Fundamentals of theory of computation 2

3rd lecture

lecturer: Tichler Krisztián
ktichler@inf.elte.hu

Basic notions, notations – Summary

A SUMMARY OF FOTOC 1

Alphabet: A finite, nonempty set

Letters: Elements of the alphabet.

Finite sequences over an alphabet V are called **words** or strings.

The **length of a word** $u = t_1 \cdots t_n$ is n, the number of letters of u. Notation: $|u| = n$. The word of length 0 is denoted by ε, and is called the **empty word** ($|\varepsilon| = 0$).

V^* is the notation for the **set of all words over V** including the empty word.

$V^+ = V^* \backslash \{\varepsilon\}$ is the **set of all non-empty words over V**.

Example: $V = \{a, b\}$, then
$V^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$.

Basic notions, notations – Summary

Let V be an alphabet, a subset L of V^* is called a **language** over V.

The empty language (the language containing no words) is denoted by $\varnothing$.
A language is called a **finite language** if it contains finite words.

Let X be a set. $\mathcal{P}(X)$ denotes the **powerset** of X, i.e.,
$\mathcal{P}(X) = \{A \mid A \subseteq X\}$.

Family of languages (or class of languages) is a set of languages.

So for an alphabet V
- $a \in V$: letter
- $u \in V^*$: word
- $L \subseteq V^*$ or $L \in \mathcal{P}(V^*)$: language
- $\mathcal{L} \subseteq \mathcal{P}(V^*)$ or $\mathcal{L} \in \mathcal{P}(\mathcal{P}(V^*))$: family of languages

Grammars – Summary

Definition

A 4-tuple $G = \langle N, T, S, P\rangle$ is called a **grammar** if
- N and T are disjoint alphabets (i.e., $N \cap T = \varnothing$). Elements of N are called **nonterminals**, while elements of T are called **terminals**.
- $S \in N$ is the **start symbol** of the grammar.
- P is a **set of production rules**, each of them having the form $x \to y$, where $x \in (N \cup T)^* N (N \cup T)^*, y \in (N \cup T)^*$.

Example: For a grammar $G = \langle \{S\}, \{a\}, S, \{S \to aaS, S \to \varepsilon\}\rangle$
$S \Rightarrow aaS \Rightarrow aaaaS \Rightarrow aaaa$ is a **derivation** of the word $aaaa$.

The set of terminal words that can be generated from the start symbol of grammar G is called **the language generated by G** and denoted by $L(G)$.

For this example $L(G) = \{a^{2n} \mid n \in \mathbb{N}\}$.

Chomsky grammar classes – Summary

Let $G = \langle N, T, S, P\rangle$ be a grammar. G is of **type** i ($i = 0, 1, 2, 3$), if P satisfies the following conditions

- for $i = 0$: no restriction,
- for $i = 1$:
  (1) the rules of P are of the form $u_1 A u_2 \to u_1 v u_2$, where $u_1, u_2, v \in (N \cup T)^*, A \in N$, and $v \neq \varepsilon$,
  (2) there is one possible exception: $S \to \varepsilon$ can be included in P, but only in the case when S does not appear on the right hand side of any rule of P.
- for $i = 2$: all rules of P are of the form $A \to v$ where $A \in N$ and $v \in (N \cup T)^*$,
- for $i = 3$: all rules of P are either of the form $A \to uB$ or of the form $A \to u$, where $A, B \in N$ and $u \in T^*$.

Let $\mathcal{G}_i$ denote the class of grammars of type i ($i = 0, 1, 2, 3$).

Chomsky hierarchy – Summary

$\mathcal{L}_i := \{L \mid \exists G \in \mathcal{G}_i,$ such that $L = L(G)\}$ denotes the **family of languages of type** i. ($i = 0, 1, 2, 3$).

Type 0 grammars are called **phrase-structure** grammars, type 1 grammars are called **context-sensitive** grammars, type 2 grammars are called **context-free** grammars. Type 3 grammars are called **regular** grammars.

The corresponding language classes are called the class of **recursively enumerable**, the class of **context-sensitive**, the class of **context-free**, and the class of **regular** languages.

Chomsky hierarchy

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0.$$

Remark: Inclusion between $\mathcal{L}_2$ and $\mathcal{L}_1$ does not follow immediately form the forms of the corresponding grammars. Proper inclusions can be proved by pumping (Bar Hillel) lemmas.

