Vertiefung Theoretische Informatik

Advanced Topics in Theoretical Computer Science

Viorica Sofronie-Stokkermans

Universität Koblenz-Landau

e-mail: sofronie@uni-koblenz.de

Acknowledgments

In preparing this lecture we used slides from the lecture of

Bernard Beckert, Theoretische Informatik II (held in Koblenz in 2007/2008;

based on slides by Katrin Erk and Lutz Priese

and on slides by Christoph Kreitz)

Many thanks!

Literature

Book:

Katrin Erk and Lutz Priese:
Theoretische Informatik: Eine umfassende Einführung.
2. Auflage.
Springer-Verlag.

Further literature

- J. Hopcroft, R. Motwani, and J. Ullman (2002).
  Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.
  Pearson.

- G. Vossen and K.-U. Witt (2004).
  Grundkurs Theoretische Informatik.
  Vieweg.

- U. Schöning (1994).
  Theoretische Informatik: kurzgefasst.
  Spektrum-Verlag.

- J. Hromkowitz (2011). Theoretische Informatik
  4. Auflage
  Studium.

Introduction

- Details about the lecture
- Motivation
- Contents

Motivation

Theoretical Computer Science studies fundamental concepts in computer science:

- Problems and their description
- Systems/Automata/Machines which solve problems
- "Solvability" of Problems
  (Computability/Decidability and their limits)
- Difficulty (complexity) of solving problems

Areas of Theoretical Computer Science

- Formal Languages
- Automata Theory
- Computability Theory
- Complexity Theory
- (Logic)

Focus of this lecture

- Formal Languages
- Automata Theory
- **Computability Theory**
- **Complexity Theory**
- (Logic)

Importance

Why is Theoretical Computer Science important?

Importance

Why is Theoretical Computer Science important?

Theoretical Computer Science

- is the “fundament” of computer science
- is important e.g. for:
  algorithm techniques, software engineering, compiler construction
- helps in understanding further topics/lectures in computer science
- does not get “old”
- is fun!

The pragmatical view

The pragmatical view

Assume you are employed as software designer.

The pragmatical view

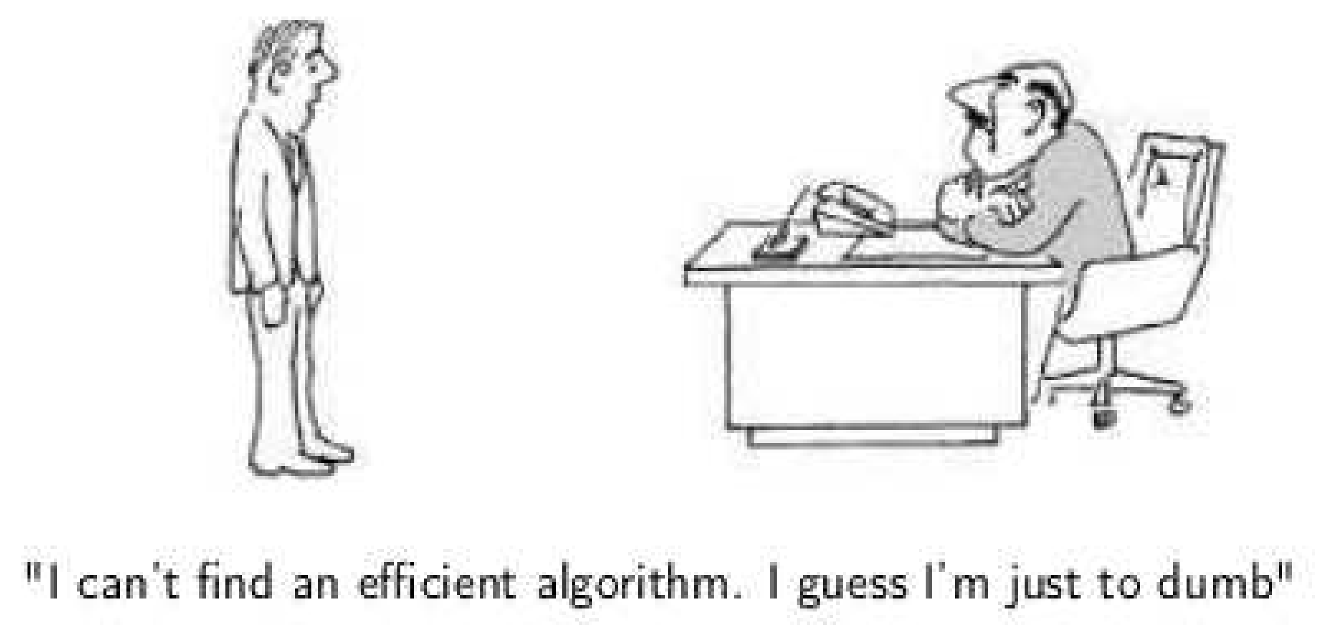
Assume you are employed as software designer.

