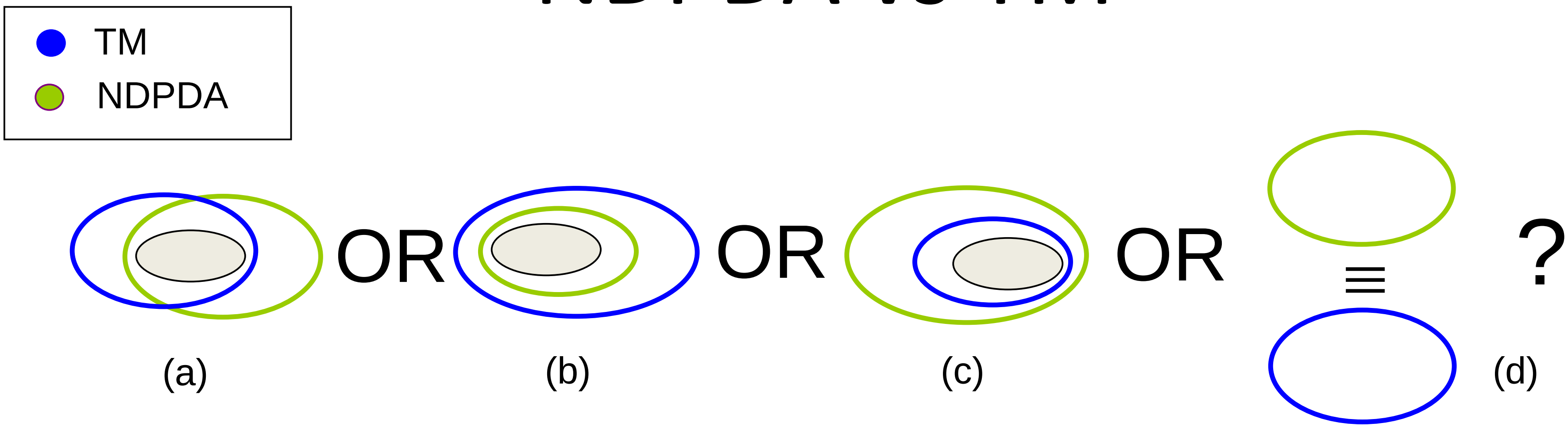


NDPDAs vs DPDAs (2)



