

# Summary – Theoretical Computer Science V 1.7

## Index

Part 1.....	4
Models in Computer Science.....	4
Meta model: Language.....	4
Translation.....	5
Operational Models.....	5
Finite State Automata (FSA).....	5
FSA as language recognizers & translators.....	6
FSA language recognizer.....	6
FSA language translator.....	6
FSA problems.....	6
Cycles.....	6
Memory.....	7
Operations.....	7
Closure property.....	7
Intersection.....	7
Union.....	8
Complement.....	8
Push-down automaton.....	8
Formalization.....	8
Example.....	9
Properties.....	10
Turing Machines.....	10
K-Tape Turing Machine.....	10
Formalization.....	10
Example.....	11
Properties.....	11
Relations with other computing models.....	12
Non-deterministic (operational) models.....	12
ND Finite Automaton.....	12
NPD Finite Automaton.....	12
Grammars.....	12
Formal definition.....	13
Derivation.....	13
Usage.....	13
Grammar classes.....	13
Grammars in comparison with other models.....	14
Grammars vs Automata.....	14
Grammars vs Turing Machine.....	14
Build G that generates L(M).....	15

Mathematical Logic as Descriptive Formalism.....	15
Language Definition.....	15
Monadic Logic.....	15
Monadic First Order Logic.....	16
String interpretation.....	16
Properties.....	17
FA $\rightarrow$ MFO.....	17
Monadic Second Order Logic.....	17
Buchi theorem.....	17
Part 2.....	18
Theory of Computation.....	18
Language recognition vs function computation problems.....	18
Church Thesis.....	19
Are there problems unsolvable by TM?.....	19
1 fact – TM are enumerable.....	19
Example: TM enumeration.....	19
Function types.....	20
2 fact – UTM Universal Turing Machine.....	20
Which problems can be solved algorithmically?.....	20
Problem of termination: “Halting problem for TM”.....	21
Problem of termination: “Undecidable problem”.....	22
Solvable problem might not be actually solvable.....	22
Decidability and semidecidability.....	23
R (Recursive).....	23
RE (Recursive Enumerable).....	23
Theorem.....	23
Other results.....	24
Rice Theorem.....	24
Reduction.....	25
Dovetailing theorem.....	25
Problem.....	25
Type.....	25
Proof.....	26
Complexity of Computing.....	26
Complexity simplified.....	26
$\Theta$ notation.....	27
Linear speed-up theorems.....	27
Space.....	27
Time.....	28
Implications.....	28
RAM Machine.....	28
TM vs real computers.....	28
RAM Machine schema.....	28
RAM Machine instructions.....	29
RAM Machine problems.....	30
RAM costs.....	31
Complexity measures wrt different computation models.....	31
Time correlation between TM and RAM.....	32
RAM simulates a k-tape TM.....	32

TM simulates a RAM.....	33
Warnings.....	33

This summary hasn't been revised by any professor

If you find any errors please contact me :)

By Flavio Primo

# Part 1

## Models in Computer Science

Different approaches to describe models are available in computer science:

- { • Discrete: like bits and Bytes
- { • Continuous: like real numbers
- { • Operational: based on state and operations to represent evolutions
- { • Descriptive: express properties (desired or undesired)

## Meta model: Language

General way to describe any system.

- **Alphabet**

Finite set of symbolic symbols that can appear in language constructs.

example:  $\{a, b, c, \dots, z\}$  ,  $\{0, 1\}$  , ...

- **String**

Ordered finite sequence of elements of alphabet A.

*properties:*

- string length:  $|a|=1$
- null string:  $\epsilon$  with  $|\epsilon|=0$
- set of all strings of A (including  $\epsilon$ ):  $A^*$

*operations:*

- concatenation:  
non associative, non commutative operation  
given  $x=abb$  ,  $y=baba$  then  $x \cdot y=abbbaba$   
note:  $\forall x | \epsilon \cdot x = x \cdot \epsilon = x$

- **Language**

Language L is a subset of  $A^*$  (  $L \subseteq A^*$  )

*operations:*

- theoretic operations:  $\cup, \cap, L_1 - L_2, \neg L = A^* - L$

- concatenation:  $L_1 \cdot L_2 = \{x \cdot y | x \in L_1, y \in L_2\}$

note:  $\{\epsilon\} \cdot L = L$  ,  $\emptyset \cdot L = \emptyset$

- power:  $L^i = L^{i-1} \cdot L$

- none:  $L^0 = \{\epsilon\}$

- some:  $L^* = \bigcup_{n=0}^{\infty} L^n$

- at least one:  $L^+ = \bigcup_{n=1}^{\infty} L^n$

Practical applications of languages:

- composition of messages over the net
- relation definitions
- filters of messages
- express problem

## Translation

Function that translates a language  $L_1$  into a language  $L_2$  .

$$y = \tau(x)$$

examples:

- $\tau_1(0010110) = 001101110$  doubles the “1”
- $\tau_2(abbbbaa) = baaabb$  switch “a” with “b” and viceversa

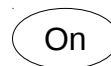
## Operational Models

### Finite State Automata (FSA)

Or Finite State Automata (FSA) or Finite State Machine (FSM) can be defined by:

- finite **state set**:  $Q$

example:  $\{On, Off\}$  and its graphic representation

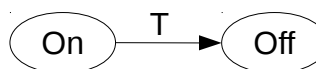


- finite **input alphabet**:  $I$

example:  $T$

- **transition function**:  $\delta: Q \times I \rightarrow Q$

example:  $On \times T \rightarrow Off$



## FSA as language recognizers & translators

### FSA language recognizer

Language recognizer  $R = \langle Q, I, \delta, q_0, F \rangle$  .

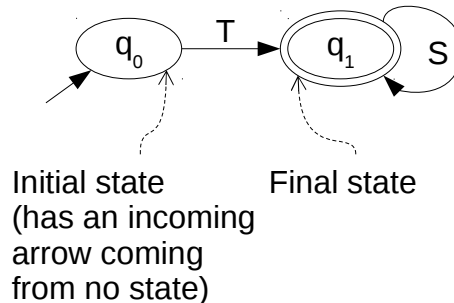
Formalization:

- Move:  $\delta: Q \times I^* \rightarrow Q$

with  $\delta^*$  defined inductively (depends on string length):

- $\delta^*(q, \epsilon) = q$
- $\delta^*(q, y \cdot i) = \delta(\delta^*(q, y), i)$

- Initial state (unique):  $q_0 \in Q$
- Final state (set):  $F \in Q$



### FSA language translator

Translation  $T = \langle Q, I, \delta, q_0, F, O, \eta \rangle$  .

With:

- $O$  output alphabet
- $\eta = Q \times I \rightarrow O^*$  transition function from input to output alphabet

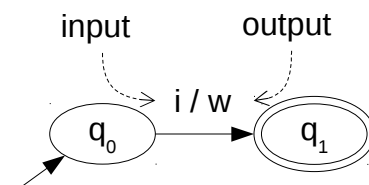
Formalization:

- Move:  $\eta^* = Q \times I^* \rightarrow O^*$

with  $\eta^*$  defined inductively (depends on string length):

- $\eta^*(q, \epsilon) = q$
- $\eta^*(q, y \cdot i) = \eta(q, y) \cdot \eta(\delta^*(q, y), i)$

- Translation:  $\tau(x)[x \in L] = \eta^*(q_0, x)[\delta^*(q_0, x) \in F]$



### FSA problems

#### Cycles

Since an automaton composes a graph, there can be cycles. Cycle is when there exists a sequence path that leads from a state  $q_i$  to the same state  $q_i$  .

**Pumping Lemma:** supports the existences of cycles.

If  $\underbrace{|x|}_{\text{number of inputs}} > \underbrace{|Q|}_{\text{number of states}}$  then  $\exists q \in Q, w \in I^+ | x = ywz \wedge \delta^*(q, w) = q \Rightarrow yw^n z \in L \forall n \geq 0$

useful to get insights for a language:

- $L = \emptyset$  remove all cycles from the automaton, you should still be able to find a path from initial to final state
- $L = \infty$  presence of cycles

## Memory

States are the memory of the automaton.