One day, your boss calls you into his office and and tells you that the company is about to enter a very competitive market, for which it is essential to know how to solve (efficiently) problem X.

Your charge is to find an efficient algorithm for solving this problem.

The pragmatical view

What you certainly don't want:



(Garey, Johnson, 1979)

The pragmatical view

It would be much better if you could prove that problem X is inherently intractable, i.e. that no algorithm could possibly solve it quickly.

The pragmatical view

Much better:

"I can't find an efficient algorithm, because no such algorithm is possible!"

(Garey, Johnson, 1979)

The pragmatical view

In this lecture we will show for instance how to prove that certain problems do not have a (terminating) algorithmic solution

$\mapsto$ undecidability results

Unfortunately, proving inherent intractability can be just as hard as finding efficient algorithms.

The pragmatical view

However, we will see that you can often answer:

"I can't find an efficient algorithm, but neither can all these famous people."

(Garey, Johnson, 1979)

Contents

- Recall: Turing machines and Turing computability
- Register machines (LOOP, WHILE, GOTO)
- Recursive functions
- The Church-Turing Thesis
- Computability and (Un-)decidability
- Complexity
- Other computation models (brief discussion)

Contents

- **Recall: Turing machines and Turing computability**
- Register machines (LOOP, WHILE, GOTO)
- Recursive functions
- The Church-Turing Thesis
- Computability and (Un-)decidability
- Complexity
- Other computation models (brief discussion)

Computability/Turing Machines: Idea

What is a problem?

Informally:

for certain inputs

certain outputs must be produced.

What is a problem?

Informally:

for certain inputs

certain outputs must be produced.

More precise definition:

Input: Word over alphabet Σ

Output: Word(s) over alphabet Σ

What is a problem?

Informally:

for certain inputs

certain outputs must be produced.

More precise definition:

Input: Word over alphabet Σ

Output: Word(s) over alphabet Σ

Problem as relation $R \subseteq \Sigma^* \times \Sigma^*$

(x, y) is in R, if y is a possible output for input x.

Problem as function

Often, for every input there exists a unique output.

In this case, we can represent a problem as a function $f : \Sigma^* \to \Sigma^*$.

The output corresponding to the input $x \in \Sigma^*$ is $f(x) \in \Sigma^*$.

Decision problems

Many problems can be formulated as Yes-No questions.

Such problems have the form:

$$P : \Sigma^* \to \{\text{Yes}, \text{No}\}$$

and are also called decision problems.

Decision problems vs. Languages

Let $L = P^{-1}(\text{Yes}) \subseteq \Sigma^*$
the set of the inputs answered with “Yes”.

Such a subset is usually called language.

Decision problems

Many problems can be formulated as Yes-No questions.

Such problems have the form:

$$P : \Sigma^* \to \{\text{Yes}, \text{No}\}$$

and are also called decision problems.

Example:

$x \in \mathbb{N} \mapsto w \in \{0, 1, \dots, 9\}^*$

$$P(x) := \begin{cases} \text{Yes} & \text{Prog(x) terminates} \\ \text{No} & \text{Prog(x) does not terminate} \end{cases}$$

$$L = P^{-1}(Yes) = \{x \mid x > 100\}$$

```
Prog($x$)
begin if $x > 100$ then return $x$
        else while true: continue
end
```

Decision problems vs. Languages

Let $L = P^{-1}(\text{Yes}) \subseteq \Sigma^*$
the set of the inputs answered with "Yes".

Such a subset is usually called language.

Central Question

Which functions are computable by an algorithm?

resp.

Which problems are decidable by an algorithm?

The motivation to study the decidability and undecidability of problems stems from the mathematician David Hilbert:

At the beginning of the 20th century, he formulated a research plan, (Hilbert's Programme) with the goal of developing a formalism which could allow to solve (algorithmically) all mathematical problems.

Central Question

Which functions are computable by an algorithm?

resp.

Which problems are decidable by an algorithm?

To clarify this question from a mathematical point of view, we must clarify what is an algorithm and what is a computer.

Central Question

Which functions are computable by an algorithm?

resp.

Which problems are decidable by an algorithm?

To clarify this question from a mathematical point of view, we must clarify what is an algorithm and what is a computer.

We need a mathematical model of computation

Central Question

Which functions are computable by an algorithm?

resp.

Which problems are decidable by an algorithm?

To clarify this question from a mathematical point of view, we must clarify what is an algorithm/computer.