Defining formal languages – Summary

- by (generative) grammars

Grammars are **synthetizing** tools, starting form a single symbol they can build words. The set of terminal words that can be built forms a formal language.

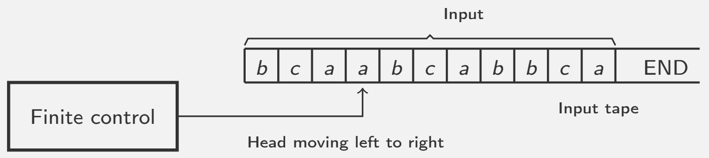
- by automata

Automata are parsing, **analyzing** tools. An input word is processed by the automata and a single bit ('yes' or 'no') is given as an output. The set of those input words giving a 'yes' output forms a formal language.

- by other ways: listing elements, regular expressions, etc.

Attention! It is not true, that all languages can be defined by each of these tools. E.g., regular expressions can not describe all type 0 languages. On the other hand, not even type 0 grammars are strong enough to describe all languages over $\{0, 1\}$.

Finite automata – Summary



- Finite automata (FA) start from their initial state. Initially an input word is written on the input tape. A head starts from the first letter and scans the input from left to right.
- The automaton works in discrete steps, in one step it reads a symbol from the tape, the state changes according to the transition function and the head moves to the right.
- While there are symbols left from the input the automaton keeps working. It does not matter if it reaches an accepting state if there are still letters to be processed. It decides on the input only at the point when the input is fully processed.

Finite automata – Summary

Definition

A (nondeterministic) **finite automaton** is a 5-tuple,
$A = \langle Q, T, \delta, Q_0, F \rangle$, where

- Q is a non-empty set, the set of states
- T is the alphabet of input symbols,
- $\delta : \ Q \times T \to \mathcal{P}(Q)$ is the transition function,
- $Q_0 \subseteq Q$ is the set of initial states,
- $F \subseteq Q$ is the set of accepting states.

Definition

If $|\delta(q, a)| = 1$ and $|Q_0| = 1$ holds $\forall (q, a) \in Q \times T$ then A is called a **deterministic finite automaton** (DFA).

Type 3 languages – Summary

The **language accepted by** a finite automaton $A = \langle Q, T, \delta, Q_0, F \rangle$ consists of those words w of T for which a state of F can be reached by fully processing w.

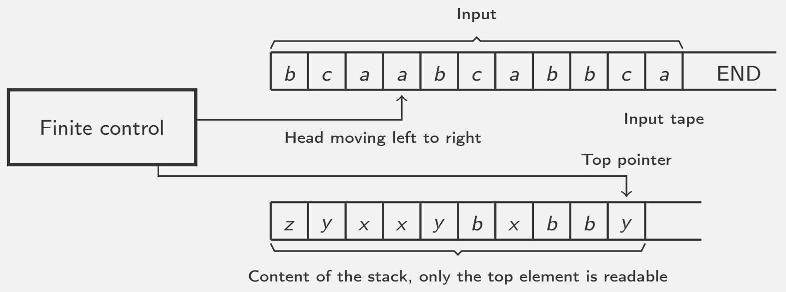
For DFA's there is exactly one way to process the input words.

Theorem

The following formal tools define exactly the type 3 languages.

- Type 3 grammars
- Type 3 grammars in normal form
- Regular expressions
- Deterministic finite automata
- Nondeterministic finite automata

Pushdown automata – Summary


Input
b c a a b c a b b c a END
Finite control
Head moving left to right
Input tape
Top pointer
z y x x y b x b b y
Content of the stack, only the top element is readable

- Pushdown automaton (PDA) is a generalization of FA, with a stack of unbounded capacity and finite control.
- New data is pushed into the stack over the current content, elements can popped one by one in an opposite order as they were pushed into the stack.
- pushdown automata are nondeterministic by default

Pushdown automaton – Summary

Notation: Let X be a set, $\mathcal{P}_{\text{fin}}(X)$ denotes the set of finite subsets of X.

Definition

A **pushdown automaton** is a 7-tuple $A = \langle Z, Q, T, \delta, z_0, q_0, F \rangle$, where

- Z is the stack alphabet
- Q is a non-empty finite set of states,
- T is the input alphabet,
- $\delta : Z \times Q \times (T \cup \{\varepsilon\}) \to \mathcal{P}_{\text{fin}}(Z^* \times Q)$ is the transition function
- $z_0 \in Z$ is the initial stack symbol,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states.