Automaton cannot count, because it would need an infinite memory → need for a stronger model.

Computers can do so because they operate with small numbers, so they fit in memory.

## Operations

### Closure property

$L$  is closed wrt to  $OP$  (operation) iff  $\forall L_1, L_2 \in L, L_1 OP L_2 \in L$

$R$  is a regular language if:

- accepted by a FSA
- closed wrt set theoretic operations, concatenation, \*, ...

Language mashups:

Given  $A^1 \langle Q^1, I, \delta^1, q_0^1, F^1 \rangle$  ,  $A^2 \langle Q^2, I, \delta^2, q_0^2, F^2 \rangle$

where:

- $Q$  state set
- $I$  input alphabet
- $\delta$  operations
- $q_0$  start state
- $F$  final state

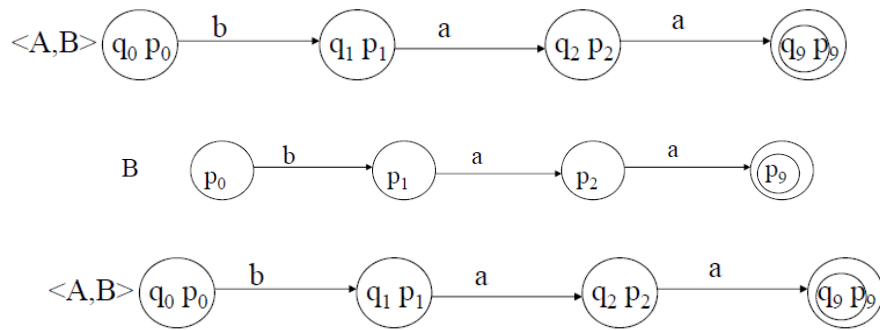
The automaton  $\langle A^1 A^2 \rangle$  defined as:

- $\langle Q^1 \times Q^2, I, \delta, \langle q_0^1, q_0^2 \rangle, F^1 \times F^2 \rangle$
- $\delta(\langle q^1, q^2 \rangle, i) = \langle \delta^1(q^1, i), \delta^2(q^2, i) \rangle$

## Intersection

$$L(\langle A^1, A^2 \rangle) = L(A^1) \cap L(A^2)$$

Simulate parallel run by coupling:



## Union

$$L(\langle A^1, A^2 \rangle) = L(A^1) \cup L(A^2) = \neg(\neg L(A^1) \cap \neg L(A^2)) \quad (\text{need the complement operation})$$

## Complement

Need a FSA for the complement language which consist in: switching final and initial state.

After switching need to check that  $\delta$  is not partial (that all languages elements are supported by the FSA). Is it necessary to add all the connection (also errors one).

In general consider opposite answer to a problem.

## Push-down automaton

Automaton with an added stack memory. The advantage is that now the FSA can count.

note: the stack alphabet is different from either the input and the output alphabet

## Formalization

- Move:
  - depends on: symbol from input tape, symbol at top of the stack, state of control device
  - the automaton:
    1. change self state
    2. moves ahead input scanning head



3. pop: changes symbol A read on top of the stack with a string  $\alpha$  of symbols (even empty)
  4. (**if translator**) writes a string (even empty) on the output tape moving ahead output writing head
- Input: input string recognized (accepted) if
    - automaton scans the string completely
    - at end of scanning string is in acceptance state
  - Translation:
    - if string is not accepted by translator, then the output string is undefined  $\tau(x) = \perp$

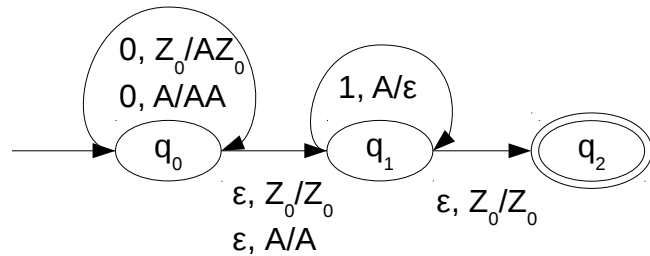
## Example

PDA such that  $\{0^n 1^n | n \geq 0\}$  :

$A\langle Q, I, \Gamma, \delta, q_0, Z_0, F \rangle$

with:

- states:  $Q = \{q_0, q_1, q_2\}$
- input alphabet:  $I = \{0, 1\}$
- stack alphabet:  $\Gamma = \{A, Z_0\}$
- start state:  $q_0$
- start stack symbol:  $Z_0$
- end state:  $F = q_2$



Foreword on the  $\delta$  instructions format:

<current input symbol>, <top of the stack>/<top stack substitution>, (<translation>)

particular cases:

- $\epsilon, \dots / \dots$  spontaneous move, moves without reading the input
- $\dots, A / AA$  push stack operation (adds an “A” in this case)
- $\dots, \dots / \epsilon$  pop stack operation

$\delta$  instructions:

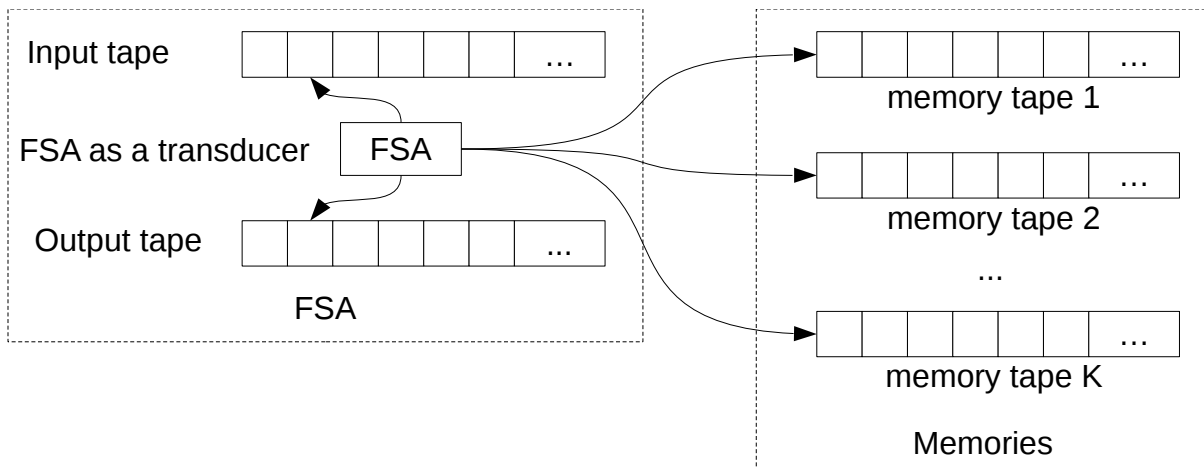
- $0, Z_0 / AZ_0$
- $0, A / AA$  } In  $q_0$  every time a “0” is read, an “A” is pushed in the stack
- $\epsilon, Z_0 / Z_0$
- $\epsilon, A / A$  } Whatever the stack condition is, it can move to state  $q_1$  without reading anything from the input tape
- $1, A / \epsilon$  Every time a “1” is encountered in the input tape it pops an “A” from the stack
- $\epsilon, Z_0 / Z_0$  The end state can be reached only if the stack is empty (with “ $Z_0$ ” symbol) and without reading from the input tape (spontaneous move)

## Properties

- Determinism: a state cannot have at the same time 2 outcomes with one being a “spontaneous move”.
- $x \in L[z = \tau(x)] \Leftrightarrow c_0 = \langle q_0, x, Z_0, [\epsilon] \rangle \vdash^* c_F = \langle q, \epsilon, \gamma, [z] \rangle, q \in F$   
with  $\vdash^*$  as reflexive, transitive closure of the relation  $\vdash$
- function  $\delta$  must be made complete with an error state
- $\epsilon$  moves can cause cycles  $\rightarrow$  not acceptance, but always exists an automaton without loops