We need a mathematical model of computation:
Turing machines

Alan Turing

Alan Turing

Alan Turing (1912 - 1954)

- Mathematician and logician; one of the founders of computer science
- 1936: Introduced "Turing machine" as a model of computability
- 1938: PhD (with Alonzo Church in Princeton)
- During the 2nd World War:
  Government Code and Cypher School (GCCS) Britain's codebreaking centre. For a time head of the section responsible for German naval cryptanalysis and devised a number of techniques for breaking German ciphers.
- After the war: National Physical Laboratory, Computing Laboratory, University of Manchester
- Contributions to AI ("Turing-Test")
- Tragical death

One of the most important awards in computer science: Turing Award.

Turing machines

A Turing machine is a device that manipulates symbols on a strip of tape according to a table of rules. It represents an algorithm or a program.

Turing machines

Alan Turing described a Turing machine (which he called “Logical Computing Machine”), as consisting of:

> “ ... an unlimited memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed.
>
> At any moment there is one symbol in the machine; it is called the scanned symbol.
>
> The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behaviour of the machine.
>
> However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings.”

Turing machines

Various definitions in the literature (all equally powerful)

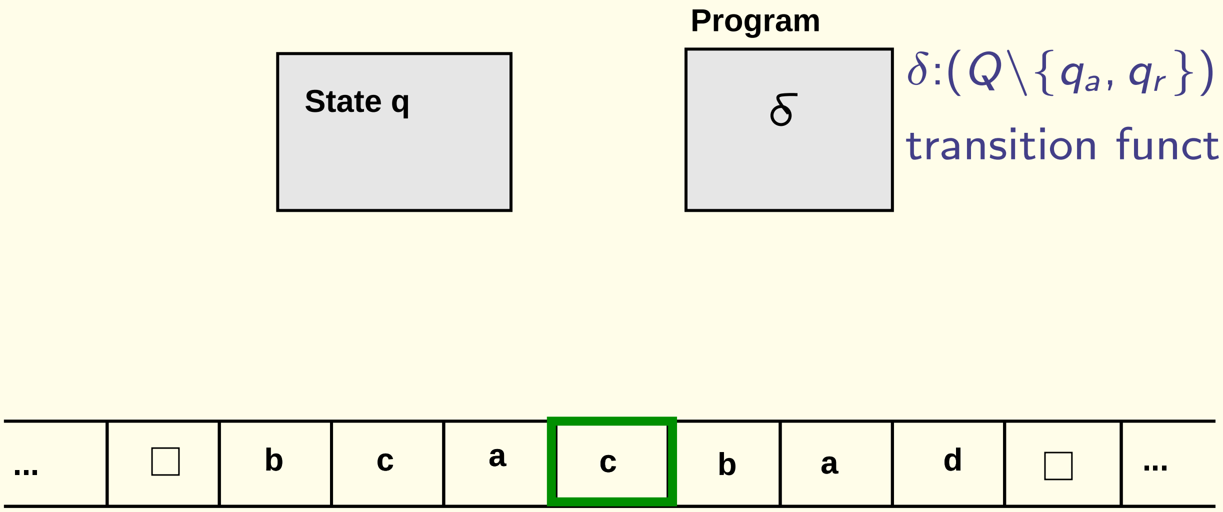
1. Definition given e.g. in “Theoretische Informatik”, J. Hromcovič

Deterministic Turing machines [Hromcovič]

Q: finite, non-empty set of states

$q_i \in Q$ initial state

$q_a \in Q$ final accepting state; $q_r \in Q$ final rejecting state



$\delta{:}(Q \setminus \{q_a, q_r\}) \times \Gamma \to Q \times \Gamma \times \{R, L, N$

transition function

read/write-head

Tape (unlimited both ways)

Σ: finite, non-empty input alphabet; $\square \in \Gamma \setminus \Sigma$, blank symbol