Pushdown automata – Summary

- Transitions depend on the current state, the symbol processed from the input and the top element of the stack.
- In every step the top element of the stack is removed and replaced by some $(0, 1, 2, \ldots)$ letters from Z.
- If $\delta(z, q, \varepsilon)$ is non-empty, then so called **ε-moves** are possible. Such moves change the state and the content of the stack without processing any symbol of the input.
- ε-moves are possible even before processing the first letter and even after processing the last one.

Pushdown automata – Summary

There are 2 acceptance modes:

The **language accepted by** a pushdown automaton $A = \langle Z, Q, T, \delta, z_0, q_0, F\rangle$ **by accepting states** consists of words w over T having a computation ending in a state from F and fully processing w.

Computation is nondeterministic and the content of the stack is irrelevant at the end of computation. This language is denoted by $L(A)$.

The **language accepted by** a pushdown automaton $A = \langle Z, Q, T, \delta, z_0, q_0, F\rangle$ **by empty stack** consists of words w over T having a computation ending with an empty stack and fully processing w.

Computation is nondeterministic and the current state is irrelevant at the end of computation. This language is denoted by $N(A)$.

Pushdown automata – Summary

Deterministic pushdown automaton

Definition

A pushdown automaton $A = \langle Z, Q, T, \delta, z_0, q_0, F\rangle$ is called **deterministic** if $|\delta(z, q, a)| + |\delta(z, q, \varepsilon)| = 1$ holds for every $(z, q, a) \in Z \times Q \times T$.

Therefore for every $q \in Q$ and $z \in Z$ we have

- either $\delta(z, q, a)$ contains exactly one element for every input symbol $a \in T$ and $\delta(z, q, \varepsilon) = \varnothing$,
- or $\delta(z, q, \varepsilon)$ contains exactly one element and $\delta(z, q, a) = \varnothing$ for every input symbol $a \in T$.

Remark: If $|\delta(z, q, a)| + |\delta(z, q, \varepsilon)| \leqslant 1$ for every $(z, q, a) \in Z \times Q \times T$ then, by adding extra transitions to a new, trap state, the pushdown automaton can be extended to a deterministic PDA of the same accepted language.

Pushdown automata – Summary

PDA's and type 2 languages

Theorem

The following statements are equivalent for every language L

- L is context-free (can be generated by a context-free grammar),
- L can be generated by a grammar in Chomsky NF,
- $L = L(A)$ for some (nondeterministic) PDA A,
- $L = N(A)$ for some (nondeterministic) PDA A

Theorem

Every regular (type 3) language can be recognized by a deterministic PDA. On the other hand, there are CF languages which can not be recognized by a deterministic PDA.

Strict types of languages – Summary

Notation: $|u|_t$: number of occurances of letter t in u

Examples:

$\in \mathcal{L}_3$	$\in \mathcal{L}_2 - \mathcal{L}_3$	$\in \mathcal{L}_1 - \mathcal{L}_2$
$\{u \mid abbab \subseteq u\}$	$\{u \in \{a,b\}^* \mid u = u^R\}$	$\{uu \mid u \in \{a,b\}^*\}$
$\{u \mid abbab \nsubseteq u\}$	$\{a^n b^n \mid n \in \mathbb{N}\}$	$\{a^n b^n c^n \mid n \in \mathbb{N}\}$
numbers divisible by 7	$\{u \in \{a,b\}^* \mid \lvert u\rvert_a = \lvert u\rvert_b\}$	$\{a^{n^2} \mid n \in \mathbb{N}\}$
$((a+bb)^* + ab)^*$	well-parenthized expr.	$\{a^{2^n} \mid n \in \mathbb{N}\}$

Algorithmic problems

For a problem $\mathcal{P} : \mathcal{I} \to \mathcal{O}$ (where $\mathcal{I}$ is the set of inputs, $\mathcal{O}$ is the set of outputs) an **algorithmic solution** is a procedure with a common list of instructions calculating $\mathcal{P}(I) \in \mathcal{O}$ for every $I \in \mathcal{I}$ input of $\mathcal{P}$.