## Turing Machines

### K-Tape Turing Machine



- States, alphabets, input and output heads: as the others FSA
- Tapes: infinite cell sequences. At the beginning is empty.  
Symbol “ ” or “\_” is the only symbol that can be contained an infinite number of times in a tape

### Formalization

- Move:
  - Reading:
    - one symbol on the input tape
    - k symbols on the k memory tapes (one for each tape)
    - state of the control device
  - Action:
    - state change  $q \rightarrow q'$

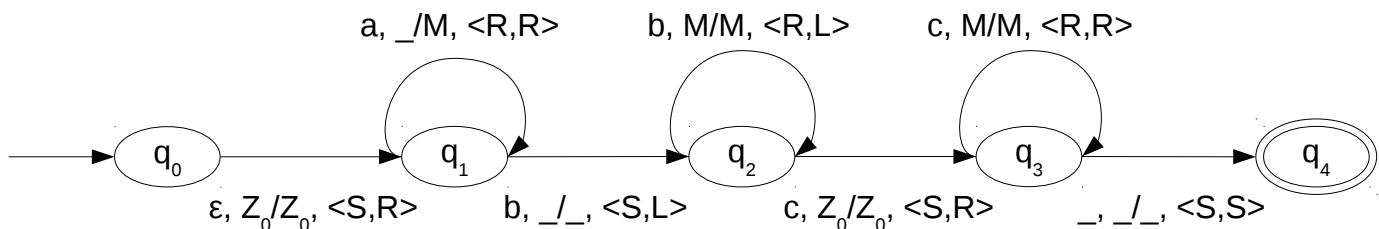
- write a symbol in place of the one read on each of the  $k$  memory tapes:
 
$$A_i \rightarrow A_i', 1 \leq i \leq k$$
- [write symbol of the output tape]
- move of the  $k_{\text{memories}} + 2_{\text{input}} + \text{output heads}$  heads:
  - input head can move: right (R), left (L), still (S)
  - output head can move: right (R), still (S)
- Initial configuration:
  - memory tapes:  $Z_0$  followed by all blanks, head position in 0
  - output: all blanks
  - initial state:  $q_0$
  - input: head position in 0
- Final configuration:
  - input accepted if: ends in accepted state  $\in F$
  - input not accepted if: TM stops in a state  $\notin F$ , TM never stops

### Example

Foreword on the  $\delta$  instructions format:

$\langle \text{current input symbol} \rangle, \langle \text{top of the stack of every } k \text{ tape} \rangle / \langle \text{top stack substitution of every } k \text{ tape} \rangle,$   
 $\langle \text{input move, memory tapes move} \rangle$

TM accepts  $\{a^n b^n c^n | n > 0\}$  : TM with 1 memory tape



### Properties

Closure property:

- intersect: OK
- union: OK
- complement: NO

Equivalent TM models:

- Single tape TM: one tape serves as input, memory and output (both directions)
- Bidimensional Tape TM: tape with  $k$  heads for each tape

## Relations with other computing models

TM can simulate a Von Neumann machine, but memory management is different:

- sequential  $\leftarrow$  TM
- direct  $\leftarrow$  Von Neumann

Solve same problems, but different needs for resources.

## Non-deterministic (operational) models

Not always can be determined the sequence of operations that will be executed:

- if  $x > y$  then  $\max := x$  else  $\max := y$
- blind search in a tree

## ND Finite Automaton

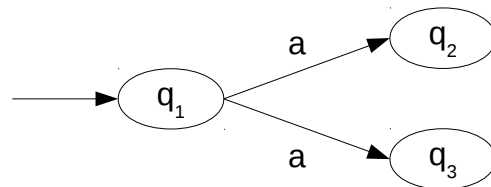
$$\delta: Q \times I \rightarrow P(Q)$$

where:

- $P(\dots)$  is the Power Set (set of sets)

String acceptance:

if exists path to a final state (among various runs with same input)



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

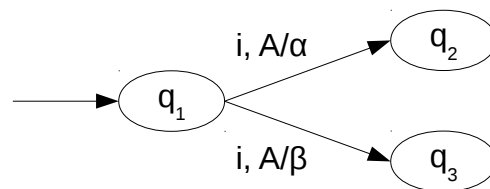
## NPD Finite Automaton

$$\delta: Q \times (I \cup \{\epsilon\}) \times \Gamma \rightarrow P_F(Q \times \Gamma^*)$$

String acceptance:

$$c_0 | -^* - \langle q, \epsilon, \gamma \rangle, q \in F$$

if exists path to a final state (among various runs with same input)



## Grammars

- **Automata:** analyzing model to recognize/accept, translate, compute strings of a language
- **Grammars:** generative model to generate strings of a language.

Set of rules to build phrases of a language.

## Formal definition

$$G = \langle V_N, V_T, P, S \rangle$$

where:

- $V_N$  : non terminal alphabet  
symbols that recall other grammar rules.
  - $V_T$  : terminal alphabet  
symbols that don't recall other grammar rules.
- $V = V_N \cup V_T$

- $P$  : set of rewriting rules

Set of rules that define the string generation for a language. Rules uses symbols from  $V$ .

*recognized \_ pattern*  $\rightarrow$  *pattern \_ replacement*

example:

- $A \rightarrow \alpha$  substitute  $A$  with  $\alpha$
- $A \rightarrow C$  substitute  $A$  with  $C$ . Since  $C$  is a non terminal symbol, there must be a rule like  $C \rightarrow \dots$  that's being called.
- $S$  : initial symbol ("axiom").

Rule that can be applied at the beginning of the string (there may be more than one rule to start a string generation).

## Derivation

Strings that can be generated with a grammar.

- $\Rightarrow^*$  zero or more rewriting steps
- $\Rightarrow$  one rewriting step

Language generated by a grammar:  $L(G) = \{x \mid x \in V_T^* \wedge S \Rightarrow^* x\}$

contains all the strings derived from rule  $S$  and terminates with a terminal symbol.

## Usage

Define syntax of programming language and defines automaton that can recognize and process it.

## Grammar classes

Regular grammar  $\subseteq$  Context-free grammar  $\subseteq$  General grammar

- **Context-free grammars**

$\forall rule(\alpha \rightarrow \beta) \in P, |\alpha| = 1$ , context-free (rewriting of  $\alpha$  does not depend on its context)

- **Regular Grammars**

$\forall rule(\alpha \rightarrow \beta) \in P, |\alpha| = 1, \beta \in (V_T \cdot V_N) \cup V_T \cup \epsilon$ , subset of “Context-free grammars”

## Grammars in comparison with other models

### Grammars vs Automata

Define equivalence between grammars and automata:

- **FA  $\rightarrow$  RG**

Given: Finite Automaton FA

Define:

- $V_N = Q$
- $V_T = I$
- $S = \langle q_0 \rangle$
- $\forall \delta(q, i) = q' \Rightarrow \langle q \rangle \rightarrow i \langle q' \rangle$
- $q' \in F \Rightarrow \langle q \rangle \rightarrow i$
- by induction also:
  - $\delta^*(q, x) = q' \Leftrightarrow \langle q \rangle \Rightarrow^* x \langle q' \rangle$
  - $q' \in F \Rightarrow \langle q \rangle \rightarrow x$

- **RG  $\rightarrow$  FA**

Given: Regular Grammar RG

Define:

- $Q = V_N \cup \{q_F\}$
- $I = V_T$
- $\langle q_0 \rangle = S$
- $F = \{q_F\}$
- $\forall A \rightarrow bC \Rightarrow \delta(A, b) = C$
- $\forall A \rightarrow b \Rightarrow \delta(A, b) = q_F$

In this way obtained FA is ND

### Grammars vs Turing Machine

Given grammar G, build a non-deterministic Turing Machine M accepting L(G).

- *Input tape*: initialized with input string x
- *Memory tape* (unique): initialized with  $Z_0 S$
- *M*: operates in a non-deterministic way until it finds a derivation.