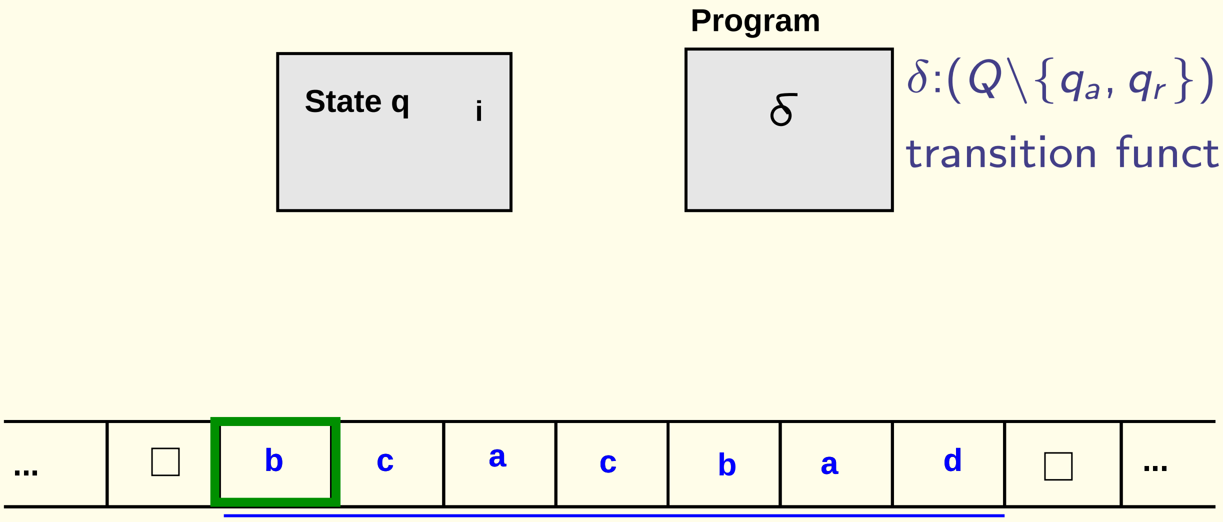
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

Turing machines: Input

Q: finite, non-empty set of states

$q_i \in Q$ initial state

$q_a \in Q$ final accepting state; $q_r \in Q$ final rejecting state



$\delta{:}(Q\backslash\{q_a, q_r\})\times\Gamma\rightarrow Q\times\Gamma\times\{R, L, N$

transition function

read/write-head

Tape (unlimited both ways)

Σ: finite, non-empty input alphabet; $\square \in \Gamma\backslash\Sigma$, blank symbol

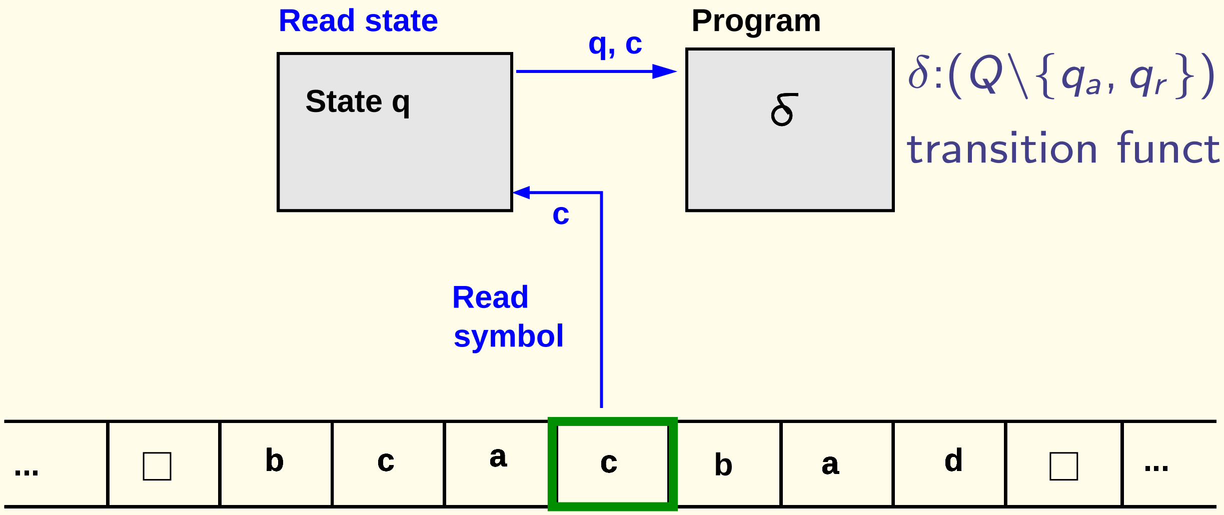
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

Turing machines: Computation step

Q: finite, non-empty set of states

$q_i \in Q$ initial state

$q_a \in Q$ final accepting state; $q_r \in Q$ final rejecting state



Read state

State q

q, c

Program

δ

$\delta{:}(Q\setminus\{q_a, q_r\})\times\Gamma\rightarrow Q\times\Gamma\times\{R, L, N$

transition function

c

Read symbol

...	□	b	c	a	c	b	a	d	□	...

read/write-head

Tape (unlimited both ways)

Σ: finite, non-empty input alphabet; $\square \in \Gamma\setminus\Sigma$, blank symbol