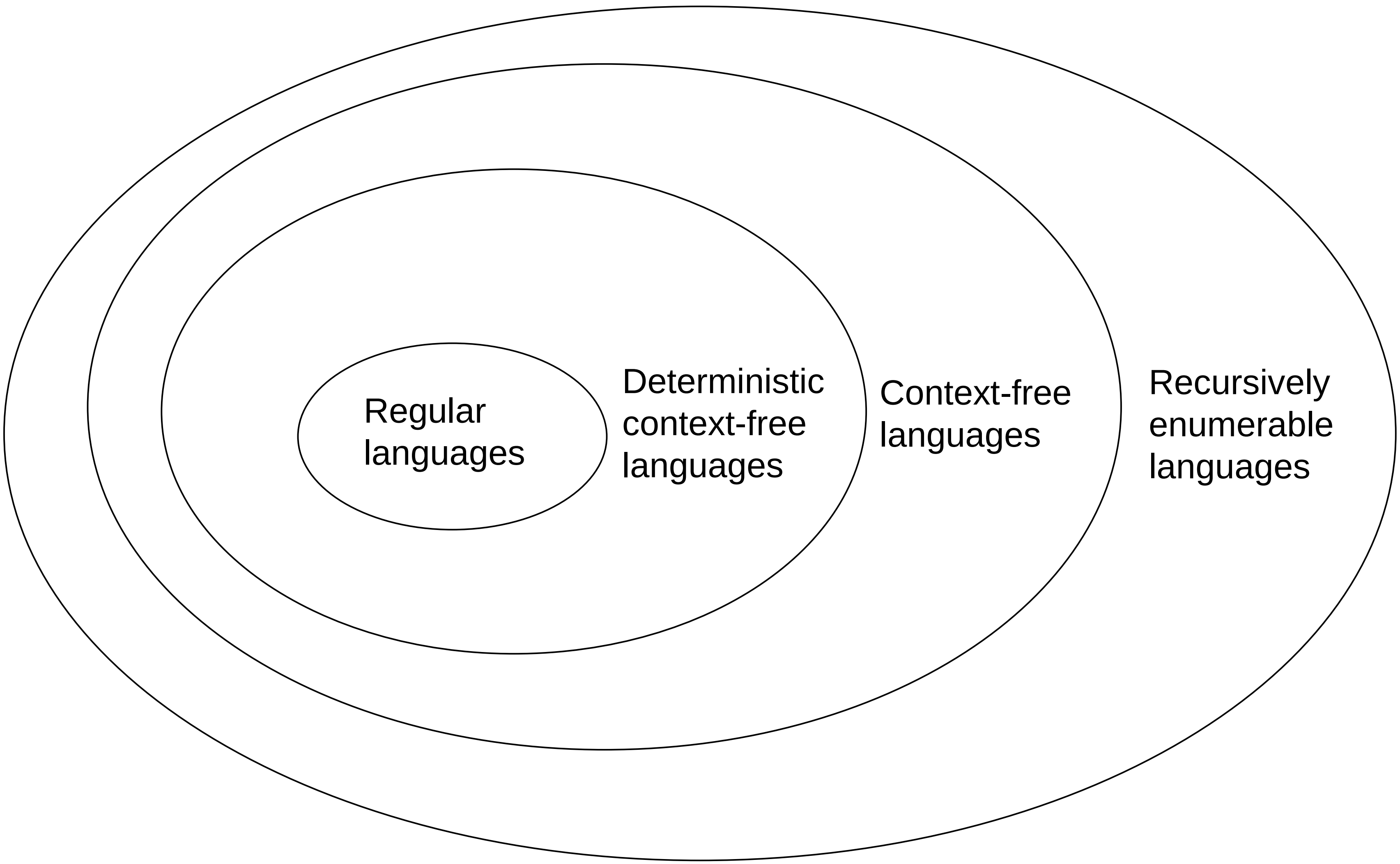
Languages recognizable by NDPDAs are called <u>context-free</u> languages

NDPDA vs TM



- (a) and (c): NO!
  - A (N)DTM can simulate a NDPDA by using the tape as a stack
- (d): NO!
  - The stack is still a destructive memory
  - $\{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$ is recognizable by both

The bull's-eye

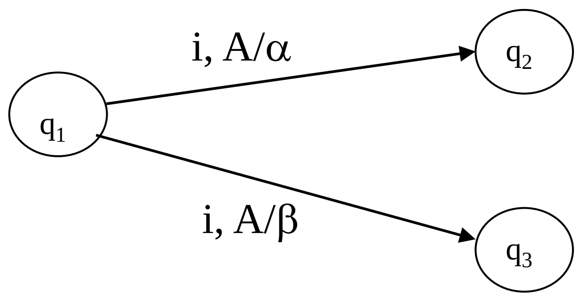


Closure properties in DPDAs

- In DPDAs we have
  - **Closure w.r.t. complement**
  - Non-closure w.r.t. union, intersection, difference
- Does changing the power of the automata change their behavior w.r.t. operations?
  - It can happen
  - New power, new language