Scans LtR and substitutes RtL.

**Process**  $(\alpha \Rightarrow \beta) \Leftrightarrow c = \langle x, q_s, Z_0 \alpha \rangle \vdash^* \langle x, q_s, Z_0 \beta \rangle$

**Acceptance** if  $y \in V_T^* == x$  then x is accepted

**Problem** if  $x \notin L(G)$  then M might try an infinite number of ways

## Build G that generates L(M)

1. G generates all strings of type  $x \$ X$

with:

- $x \in V_T^*$
- $X \in V_N^*$  copy of x

2. G simulates successive configs of M using string obtained with X

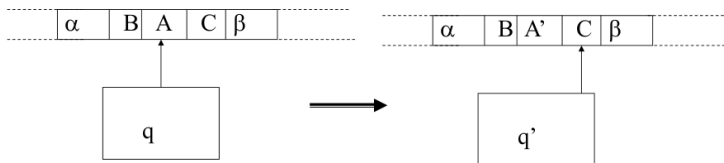
3. G defined as  $(x \$ X \Rightarrow^* x) \Leftrightarrow x$  accepted by M

The idea is to simulate each move of M by an immediate derivation of G:

so we associate every possible move of a TM with a possible outcome:

- $\delta\langle q, A \rangle = \langle q', A', R \rangle \rightarrow q$  state right move
- $\delta\langle q, A \rangle = \langle q', A', S \rangle \rightarrow q$  state still move
- $\delta\langle q, A \rangle = \langle q', A', L \rangle \rightarrow q$  state left move

example:



Right move:

$$x \$ \alpha B q A C \beta \Rightarrow x \$ \alpha B A' q' C \beta$$

## Mathematical Logic as Descriptive Formalism

Logic is a universal formalism that can be adapted to describe languages, properties, timings.

Logic power depends on the use of:

- interpretation domain (universe)
- set of symbols (alphabet)
- variables quantification for
  - 1<sup>st</sup> order elements of the universe
  - 2<sup>nd</sup> order sets/relations of elements

## Language Definition

$$\text{example } L = \{a^n b^n \mid n \geq 1\} \Rightarrow \forall x (x \in L \Leftrightarrow \exists n (n \geq 1 \wedge x = a^n \cdot b^n))$$

## Monadic Logic

Predicates with one argument with, as a successor, a binary predicate.

## Monadic First Order Logic

Given:

- alphabet  $I$
- string  $w \in I^*$  with:  $|w| = n$  ,  $w = w_0 \cdot w_1 \cdot \dots \cdot w_{n-1}$

Operators:

- element position in a string (monadic)  
 $w = acbaa \rightarrow a(0) = true$  ,  $b(1) = false$
- equality (binary)  
 $x = y$
- successor (binary)  
 $S(x, y) \rightarrow$  "y" is the string position that immediately follows "x"
  - successor:  $S(x, y) \Rightarrow y = x + 1$   
successors:  $\exists z_1, \dots, z_{k-1} (z_1 = x + 1 \wedge \dots \wedge y = z_{k-1} + 1) \Rightarrow \forall k > 1 | y = x + k$
  - predecessor:  $S(y, x) \Rightarrow y = x - 1$   
predecessors:  $x = y + k \Rightarrow y = x - k$

Derived operators:

- less than  $x < y$
- less than or equal  $\neg(y, x) \Rightarrow x \leq y$
- 0 constant  $\neg \exists y S(y, x) \Rightarrow x = 0$
- integer constants *successors of*  $0, 1, 2, \dots \Rightarrow 1, 2, 3, \dots$
- first string position  $\neg \exists y S(y, x) \Rightarrow first(x)$
- last string position  $\neg \exists x S(y, x) \Rightarrow last(y)$
- number of same element in a string  $\#_a x$

## String interpretation

Given string  $w = acbaa$

then:

- alphabet (used symbols):  $I = \{a, b, c\}$
- universe (available positions in the string interpreted as symbols):  $U = [0, \dots, n-1] = [0, \dots, 4]$



- successor relations  $S = \{(0,1), (1,2), (2,3), (3,4)\}$
- less than relation  $< = \{(0,1), (0,2), \dots, (1,2), (1,3), \dots, (3,4)\}$

Language defined by sentences (formulas with quantified variables):  $L(\phi) = \{w \mid w \models \phi\}$

## Properties

MFO defines star-free languages, obtained by starting from any finite language by means of a finite number of: union, intersection, complement and concatenation.

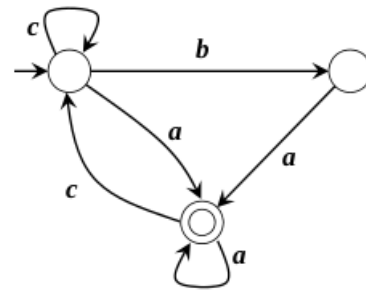
- Closed to set operations:  $\wedge, \vee, \neg$
- Cannot exist language with 2 symbols with one symbol in all even positions on even strings
- FA more powerful than MFO
- Not closed under Kleene Star  $*$

## FA $\rightarrow$ MFO

$$\varphi = \left( \begin{array}{c} \text{no 'a' is followed by a 'b'} \\ \wedge \\ \text{every 'b' is followed by an 'a'} \\ \wedge \\ \text{the string ends with an 'a'} \end{array} \right)$$



$$\varphi = \left( \begin{array}{c} \neg \exists x \exists y (S(x, y) \wedge a(x) \wedge b(y)) \\ \wedge \\ \forall x (b(x) \rightarrow \exists y (S(x, y) \wedge a(y))) \\ \wedge \\ \exists x (last(x) \wedge a(x)) \end{array} \right)$$



## Monadic Second Order Logic

MFO empowered by the use of variables denoting predicates (sets of numbers). Those variables are expressed as uppercase.

$$\begin{aligned} & \forall x (x=0 \Rightarrow \neg E(x)) \wedge \\ & \forall y (y=x+1 \Rightarrow (\neg E(x) \Leftrightarrow E(y))) \wedge \\ & \exists E(\dot{i}(E(x) \Rightarrow a(x)) \wedge \dot{i} \forall y (last(y) \Rightarrow E(y))) \end{aligned}$$

## Buchi theorem

Language (with finite-length words) is recognizable by a finite automaton iff it is definable by a MSO sentence.

## FA → MSO

1. automaton  $A = (Q, I, q_0, \delta, F)$  with  $Q = \{q_0, \dots, q_k\}$  reads string  $w$
2. uses predicates  $X_i = \{\text{positions of } w \text{ where } A \text{ reaches state } q_i\}$
3. formula  $\phi$  equivalent to  $A$

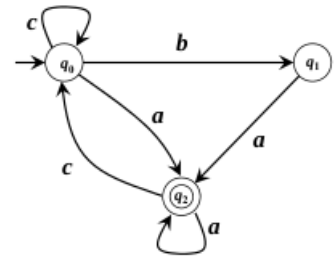
$$\phi = \exists X_0 \dots \exists X_k \left( \begin{array}{l} \wedge_{i \neq j} \forall x \neg (X_i(x) \wedge X_j(x)) \\ \wedge \forall x (\text{first}(x) \rightarrow X_0(x)) \\ \wedge \forall x \forall y \left( S(x, y) \rightarrow \bigvee_{q_j \in \delta(q_i, a)} (X_i(x) \wedge a(x) \wedge X_j(y)) \right) \\ \wedge \forall x \left( \text{last}(x) \rightarrow \bigvee_{\exists q_j \in F: q_j \in \delta(q_i, a)} (X_i(x) \wedge a(x)) \right) \end{array} \right)$$

state positions pairwise disjoint
initial state

final state
transitions

example: li

$$\exists X_0 \exists X_1 \exists X_2 \left( \begin{array}{l} \forall x (\neg (X_0(x) \wedge X_1(x)) \wedge \neg (X_0(x) \wedge X_2(x)) \wedge \neg (X_1(x) \wedge X_2(x))) \\ \wedge \forall x (\text{first}(x) \rightarrow X_0(x)) \\ \wedge \forall x \forall y \left( S(x, y) \rightarrow \left( \begin{array}{l} X_0(x) \wedge c(x) \wedge X_0(y) \vee X_0(x) \wedge b(x) \wedge X_1(y) \\ \vee X_0(x) \wedge a(x) \wedge X_2(y) \vee X_1(x) \wedge a(x) \wedge X_2(y) \\ \vee X_2(x) \wedge a(x) \wedge X_2(y) \vee X_2(x) \wedge c(x) \wedge X_0(y) \end{array} \right) \right) \\ \wedge \forall x (\text{last}(x) \rightarrow (X_0(x) \wedge a(x) \vee X_1(x) \wedge a(x) \vee X_2(x) \wedge a(x))) \end{array} \right)$$



Where:  $X_0$  is the set of the initial states.