If $\mathcal{O} = \{\text{'yes'}, \text{'no'}\}$, then we say that $\mathcal{P}$ is a **decision problem**, for the general case we say $\mathcal{P}$ is a **counting problem**.

Examples:

- $\mathcal{I} = \mathbb{N} \times \mathbb{N}$, $\mathcal{O} = \mathbb{N}$. $\mathcal{P}$ is addition. Algorithmic solution: the algorithm we learn in elementary school.
- $\mathcal{I} = \{G \mid G \text{ is a CF grammar}\} \times T^*$, $\mathcal{O} = \{\text{'yes'}, \text{'no'}\}$, where T is an alphabet. $\mathcal{P}$ is membership problem. Algorithmic solution: CYK algorithm.
- $\mathcal{I} = \{\text{first order formulas}\}$, $\mathcal{O} = \{\text{'yes'}, \text{'no'}\}$, $\mathcal{P}(\varphi) =$'yes', iff $\models \varphi$. No algorithmic solution is known.

Decision problems

Given an algorithmic decision problem $\mathcal{P}$. **yes-instances** are those inputs having the output 'yes', **no-instances** are those inputs having the output 'no'.

Yes-instances can be considered as a formal language $L_{\mathcal{P}} = \{I \in \mathcal{I} \mid \mathcal{P}(I) = \text{'yes'}\}$ over an appropriate alphabet.

A **partial algorithmic solution** is an algorithm giving the answer 'yes' exactly for $L_{\mathcal{P}}$ not necessarily terminating for no-instances. A **decision algorithm** is an algorthm terminating for all inputs and giving the answer 'yes' exactly for the words of $L_{\mathcal{P}}$.

Decision problems and automata

So far we have learnt some types of parsing automata (DFA, deterministic PDA) giving a 'yes'/'no' output for the input words. These automata can be considered as decision algorthms (they follow a finite set of instructions) for the language recognized by the automata as they terminate for all words and give the answer 'yes' for exactly the words of this language.

So these automata can be considered as a kind of restricted models for algorithms.

We can say that a problem $\mathcal{P}$ is algorhmically solved (decided) if there is an automaton recognizing exactly the words of $L_{\mathcal{P}}$.

Decision problems and automata

PDA is a generalization of FA, more problems can be decided by a PDA than by a FA.

Is PDA general enough to model the concept of algorithm?

No, PDA's can recognize only CF (type 2) languages. On the other hand, elements of an $\mathcal{L}_0 (\supset \mathcal{L}_2)$ language can be synthetized by an algorithm (by a type 0 grammar).

Is there an automata model (or any formal tool, model of computation) strong enough to decribe the concept of an algorithm?

Models of computation (algorithm)

From 1930's several models of computation were introduced with the intention to define what is an algorithm, what is computability. Some of these models were

- Kurt Gödel's recursive functions
- Alonso Church's (Kleene, Rosser): λ-calculus
- Alan Turing's Turing machines

Which of them is the "real" one?

From the middle of 1930's several theorems were proven stating equivalence of some of these models.
Later further models were introduced. Several of them turned to be having the same computational power as Turing machines. Some of these were

- type 0 grammars
- PDA's with 2 or more stacks
- C, Java, etc.

The Church-Turing thesis

Actually, we do not know any model of computation with greater computational power than TM's. For most models it is proved to have the same or less computational power than Turing machines.

The following thesis appeared already in the 1930's

Church-Turing thesis

The above models for computable functions all describe the informal notion of effectively calculable functions.

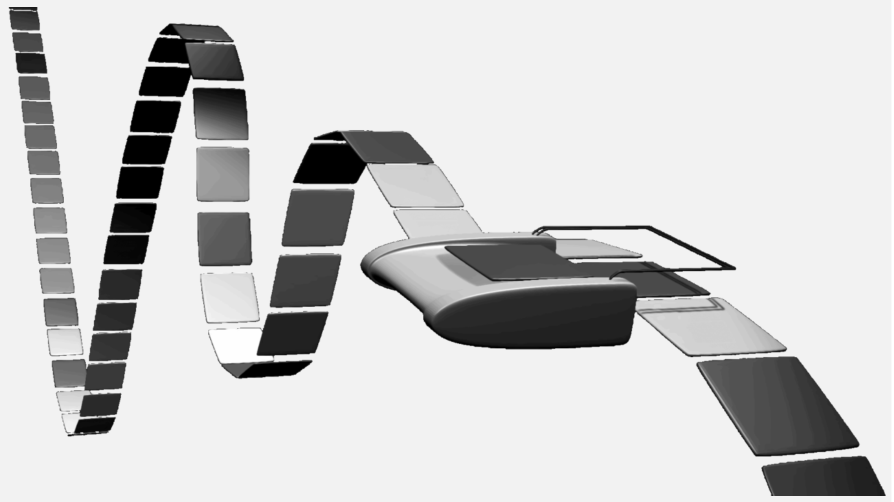
'effectively calculable': can be computed by a 'paper-pencil' method.