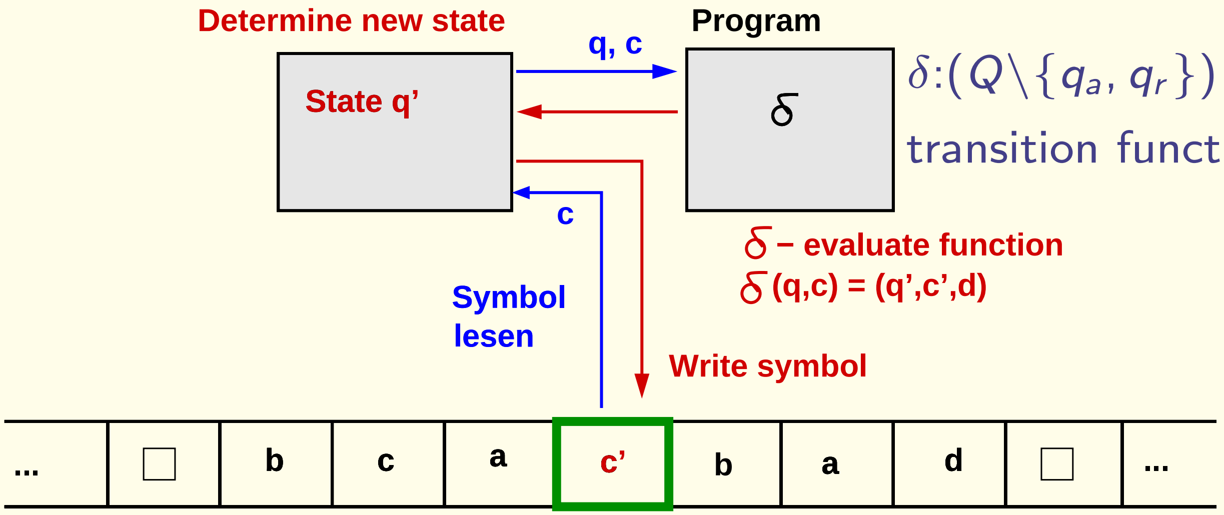
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

Turing machines: Computation step

Q: finite, non-empty set of states

$q_i \in Q$ initial state

$q_a \in Q$ final accepting state; $q_r \in Q$ final rejecting state



Determine new state

State q’

q, c

Program

δ

$\delta:(Q\backslash\{q_a,q_r\})\times\Gamma\rightarrow Q\times\Gamma\times\{R,L,N$

transition function

c

Symbol lesen

δ – evaluate function

δ(q,c) = (q’,c’,d)

Write symbol

... ☐ b c a c’ b a d ☐ ...

read/write-head

Tape (unlimited both ways)

Σ: finite, non-empty input alphabet; $\square \in \Gamma\backslash\Sigma$, blank symbol

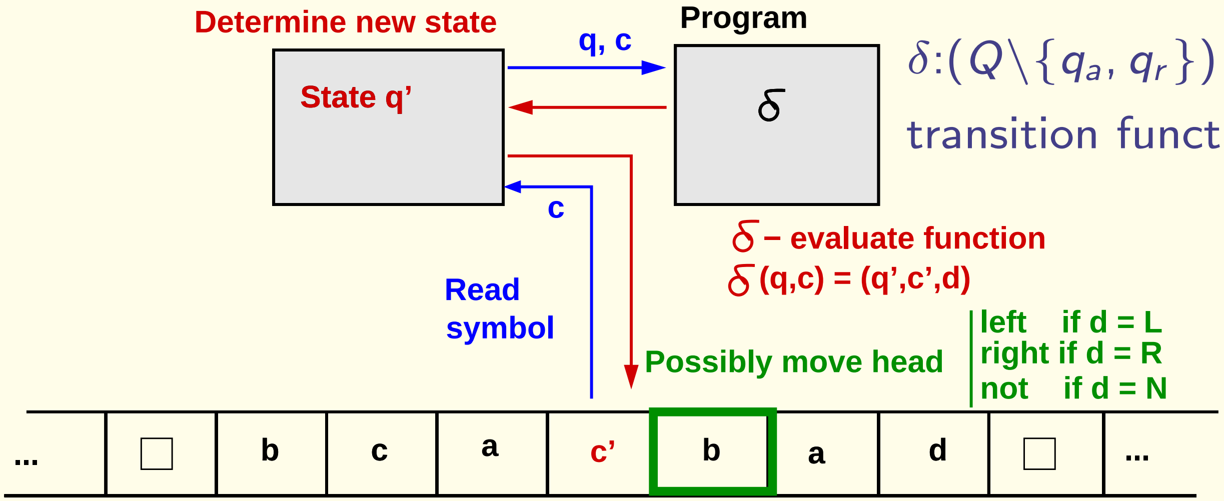
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