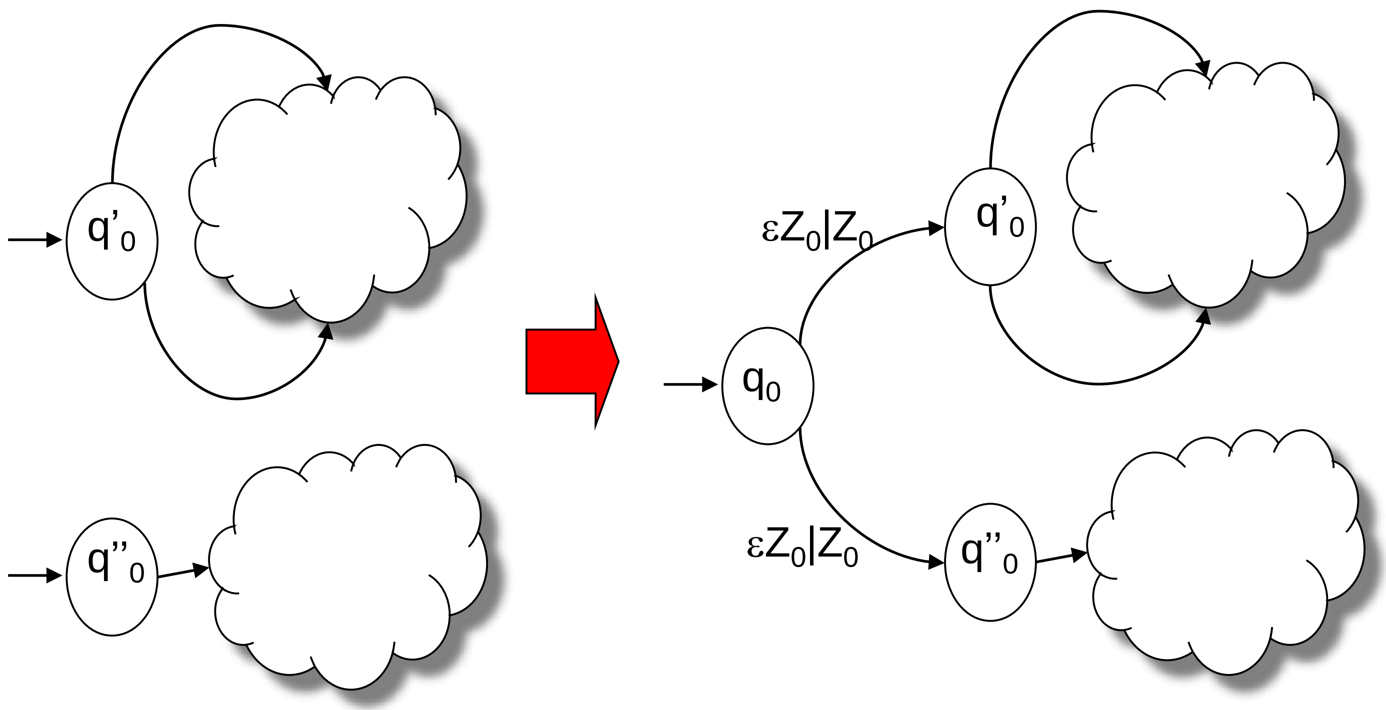
Union (1)

- NDPDAs are closed under union
  - Intuition:



- Given two NDPDAs, P_1 and P_2, we can always build a NDPDA that represents the union by creating a new initial state that is connected to both initial states of P_1 and P_2 with an ε-move

Union (2)



Intersection

- The closure w.r.t. intersection still does <u>not</u> hold
- Consider
  - $\{a^n b^n c^*\}$
  - $\{a^* b^n c^n\}$

  both are recognizable by (N)DPDAs, but

  $\{a^n b^n c^*\} \cap \{a^* b^n c^n\} = \{a^n b^n c^n\}$ is not recognizable by any NDPDA

Complement (1)

- If a class of languages is closed w.r.t. union, but not w.r.t. intersection it cannot be closed w.r.t. complement
  - **We can write intersection in terms of union and complement**
- NDPDAs are <u>not closed</u> w.r.t complement

Remarks

- If a machine is deterministic and its computation terminates, the complement can be obtained by
  - Completing the machine
  - Swapping accepting and non accepting states
- **Nondeterminism or infinite computation does not allow the application of this approach**