By other words: every formalizable problem, that can be solved by an algorithm can be solved by a Turing machine as well (or in any computational model equivalent with Turing machines)

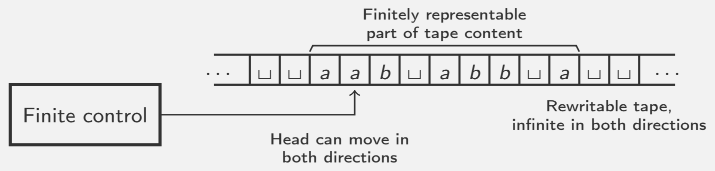
Not a theorem!!!. An intutive hypothesis, can not be proved.

Accepting the thesis, intuitively, we can consider any of the above concepts as a mathematical model for algorithms (computability).

Turing machines



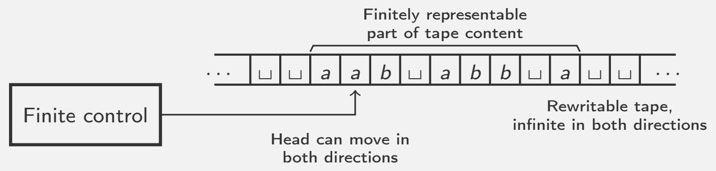
Turing machines – Informal introduction



Finitely representable part of tape content
… ⊔ ⊔ a a b ⊔ a b b ⊔ a ⊔ ⊔ …
Finite control
Head can move in both directions
Rewritable tape, infinite in both directions

- a Turing machine (TM) is a model for the concept of algorithm
- a TM is programmed to solve a specific problem (but for any input)
- parts of the machine are (informally): finite control (finitely many states); a tape, infinite in both directions; a head capable for moving in both directions
- initially there's an input word w on the tape (the tape is empty in the case of $w = \varepsilon$ input), the head starts from the first letter of w and the head moves according to transition rules. It accepts in its single accepting state, rejects in its single rejecting state. There's a 3rd possibility: "infinite loop"

Turing machines – Informal introduction



Finitely representable part of tape content
… ⊔ ⊔ a a b ⊔ a b b ⊔ a ⊔ ⊔ …
Finite control
Head can move in both directions
Rewritable tape, infinite in both directions

- the machine is deterministic by default, transitions are well defined in all cases
- infinite tape means infinite storage
- for a problem $\mathcal{P}$ yes-instances form a formal language $L_{\mathcal{P}}$ (with an appropriate coding). $L_{\mathcal{P}}$ (and so the problem itself as well) is decidable if there's an always terminating TM accepting exactly the words of $L_{\mathcal{P}}$.
- according to Church-Turing thesis the problems decidable by a TM corresponds to the algorithmically decidable problems.

Turing machines – Formal definition

Definition

A Turing machine (TM from now) is a $M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r \rangle$ 7-tuple, where

- Q is a finite, non-empty set of states,
- $q_0, q_a, q_r \in Q$, q_0 is the starting q_a is the accepting q_r is the rejecting state,
- Σ and Γ are alphabets, the input alphabet and the tape alphabet respectively $\Sigma \subseteq \Gamma$ and $\sqcup \in \Gamma \backslash \Sigma$.
- $\delta : (Q \backslash \{q_a, q_r\}) \times \Gamma \to Q \times \Gamma \times \{L, S, R\}$ is the transition function.

The set $\{L, S, R\}$ is the set of directions (left, stay, right).

Configurations of Turing machines

Definition

A configuration of a TM is a word uqv , where $q \in Q$ and $u, v \in \Gamma^*, v \neq \varepsilon$.