Turing machines: Computation step

Q: finite, non-empty set of states

$q_i \in Q$ initial state

$q_a \in Q$ final accepting state; $q_r \in Q$ final rejecting state



$\delta{:}(Q\setminus\{q_a, q_r\})\times\Gamma\rightarrow Q\times\Gamma\times\{R, L, N$

transition function

read/write-head

Tape (unlimited both ways)

Σ: finite, non-empty input alphabet; $\square \in \Gamma\setminus\Sigma$, blank symbol

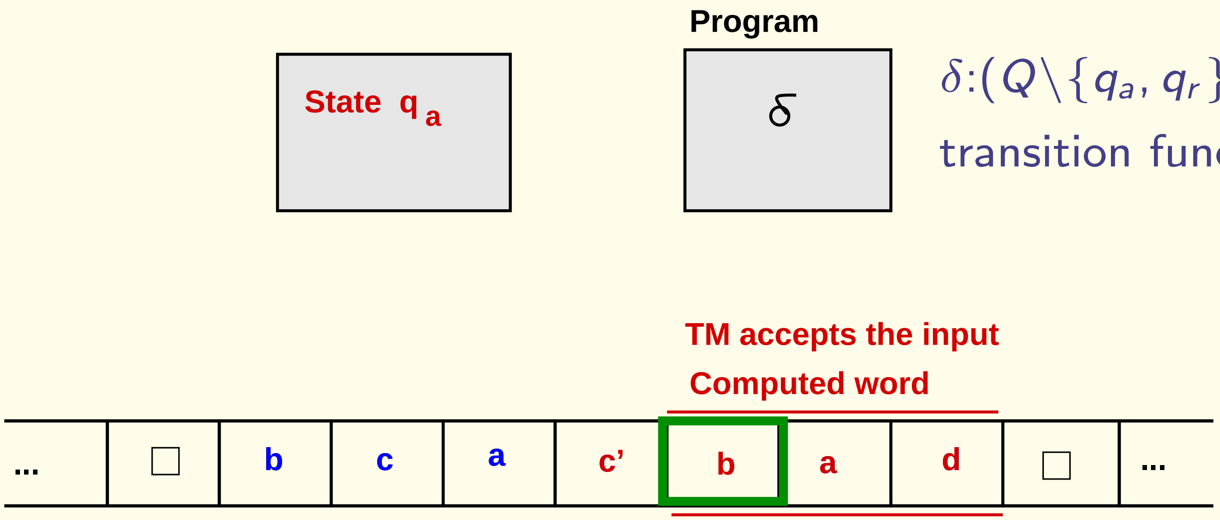
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

Turing maschines: End of the computation

Q: finite, non-empty set of states

$q_i \in Q$ initial state

$q_a \in Q$ final accepting state; $q_r \in Q$ final rejecting state



Program

State q_a

δ

δ:$(Q \setminus \{q_a, q_r\}) \times \Gamma \rightarrow Q \times \Gamma \times \{R, L,$

transition function

TM accepts the input

Computed word

...	□	b	c	a	c'	b	a	d	□	...

read/write-head

Tape (unlimited both ways)

Σ: finite, non-empty input alphabet; $\square \in \Gamma \setminus \Sigma$, blank symbol

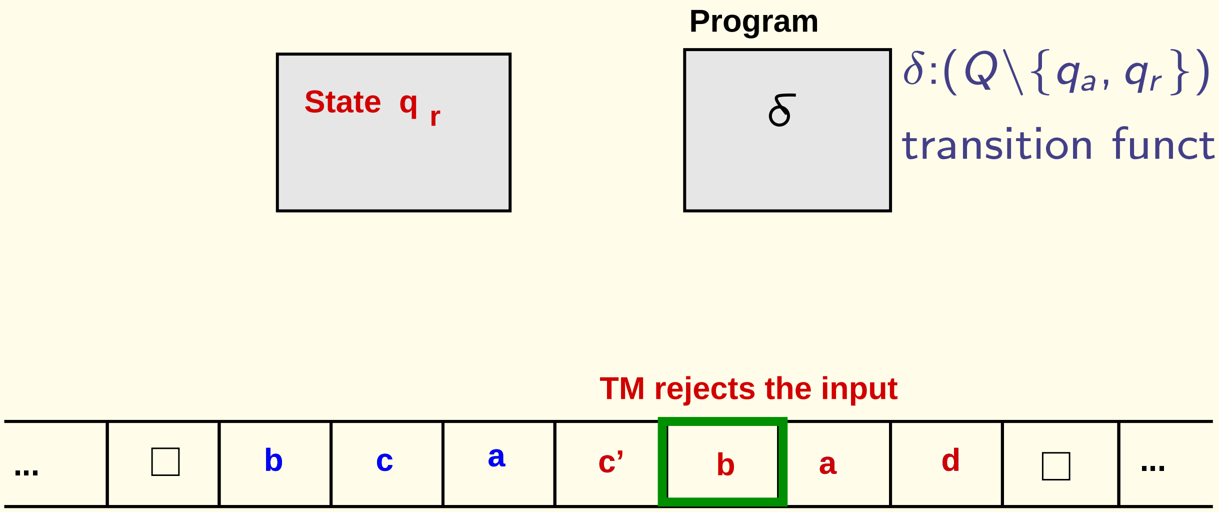
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