[...]

## Part 2

### Theory of Computation

Languages let formalize any “computer science problem” to understand whether it is solvable or not.

### Language recognition vs function computation problems

Given:

- $x \in L$  language recognition problem
- $y = \tau(x)$  function computation problem

I can reduce both formulations to:

- I have a machine that can solve  $y = \tau(x)$  and I want to use it to solve  $x \in L$

$$\rightarrow \text{define } \begin{cases} \tau(x) = 1 & \text{if } x \in L \\ \tau(x) = 0 & \text{if } x \notin L \end{cases}$$

- I have a machine that can solve  $x \in L$  and I want to use it to solve  $y = \tau(x)$

$$\rightarrow \text{define } L_\tau = \{x \$ y \mid y = \tau(x)\}$$

Then:

For a fixed  $x$  I can enumerate all possible strings  $y$  over the output alphabet and for each of them ask the machine if  $x \$ y \in L_\tau$

After some attempts, if  $\tau(x)$  is defined  $\rightarrow$  found string  $y$  for which the machine answers positively.

## Church Thesis

TM is the most powerful computational device.

No algorithm can solve problems that TM can't.

## Are there problems unsolvable by TM?

### 1 fact – TM are enumerable

$$E: S \leftrightarrow N$$

- E enumeration, matches uniquely an element from  $S$  to an element of  $N$

- $S$  set to enumerate
- $N$  enumeration set

$$\begin{array}{cccccccccccc} \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \dots\} \\ \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \\ \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \dots\} \end{array}$$

### Example: TM enumeration

Given:

- fix unique alphabet  $A$  ( $|A|=2, A=\{0,1\}$ )
- single tape TM

Enumerate 2-states TM:

	0	1
$q_0$	$\perp$	$\perp$
$q_1$	$\perp$	$\perp$

TM<sub>0</sub>

	0	1
$q_0$	$\perp$	$\perp$
$q_1$	$\perp$	$\langle q_0, 0, S \rangle$

TM<sub>1</sub>

.....

How many 2-states TM do exist? → How many functions  $f: D \rightarrow R$  do exist?

$|R|^{|D|} = (2 \cdot 2 \cdot 3 + 1)^{(2 \cdot 2)}$  for each x we have a  $|R|$  choices:

- $|R| = |Q| \cdot |A| \cdot \# \text{ input} + 1$
- $|D| = |Q| \cdot |A|$

obtain enumeration:  $E: \{TM\} \leftrightarrow N$

$E(M)$  is the “Goedel Number” of M (tells the position assumed in the enumerable set), where:

- E is a “goedelization”
- M current TM

$f_y(x) = \perp \Leftrightarrow M_y$  does not stop when takes x as input

### Function types

- *total (partial)*: defined for every (undefined for some) value of it's domain
- *computable (not computable)*: there's a TM that computes

	Total	Partial
Computable	$f(x) = x + 1$	$f(x) = \perp$ for all x
Not computable	$f(x) = \begin{cases} 1 & \text{if } f_x(y) = 2 \cdot y \\ 0 & \text{otherwise} \end{cases}$	$f(x) = \begin{cases} 1 & \text{if } x \text{ is even} \wedge f_{x/2} \text{ is total} \\ 0 & \text{if } x \text{ is even} \wedge f_{x/2} \text{ is partial} \end{cases}$

### 2 fact – UTM Universal Turing Machine

There exist a universal TM (UTM) that computes function  $g(x, y) = f_y(x)$  (still enumerable  $N \times N \leftrightarrow N$ ). Where:

- y program
- x input to program

So it can be defined a function  $g^{\wedge}(n) = g(d^{-1}(n)) = g(x, y)$ , where  $n = d(y, x)$  is the enumeration that corresponds to a unique TM. So the UTM is given n and it computes  $d^{-1}(n)$ .

### Which problems can be solved algorithmically?

How many functions?

$\{f: N \rightarrow \{0, 1\}\} \subseteq \{f: N \rightarrow N\} \Rightarrow |\{f: N \rightarrow \{0, 1\}\}| \leq |\{f: N \rightarrow N\}| = |\wp(N)| = 2^{\aleph_0}$  ( $\aleph_0$  is an type of infinite)

How many computable functions, which is a denumerable set?

$\{f_y: N \rightarrow N\} = \aleph_0 < 2^{\aleph_0}$

Less computable functions than total set of functions, so not all functions can be solved algorithmically.

## Problem of termination: “Halting problem for TM”

### Question

A program can not terminate (not solving the problem), can it be determined in advance?

In TM formalism:  $g(y, x) = \begin{cases} 1 & \text{if } f_y(x) \neq \perp \\ 0 & \text{if } f_y(x) = \perp \end{cases}$  does a TM that computes  $g$  exists?

### Answer

NO: determining if program terminates is unsolvable.

YES: determining if arithmetic expression terminates is instead solvable.

### Proof (Diagonal Technique)

Diagonal technique (from Cantor theorem: given a set, there's always a set of greater cardinality, even when using infinite sets) to show that  $\aleph_0 < 2^{\aleph_0}$ .

Assume by contradiction that  $g(y, x) = \begin{cases} 1 & \text{if } f_y(x) \neq \perp \\ 0 & \text{if } f_y(x) = \perp \end{cases}$  is computable.

Then  $h(x) = g(x, x) = \begin{cases} 1 & \text{if } f_x(x) \neq \perp \\ 0 & \text{if } f_x(x) = \perp \end{cases}$  is computable too:

go on diagonal  $y=x$ , change  $g(x, x)=0 \Rightarrow h(x)=1$  into a yes and  $g(x, x)=1 \Rightarrow h(x)=\perp$  into a nontermination.

If  $h$  is computable then  $\exists xh | h = f_{xh}$ , but  $h(xh) = 1 \vee h(xh) = \perp$  ?

- assume  $h(xh) = f_{xh}(xh) = 1$   
 $g(xh, xh) = 0 \Rightarrow f_{xh}(xh) = \perp \rightarrow$  contradiction
- assume  $h(xh) = f_{xh}(xh) = \perp$   
 $g(xh, xh) = 1 \Rightarrow f_{xh}(xh) \neq \perp \rightarrow$  contradiction

### Corollary

Given the general case  $g(y, x) = \begin{cases} 1 & \text{if } f_y(x) \neq \perp \\ 0 & \text{if } f_y(x) = \perp \end{cases}$  and a special case  $h(x) = \begin{cases} 1 & \text{if } f_x(x) \neq \perp \\ 0 & \text{if } f_x(x) = \perp \end{cases}$ .

if a **problem is unsolvable**, then:

- a **special case** of it **might be solvable**

if a **problem is solvable**, then:

- a **special case** of it is **solvable**

- a **more general** problem of it is **unsolvable**

- a **more general** problem of it **might be solvable**

## Problem of termination: “Undecidable problem”

$$k(y) = \begin{cases} 1 & \text{if } f_y(x) \neq \perp \forall x \in N(\text{total}) \\ 0 & \text{if } f_y(x) = \perp \end{cases}$$

Similar to previous problem, but here we have **quantification wrt all possible input data**.