A configuration uqv briefly describes the current situation with the TM. It contains all relevant information: the machine is in state q, the content of the tape is uv (only $\sqcup$'s before and after) and the head is on the first letter of v.

Two configurations are considered to be the same if they differ only in $\sqcup$'s on the left or on the right.

For a word $u \in \Sigma^*$ the starting configuration is the word $q_0 u \sqcup$ (i.e, it is $q_0 u$ if $u \neq \varepsilon$, and $q_0 \sqcup$, if $u = \varepsilon$).

Accepting configurations are the configurations, where $q = q_a$. Rejecting configurations are the configurations, where $q = q_r$. A halting configuration is either an accepting or a rejecting configuration.

One step transition of TM's

Let C_M be the set of all possible configurations for TM M. M the $\vdash \subseteq C_M \times C_M$ **one step transition relation** is defined as follows.

$\vdash \subseteq C_M \times C_M$ one-step transition relation

Let $uqav$ be a configuration, where $a \in \Gamma,\ u, v \in \Gamma^*$.

- If $\delta(q, a) = (r, b, R)$, then $uqav \vdash ubrv'$, where $v' = v$, if $v \neq \varepsilon$, otherwise $v' = \sqcup$,
- if $\delta(q, a) = (r, b, S)$, then $uqav \vdash urbv$,
- if $\delta(q, a) = (r, b, L)$, then $uqav \vdash u'rcbv$, where $c \in \Gamma$ and $u'c = u$, if $u \neq \varepsilon$, otherwise $u' = u$ and $c = \sqcup$.

If $C \vdash C'$ we say that **C yields C' in one step**.

Example: Suppose, that $\delta(q_2, a) = (q_5, b, L)$ and $\delta(q_5, c) = (q_1, \sqcup, R)$. Furthermore let $C_1 = bcq_2a\sqcup b$, $C_2 = bq_5cb\sqcup b$, $C_3 = b\sqcup q_1b\sqcup b$.
Then $C_1 \vdash C_2$ és $C_2 \vdash C_3$.

Multistep transition of TM's; recognized language

Multistep transition relation: reflexive, transitive closure of $\vdash$ denoted by $\vdash^* \subseteq C_M \times C_M$.

Definition

$C \vdash^* C' \Leftrightarrow$

- $C = C'$ or
- $\exists\, n > 0 \wedge C_1, C_2, \ldots, C_n \in C_M$, then $\forall\, 1 \leqslant i \leqslant n-1$ $C_i \vdash C_{i+1}$ holds, furthermore $C_1 = C$ and $C_n = C'$.

If $C \vdash^* C'$ we say that **C yields C' in finite steps**.

Example (cont'd): Let C_1, C_2, C_3 as in the previous example. We have seen, that $C_1 \vdash C_2$ and $C_2 \vdash C_3$ holds, so $C_1 \vdash^* C_1$, $C_1 \vdash^* C_2$, $C_1 \vdash^* C_3$ hold, too.

Definition

The **language recognized by a TM** $M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r\rangle$ is $L(M) = \left\{u \in \Sigma^* \mid q_0u\sqcup \vdash^* xq_ay \text{ for some } x, y \in \Gamma^*, y \neq \varepsilon\right\}$.

RE and R

Definition

$L \subseteq \Sigma^*$ is **Turing-recognizable**, if $L = L(M)$ holds for some TM M. In this case we say that such a TM M **recognizes** L.

$L \subseteq \Sigma^*$ is **decidable**, if there is a TM M halting on all inputs and $L(M) = L$. In this case we say that such a TM M **decides** L.

Other names for Turing-recognizable are **recursively enumerable** (or *partially decidable*, or *semidecidable*). Another name for decidable is **recursive**.

Definition

The class of recursively enumerable languages is denoted by RE, the class of recursive languages is denoted by R.

Obviously $R \subseteq RE$ holds. Is it true, that $R \subset RE$?

Turing machines – Running time

Definition

The **running time** of a TM M on a word u is the number of steps (transitions) M takes from the starting configuration of u to a halting configuration. If there is no such number, then running time of M on u is infinity.

Definition

Time complexity of a TM M is a function $f : \mathbb{N} \to \mathbb{N}$, such that $f(n)$ is the maximum running time on any input of length n.

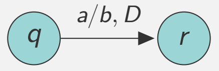
We also say that M is an **$f(n)$ time** Turing machine or the **running time** of M is $f(n)$.