Turing maschines: End of the computation

Q: finite, non-empty set of states

$q_i \in Q$ initial state

$q_a \in Q$ final accepting state; $q_r \in Q$ final rejecting state



$\delta:(Q\setminus\{q_a, q_r\})\times\Gamma\rightarrow Q\times\Gamma\times\{R, L, N$

transition function

read/write-head

Tape (unlimited both ways)

Σ: finite, non-empty input alphabet; $\square \in \Gamma\setminus\Sigma$, blank symbol

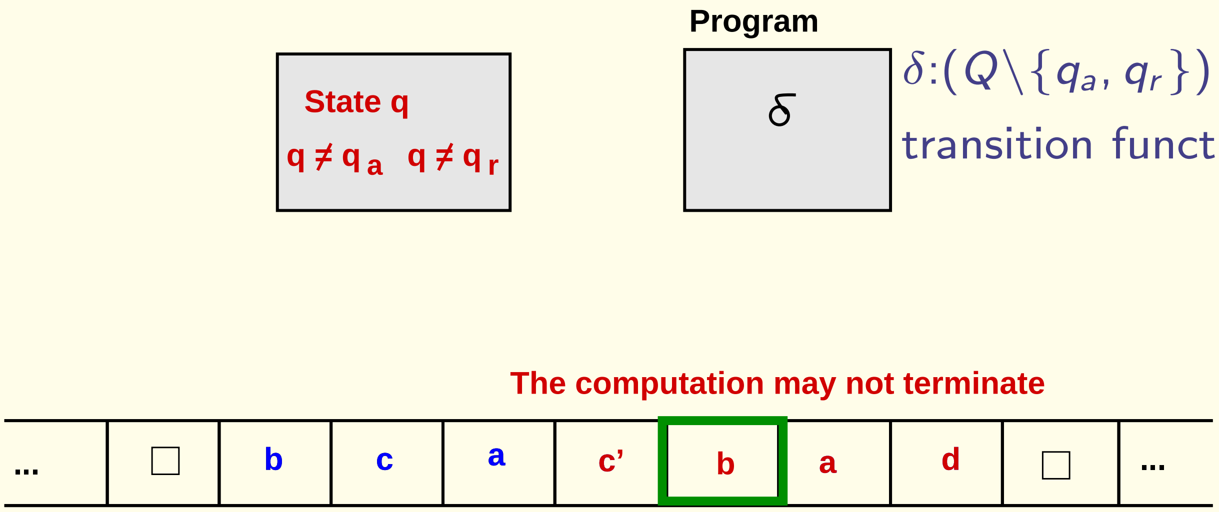
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

Turing maschines: End of the computation

Q: finite, non-empty set of states

$q_i \in Q$ initial state

$q_a \in Q$ final accepting state; $q_r \in Q$ final rejecting state



Program

State q

$q \neq q_a \quad q \neq q_r$

δ

$\delta : (Q \setminus \{q_a, q_r\}) \times \Gamma \rightarrow Q \times \Gamma \times \{R, L, N$

transition function

The computation may not terminate

read/write-head

Tape (unlimited both ways)

Σ: finite, non-empty input alphabet; $\square \in \Gamma \setminus \Sigma$, blank symbol

$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

Turing-Computability/Decidability

Definition

- A function $f : \Sigma^* \to \Sigma^*$ is Turing computable, if there exists a Turing machine, which terminates for all inputs and:

  $$\forall x, y \in \Sigma^* \quad f(x) = y \quad \text{iff} \quad M \text{ computes } y \text{ from input } x.$$

- A TM M accepts $w \in \Sigma^*$ if the computation of M on x terminates in state q_a.

Definition

- A language $L \subseteq \Sigma^*$ is Turing decidable, if there is a Turing machine, which terminates for all inputs and accepts the input w iff $w \in L$.
- A problem $P : \Sigma^* \to \{\text{Yes}, \text{No}\}$ is Turing decidable, if there exists a Turing machine, which terminates on all inputs and accepts the input w iff $w \in L = P^{-1}(\text{Yes})$.

Turing machines

Various definitions in the literature (all equally powerful)

1. Definition given e.g. in “Theoretische Informatik”, J. Hromcovič

2. Definition given in “Theoretische Informatik”, Katrin Erk & Lutz Priese

sketched on the next pages, will be discussed in detail in the next lecture.