Since  $x$  is in an **infinite set**, it's not possible to prove the property **for all the possible values**, while on the contrary it is possible to state that it is undecidable, because sooner or later we will find an  $x$  that satisfies the second property.

Talking about an algorithm, it is not known if it will terminate for all given input. Some inputs can therefore cause an infinite loop.

## Solvable problem might not be actually solvable

Problem is solvable if:

- there exists a TM that solves it
- reach conclusion that there exists a TM that solves it without actually being able to build one

Examples:

- “is it true that the number of atoms in the universe are  $10^{10^{10}}$  ?”

problem is solvable and it admits a TM that results either in true=1 or false=0 with function  $f(1)(x)=1 \forall x$  or  $f(0)(x)=0$ , but we don't know the answer yet.

- “in  $\pi$  do exists exactly  $x$  consecutive digits 5?”

can be written as  $g(x) = \begin{cases} 1 & \text{if } \exists x \text{ consecutive digits 5 in } \pi \\ 0 & \text{otherwise} \end{cases}$

How can  $g$  be computed?

Through  $f$  (where  $f(x) = x^{\text{th}}$  digit of the decimal expansion of  $\pi$ ):

compute seq  $\{f(0)=3, f(1)=1, f(2)=4, f(3)=1, f(4)=5, \dots\}$  (from  $\pi=3,1415\dots$ )

example, for:  $g(1)=1$

so:

- if  $g(x)=1 \rightarrow$  sooner or later we will find it (computable)
- if  $g(x)=0 \rightarrow$  impossible to compute for every  $\pi$  digits since they are infinite (not computable)

hence, it cannot be defined whether  $g$  is computable or not.

- “in  $\pi$  do exists at least  $x$  consecutive digits 5?”

can be written as  $h(x) = \begin{cases} 1 & \text{if } \exists \text{ at least } x \text{ consecutive digits 5 in } \pi \\ 0 & \text{otherwise} \end{cases}$  (if  $g(x) = 1$  then also

$h(x) = 1$  since it's a special case)

- if  $h(x) = 1 \rightarrow h(y) = 1 \forall y \leq x$  ( $h$  is downward closed, computable)
- if  $h(x) = 0 \rightarrow h(y) = 0 \forall y > x$  (computable)

hence,  $h$  is computable (even if we don't know the actual TM/TMs that solves it).

## Decidability and semidecidability

Given a binary (can be true or false) problem: given set  $S \subseteq N$  and  $x \in N : x \in S$  ?

$c_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$  with  $c_S$ , characteristic function, total by definition.

### **R (Recursive)**

$S$  recursive (or decidable)  $\Leftrightarrow c_S$  computable

### **RE (Recursive Enumerable)**

$S$  recursive enumerable RE (or semidecidable)  $\Leftrightarrow S = \emptyset \vee S = I_{g_S} = \{x \mid x = g_S(y), y \in N\}$

if the image of the computable function is enumerable, then soon or late will find an  $x$  and can get a correct answer to the problem.

Unsolvable problems are not RE (complement is RE, check “Dovetailing”).

### **Theorem**

1)  $S$  recursive (or decidable)  $\Rightarrow S$  RE (or semidecidable)

**proof:** assume  $S = \emptyset$  and  $c_S$  it's characteristic function.

$$\exists k \in S \rightarrow \exists k \mid c_S = 1$$

$$g_S(x) = \begin{cases} x & \text{if } c_S(x) = 1 \\ k & \text{otherwise} \end{cases} \text{ with } g_S \text{ total and computable and } I_{g_S} = S$$

hence  $S$  is recursive enumerable.

2)  $S$  recursive (or decidable)  $\Leftrightarrow S \wedge S^{\wedge} = N - S$  (complement) are RE

**proof:**

- $S$  recursive (or decidable)  $\Rightarrow S \wedge S^{\wedge} = N - S$  are R

◦  $S$  recursive (or decidable)  $\Rightarrow S$  RE (or semidecidable) proved before.

◦  $S$  recursive (or decidable)  $\rightarrow c_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases}$  (computable)

$$\rightarrow c_{S^c}(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases} \text{ (computable)}$$

$$\rightarrow S^c \text{ recursive} \rightarrow S^c \text{ RE}$$

•  $S$  recursive (or decidable)  $\Leftrightarrow S \wedge S^c = N - S$  are RE

$S$  RE  $\rightarrow$  build enumeration  $S = \{g_S(0), g_S(1), g_S(2), \dots\}$

$S^c$  RE  $\rightarrow$  build enumeration  $S^c = \{g_{S^c}(0), g_{S^c}(1), g_{S^c}(2), \dots\}$

then  $S \cup S^c = N \wedge S \cap S^c = \emptyset \rightarrow \forall x \in N, x \in S \vee S^c$  hence  $c_S$  can be computed.

### Other results

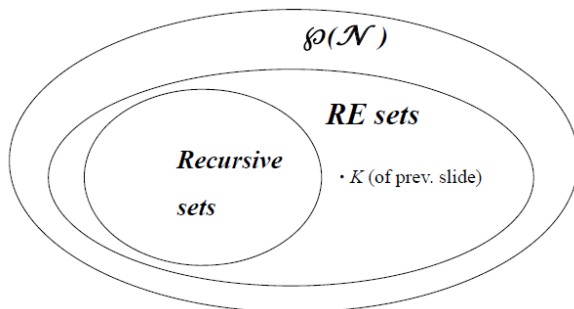
•  $S$  is RE  $\Leftrightarrow S = D_h(\text{domain}) = \{x | h(x) \neq \perp\}$

•  $S$  is RE  $\Leftrightarrow S = I_g(\text{image}) = \{x | x = g(y), y \in N\}$

• Exist semidecidable sets that are not decidable:

$K = \{x | f_x(x) \neq \perp\}$  semidecidable because  $K = D_h$  with  $h(x) = f_x(x)$ ,

but  $c_K(x)$  not computable  $\rightarrow K$  not decidable



Inclusions are all strict  
corollary: class of RE sets (language recognized by TM) is not closed wrt complement.

### Rice Theorem

Given: •  $F$  set of computable functions

•  $S = \{x | f_x \in F\}$  set of the indexes of TM that compute the functions of  $F$

Then: • in trivial cases:  $S$  is R (decidable)  $\Leftrightarrow F = \emptyset \vee F$  set of all computable functions

• in non trivial cases: not decidable. Following problems not solvable:

◦ Program correctness: "does  $P$  solves a given specified problem?"



- Program equivalence: “does P solve same problem as P’?”

How to determine whether problem is decidable or not:

Given an algorithm that:

- always terminates → problem decidable
- may not terminate → problem semidecidable (prove with Rice Theorem or reduction)

## **Reduction**

Method to prove that a problem is semidecidable. 2 ways method:

- **Given an algorithm to solve problem P one can use it to solve P’**

Want to solve  $x \in S'$  and I can solve  $y \in S \rightarrow$

If I have  $t_{total\ function} | x \in S' \Leftrightarrow t(x) \in S$  then can answer  $x \in S'$

ie: *Reduced problem  $x \in S'$  to problem  $y \in S$*

- **Opposite**

Want to know if I can solve  $x \in S$  and I know that cannot solve  $y \in S' \rightarrow$

If I find a  $t_{computable \wedge total} | y \in S' \Leftrightarrow t(y) \in S$  then I can conclude that  $x \in S$  is not decidable.

*Example: “Does a generic program P access an uninitialized variable?” is decidable?*

Assume that it is decidable and create instance P’ of program P:

```
begin
  var x, y; /* x and y are not in P */
  P;
  y = x;
end
```

It is clear that “y=x” results in an uninitialized var, because x and y do not occur in P.

Uninitialized var is accessed in P’ iff P terminates.

Can apply same technique to: array indices out of bound, division by 0, dynamic type compatibility, instruction execution, runtime errors, ...

## **Dovetailing theorem**

### **Problem**

halt of a TM, division by 0, runtime errors, ...