Note, that if M is an $f(n)$ time TM, then for any input word $u \in \Sigma^*$ M has a running time of at most $f(|u|)$ on u.

In several cases we are satisfied with a good asymptotic upper bound on time complexity.

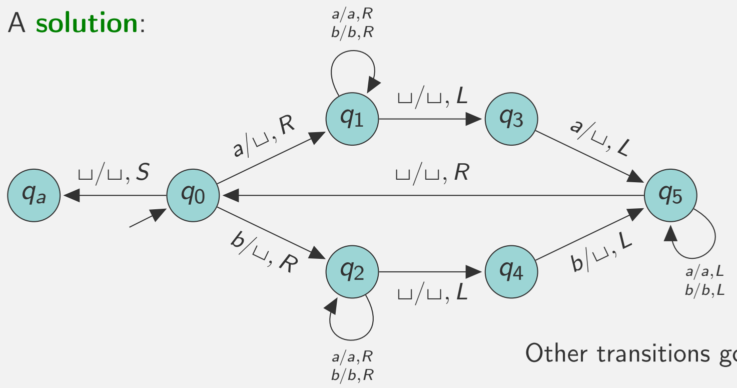
Turing machines – An example

Exercise: Construct a Turing machine M so, that
$L(M) = \{ww^R \mid w \in \{a, b\}^*\}$!
Transition diagram.



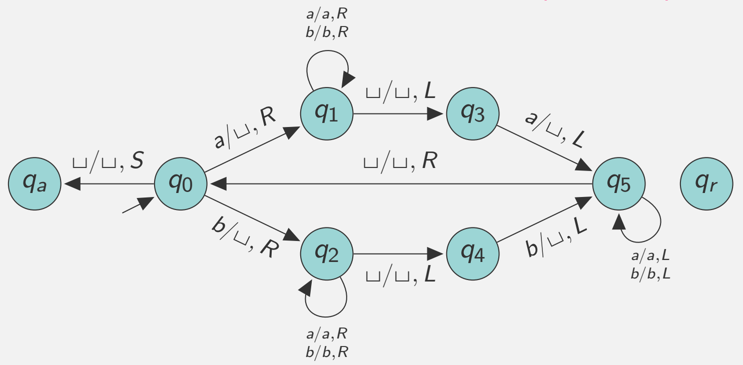
notation of $\delta(q, a) = (r, b, D)$
$(q, r \in Q, a, b \in \Gamma, D \in \{L, S, R\})$

A **solution**:



Other transitions go to q_r.

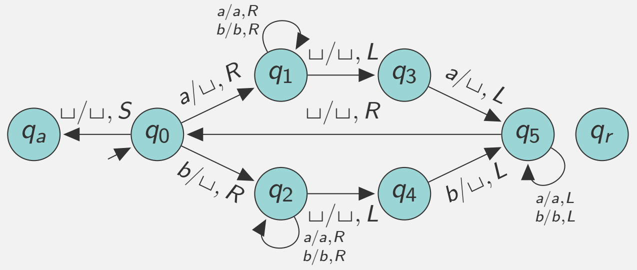
Turing machines – An example (cont'd)



An example for the sequence of transitions for the input *aba*:
$q_0 aba \vdash q_1 ba \vdash bq_1 a \vdash baq_1 \sqcup \vdash bq_3 a \vdash q_5 b \vdash q_5 \sqcup b \vdash q_0 b \vdash q_2 \sqcup \vdash q_4 \sqcup \vdash q_r \sqcup$.

For the input *aba* the machine reaches a halting configuration in 10 steps. In this example we may have been able to compute the exact time complexity. But sometimes it is simpler (and enough to) give a good asymptotic upper bound.

Turing machines – An example (cont'd)



It is a $O(n^2)$ time TM, since there are $O(n)$ steps in each of the $O(n)$ iterations, +1 step to go to either q_a or q_r.

Is there a better asymptotic upper bound for the time complexity? **No**, there are infinite words with $\Omega(n^2)$ steps.

Is this TM decides the language $L = \{ww^R \mid w \in \{a, b\}^*\}$ or "just" recognizes it? **This TM decides L.**

Is there a TM, which recognizes L but does not decide it? **Yes**, change the transitions going to q_r by redirecting them to a new state with an infinite loop, i.e., no transition going out of that state.