Complement (2)

- For NDPDAs, computations can always be made terminating (as for DPDAs)
- However, ND can cause this problem:

  One can have two computations:

  – $c_0=\langle q_0,x,Z_0\rangle \mid\!-^{*}\!- c_1=\langle q_1,\varepsilon,\gamma\rangle$

  – $c_0=\langle q_0,x,Z_0\rangle \mid\!-^{*}\!- c_2=\langle q_2,\varepsilon,\gamma\rangle$

  with $q_1 \in F$ and $q_2 \notin F$

$\rightarrow$ even if we swap accepting and non accepting state, x is still accepted

Theory of Computation

A bit of History and Context

Lecture 8b - Manuel Mazzara

Gottfried Wilhelm Leibniz

- *"It is unworthy of excellent men to lose hours like slaves in the labor of calculation which could safely be regulated to anyone else if machines were used."*

- 1646 - 1716

The “decision problem” (1928)

- The problem asks for an algorithm that takes as input a statement of a first-order logic and answers "Yes" or "No" according to whether the statement is *provable from the axioms using the rules of logic*.”

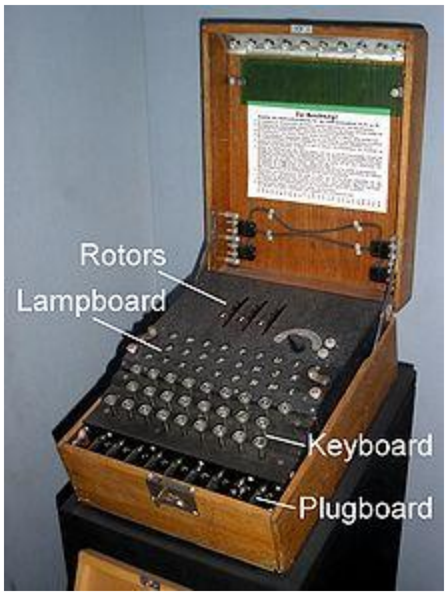
- David Hilbert, 1928

Alan Turing (1)

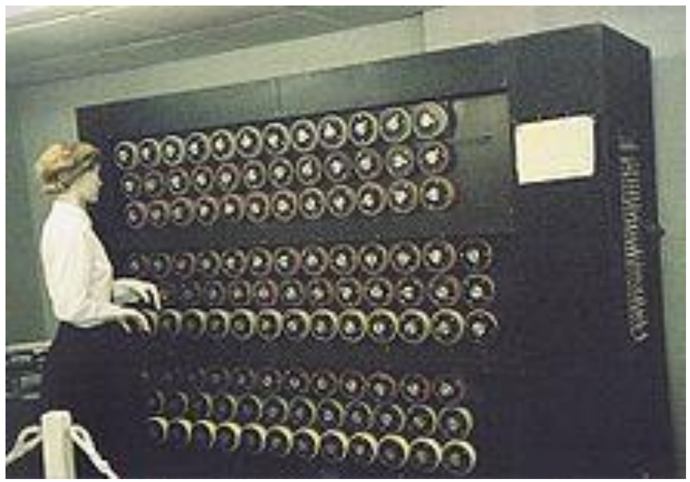
- Independently negatively answered the decision problem
- As we have seen he defined the nowadays Turing Machine – a machine foundation for computing

Alan Turing (2)

- Led to Von Neumann computers and family of imperative programming languages



The Enigma Cryptographic Device



Turing designed the machine to decrypt Enigma messages.

Alonzo Church (1)

- Church's Theorem (1936)
- Independently negatively answered the decision problem

- 1903-1995

Alonzo Church (2)

- Defined the Lambda (λ) Calculus - a language foundation for computing
- Led to family of functional programming languages
- Today the Lambda Calculus serves as a mathematical foundation for the study of functional programming languages.