### **Type**

“Problem of determining if  $\exists z | f_x(z) \neq \perp$  is semidecidable”

**Semidecidable: if there’s an error I can find it, cannot find it’s absence (undecidable).**

1. if TM stops  $\rightarrow$  soon or late find it
2. if exists any  $x$  input of a program  $P$  such that  $P$  executed on input  $x$  eventually executes a division by 0  $\rightarrow$  soon or late I find it

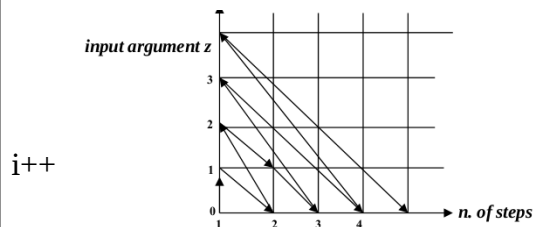
### Proof

$$f_x(0) = \begin{cases} \neq \perp & \text{ok, answer} \\ = \perp & \text{doesn't terminate and } f_x(1) \neq \perp, \text{ so??} \end{cases}$$

Compute  $f_x(i)$  (start with  $i=0$ )

- $f_x(i) \neq \perp$  ok, answer
- $f_x(i) = \perp$  (not terminate)

Diagonal technique "Dovetailing":



simulate 1 execution step of  $f_x(i)$  :

- $f_x(i) \neq \perp \rightarrow$  ok, answer
- else make one step for all  $f_x(\dots)$  previously considered and check for  $f_x(\dots) \neq \perp$

## Complexity of Computing

General principles to set individual problems into the correct framework to examine effectively them.

Once known that an algorithm is solvable, we want to know how much does it cost to solve it:

- **Time:** compilation, execution
- **Space:** memory

These considerations don't depend on how the problem is stated but on how it is solved.

### Complexity simplified

$f(x) \rightarrow f(n=|x|)$  with  $n$  as the size of the input data  $x$ . But in general  $|x_1|=|x_2| \Rightarrow T_M(x_1)=T_M(x_2)$ .

- **Worst case (most used)**

$$T_M(n) = \max\{T_M(x), |x|=n\} \quad (\text{same for } S_M(n))$$

- **Average case**

$$T_M(n) = \frac{\sum_{|x|=n} T_M(x)}{k^n} \quad \text{where } k = \text{cardinality of the alphabet}$$

## Θ notation

Θ estimate dominant factor in the growth of a complexity function  $f(n)$  and its asymptotic behaviour. Independent from computing device.

$$f \Theta g \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c, c \neq 0, c \neq \infty \quad \Theta \text{ is an equivalence relation}$$

example: given  $f(n) = n$  then  $T_M \in \Theta(n)$  (linear)

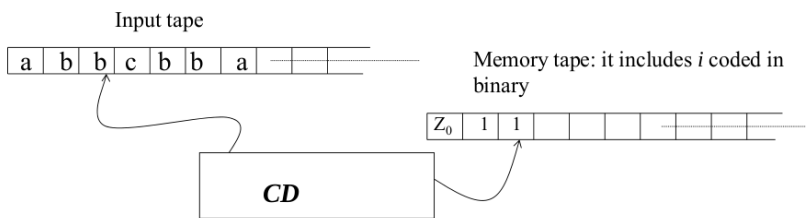
there can be trade-off in algorithms between time and space complexity.

Θ in computing models:

- FA:  $S_A(n)$  is  $\Theta(k)$ ,  $T_A(n)$  is  $\Theta(n)$  ( $T_A(n) = n \rightarrow$  FA are real-time machine)
- PDA:  $T_A(n)$

Example: given problem  $\{wcw^R\}$

, algorithm is: store in memory only the position “i” of the symbol to examine; then move the scanning head in position “i” and “n-i+1” to compare the 2 read symbols.



	$S_A(n)$	$T_A(n)$
<b>FA</b>	$\Theta(k)$	$\Theta(n)$ or $T_A(n) = n$ because FA are real-time machines
<b>PDA</b>	$\leq \Theta(n)$	$\Theta(n)$
<b>TM (single tape)</b>	$\Theta(n^2)$	$\Theta(n^2)$

Single tape TM more powerful than PDA, but sometimes less efficient.

## Linear speed-up theorems

### Space

Collapse  $r$  symbols of  $M$  into a single one in  $M'$  (thus enriching the alphabet).

If  $L$  is accepted by a  $k$ -tape TM  $M$  with complexity  $S_M(n)$  :

- Reduce space**

for each  $c > 0$  one can build a  $k$ -tape TM  $M'$  with complexity  $S_{M'}(n) < c \cdot S_M(n)$

- **Reduce #tapes**

build a 1-tape (not a single tape) TM  $M'$  with complexity  $S_{M'}(n) = S_M(n)$

- **Reduce space & #tapes**

for each  $c > 0$  one can build a 1-tape TM  $M'$  with complexity  $S_{M'}(n) < c \cdot S_M(n)$

## ***Time***

Collapse  $r$  symbols of  $M$  into a single one in  $M'$  (thus enriching the alphabet). And:

- read and translate all input  $\rightarrow n$  moves
- $M'$  can use a fixed number of moves to simulate groups of  $r$  moves of  $M$
- by suitably choosing  $r$  one can reduce the complexity by an arbitrary (linear) factor

Can't get less than  $\Theta(n)$ .

If  $L$  is accepted by a  $k$ -tape TM  $M$  with complexity  $T_M(n)$  :

- **Reduce time**

for each  $c > 0$  devise a  $(k+1)$ -tape TM  $M'$  with complexity  $T_{M'}(n) = \max\{n+1, c \cdot T_M(n)\}$

## ***Implications***

Proof scheme is valid for every computation model.

Order of magnitude cannot be changed (the  $\Theta$  class) by increasing parallelism (which leads to performance improvements as seen) but by changing the algorithm.

## **RAM Machine**

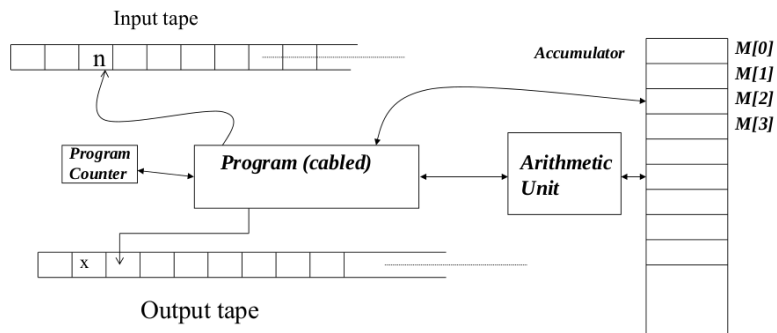
### ***TM vs real computers***

TM and real computers behave differently (with the latter being more performant):

- sum of 2 numbers: computers provide this operation as one machine instruction, while TM has to decode a number from a string that encodes the 2 numbers.
- Computers have direct access to any memory cell, while TM has only sequential access.

### ***RAM Machine schema***

Need a computational model that is more similar to how a computer behaves, the RAM Machine:



- every cell contains an integer
- **Program Counter:** points to next instruction to execute
- **Registers** ( $M[i]$  with  $i > 0$ )
- **Accumulator** ( $M[0]$  register): stores intermediate results

## RAM Machine instructions

	Syntax	Description	Log cost
Transfer from register to accumulator (and viceversa)	LOAD[*,*] X	Load integers from the register to the accumulator.	$\rightarrow l(x)$ $\rightarrow l(x) + l(M[x])$ $\rightarrow l(x) + l(M[x]) + l(M[M[x]])$
	STORE[*] X	Store integer in memory.	$\rightarrow l(x) + l(M[x])$ $\rightarrow l(x) + l(M[x]) + l(M[0])$
	ADD, SUB, MULT, DIV	Basic arithmetic operations between accumulator and registers. Result is stored in the accumulator.	$\rightarrow l(x) + l(M[x]) + l(M[0])$ $\rightarrow l(x) + l(M[x]) + l(M[0]) + l(M[M[x]])$
Transfer from input to registers and from registers to output	READ[*] X	Reads from input tape.	$\rightarrow l(x) + l(\text{curr input value})$ $\rightarrow l(x) + l(\text{curr input value}) + l(M[x])$
	WRITE[*,*] X	Writes on the output tape.	$\rightarrow l(x)$ $\rightarrow l(x) + l(M[x])$ $\rightarrow l(x) + l(M[x]) + l(M[M[x]])$
Change the PC conditionally or unconditionally	JUMP label	Change the PC unconditionally.	1
	JZ, JGZ, ... label	Change the PC if condition is met reading from accumulator: <ul style="list-style-type: none"> <li>• JZ # jump if =0</li> <li>• JGZ # jump if &gt;0</li> </ul>	$l(M[0])$
	HALT	Finishes computation	1

- $M[i]$  is referencing the value in the “i” registry
- label points to other instructions

- instruction semantic:

- **INSTR operand**: register index operand

example:  $LOAD\ X \rightarrow M[0] := M[X]$  # copy  $M[X]$  to  $M[0]$  (accumulator)

- **INSTR = operand**: immediate operand

example:  $M[0] := X$  # immediate value

- **INSTR \* operand**: indirect reference

example:  $LOAD *X \rightarrow M[0] := M[M[X]]$  #  $X$  address to cell  
which contains the index to the register which contains the value

Example: RAM program that calculate  $is\_prime(n) = \begin{cases} 1 & \text{prime} \\ 0 & \text{else} \end{cases}$

Read first int and subtract it to 1	READ 1 # read first int from input and store it on $M[1]$ LOAD= 1 # load value 1 into the accumulator ( $M[0]$ ) SUB 1 # subtraction $M[0] = M[0] - M[1]$ ( $M[0] = 1 -$ JZ YES $M[1]$ ) # if $M[0] == 0$ jump to "YES" label
Load $M[2] = 2$	LOAD= 2 # load value 2 in the acc ( $M[0] = 2$ ) STORE 2 # store value 2 in the registry ( $M[2] = M[0]$ )
Loop	LOOP: LOAD 1 # if $M[1] = M[2]$ then n is prime SUB 2 JZ YES  LOAD 1 # if $M[1] = (M[1] \text{ div } M[2]) * M[2]$ MULT 2 then $M[2]$ is a divisor of $M[1] \rightarrow M[1]$ not prime SUB 1 JZ NO  # increase $M[2]$ by 1 and repeat the loop LOAD 2 ADD= 1 STORE 2 JUMP LOOP
n is prime and terminate	YES WRITE= 1 HALT
n is not prime and terminate	NO WRITE= 0 HALT

Complexity:

- $S_R(n) = \Theta(2)$
- $T_R(n) = \Theta(n)$

with  $n$  being the input data size.

## RAM Machine problems

RAM Machine is too abstract:

- memory cell can store arbitrary big number
- any arithmetic operation has always unitary cost

Conceptual solution: should consider precision in all the operations

**Practical solution:** logarithmic cost  $\log(i)$  for load operations. As many elementary microoperations as the number of bits necessary to encode  $I$ .

## RAM costs

After calculating the cost of each instruction, can calculate the upper bound cost.

Example:

LOOP: LOAD 1	$1 + \log(n)$	Upper bound to content of the loop is $\Theta(\log(n))$
SUB 2	$\log(n) + 2 + \log(M[2])$	
JZ YES	$\log(M[0])$	
LOAD 1	$1 + \log(n)$	Hence the overall time complexity is $\Theta(n \cdot \log(n))$
MULT 2	$\log(n) + 2 + \log(M[2])$	
SUB 1	$\log(n/M[2]) + 2 + \log(M[2]) (< \log(n))$	
JZ NO	$\log(M[0]) + 1 + \log(n) < 2 \cdot \log(n) + 1$	
LOAD 2	$\leq \log(n)$	
ADD= 1	$\leq \log(n) + k$	
STORE 2		
JUMP LOOP		

Often cost can be calculated:

- overall (logarithmic criterion) cost = overall (constant criterion) cost \*  $\log(n)$
- overall (logarithmic criterion) cost = overall (constant criterion) cost \*  $\log(\text{overall constant criterion cost})$

Criterion to adopt:

- **Constant:** if input doesn't change the magnitude of the algorithm (example:  $n^2$ )
- **Logarithmic:** if input changes the magnitude of the algorithm (example:  $2^n$ )

## Complexity measures wrt different computation models

No M performs better than the others for all possible problems.

There's no Church thesis for Complexity but:

## Polynomial Correlation Theorem

If problem can be solved by computation model  $M_1$  with complexity  $C_1(n)$ , then it can be solved by any other computation model  $M_2$  with complexity  $C_2(n) \leq P_2(C_1(n))$ , with  $P_2$  being a suitable polynomial.

Polynomials always better than exponential → **Tractable problems**: problems that can be solved in polynomial time P.

## ***Time correlation between TM and RAM***

### **RAM simulates a k-tape TM**

#### Structure

1 RAM cell for each cell of the TM tape.

k cells block of RAM associated to each k-tuple of cells taken for each position of the tape + a base block.

- block 0 (K+1 cells): state+position of the K heads
- block 1 → i (K cells): i-th symbol of every tape

#### Execution

Read h:

1. check content of the 0 block (k+1 accesses)
2. k indirect accesses to k blocks to examine the content of the cells corresponding to the heads

Writing h:

1. state is changed in block 0
2. update content of the cells corresponding to the head positions through indirect STORE ops
3. update values of the k heads in block 0

#### Costs

TM move requires  $h \cdot k$  RAM moves:

- $T_R \in \Theta(T_M)$  constant cost
- $T_R \in \Theta(T_M \cdot \log(T_M))$  logarithmic cost

### **TM simulates a RAM**

#### Structure

3 tapes:

- encodes content of the RAM memory

...	\$	i <sub>j</sub>	#	M[i <sub>j</sub> ]	\$	...	\$	i <sub>k</sub>	#	M[i <sub>k</sub> ]	\$	...
-----	----	----------------	---	--------------------	----	-----	----	----------------	---	--------------------	----	-----



- RAM cells are kept in order
- initially empty tape, after some time it only includes the cells that have an assigned value
- $i_j$  and  $M[i_j]$  represented in binary encoding
- $M[0]$  in binary
- “service” tape

## Execution

LOAD h:

1. search the value h on the main tape (if not found  $\rightarrow$  error)
2. copy the part next to  $M[h]$  into  $M[0]$

STORE h

1. search the value h. if found make a “hole” using the service tape
2. store h and copy  $M[0]$  in the part next to it ( $M[h]$ ). copy the successive part from the service tape
3. if h already exists copy  $M[0]$  in the part next to it ( $M[h]$ );  
“service tape” required if #cells already occupied  $\leq$  to the of  $M[0]$

## Costs

Simulating a RAM can take a #moves with an upper bound  $c \cdot$  (length of the main tape that stores the content of the RAM memory).

Lemma: the length of the main tape has an upper bound equal to a function that is  $\Theta(T_R)$ .

Each  $i_j$  cell:

- of the RAM requires in the tape  $l(i_j) + l(M[i_j]) + 2$  tape cells.
- Exists in the tape iff the RAM has executed at least a STORE in it

Store cost for the RAM:  $l(i_j) + l(M[i_j])$

Hence the RAM needs a time (wrt logarithmic criterion) that is at least proportional to the same value.

Simulate RAM move: TM needs a time at most  $\Theta(T_R)$

RAM move costs at least 1  $\rightarrow$  since RAM has at least complexity  $T_R$  it executes at most  $T_R$  moves  
 $\rightarrow$  the complete simulation of the RAM by the TM costs at most  $\Theta(T_R^2)$ .

## Warnings

- Check “data dimension” parameter:
  - input string length (absolute value)
  - value of datum (n)
  - ...
- in practice avg case