(the definition given in “Grundlagen der TI”; will be used in this lecture)

Turing Machines [Erk, Priese]

Definition (Deterministic Turing Machine (DTM))

A deterministic Turing Machine (DTM) $\mathcal{M}$ is a tuple

$$\mathcal{M} = (K, \Sigma, \delta, s)$$

where:

- K is a finite set of states with $h \notin K$;
  (h is the halting state)
- Σ is an alphabet with $L, R \notin \Sigma, \# \in \Sigma$
- $\delta : K \times \Sigma \to (K \cup \{h\}) \times (\Sigma \cup \{L, R\})$ is a transition function
- $s \in K$ is an initial state

Number of states: $|K| - 1$ (initial state is not counted)

Turing Machines

How does a Turing Machine work?

Transition $\delta(q, a) = (q', x)$ means:

Depending on the:

- current state $q \in K$
- symbol $a \in \Sigma$ on which the read/write head is positioned

the following happens:

- a step to the left (if $x = L$)
- a step to the right (if $x = R$)
- the symbol a which currently stands below the read/write head is overwritten with symbol $b \in \Sigma$ (if $x = b \in \Sigma$)
- the state is changed to $q' \in K \cup \{h\}$.

Turing Machine: Accepted language

Definition

- A word w is accepted by a DTM $\mathcal{M}$ if $\mathcal{M}$ halts on input w (such that at the end, the head is positioned on the first blank on the right of w)
- A language $L \subseteq \Sigma^*$ is accepted by a DTM $\mathcal{M}$ iff the words from L (and no other words) are accepted by $\mathcal{M}$.

Attention:

For words which are not accepted, the DTM does not need to halt

(it is not allowed to halt, in fact)

Contents

- Recall: Turing machines and Turing computability
- **Register machines (LOOP, WHILE, GOTO)**
- Recursive functions
- The Church-Turing Thesis
- Computability and (Un-)decidability
- Complexity

Register Machines

In comparison to Turing machines:

- equally powerful fundament for computability theory
- **Advantage:** Programs are easier to understand

Register Machines

In comparison to Turing machines:

- equally powerful fundament for computability theory
- **Advantage:** Programs are easier to understand

similar to ...

the imperative kernel of programming languages

pseudo-code

Register Machines

Computation of a mod b (pseudocode)

```
$r := a$;
while $r \geq b$ do
    $r := r - b$;
end;
return r
```

Register Machines

Definition: Questions

Which instructions (if, while, goto?)

Which data types? (integers? strings?)

Which data structures? (arrays?)

Which atomic instructions?

Which Input/Output?

Register Machines

Definition: Questions

Which instructions (if, while, goto?)

Which data types? (integers? strings?)

Which data structures? (arrays?)

Which atomic instructions?

Which Input/Output?

Here: LOOP-programs; WHILE-programs; GOTO-programs

Links between LOOP, WHILE, GOTO and Turing machines.

Contents

- Recall: Turing machines and Turing computability
- Register machines (LOOP, WHILE, GOTO)
- **Recursive functions**
- The Church-Turing Thesis
- Computability and (Un-)decidability
- Complexity

Recursive functions

Motivation

Functions as model of computation (without an underlying machine model)

Recursive functions

Motivation

Functions as model of computation (without an underlying machine model)

Idea

- Simple ("atomic") functions are computable
- "Combinations" of computable functions are computable

(We consider functions $f : \mathbb{N}^k \rightarrow \mathbb{N}$, $k \geq 0$)

Recursive functions

Motivation

Functions as model of computation (without an underlying machine model)

Idea

- Simple (“atomic”) functions are computable
- “Combinations” of computable functions are computable

(We consider functions $f : \mathbb{N}^k \to \mathbb{N}$, $k \geq 0$)

Questions

- Which are the atomic functions?
- Which combinations are possible?

Recursive functions

Atomic functions:

Constant null; successor; projection (choice)

Recursive functions

Atomic functions:

Constant null; successor; projection (choice)

Composition

function composition

Recursive functions

Atomic functions:

Constant null; successor; projection (choice)

Composition

function composition

Recursion

primitive recursion $\mapsto$ primitive recursive functions

$+$ μ-operator $\mapsto$ recursive functions

Contents

- Recapitulation: Turing machines and Turing computability
- Recursive functions
- Register machines (LOOP, WHILE, GOTO)
- **The Church-Turing Thesis**
- Computability and (Un-)decidability
- Complexity

The Church-Turing Thesis

Informally: The functions which are intuitively computable are exactly the functions which are Turing computable.

The Church-Turing Thesis

Informally: The functions which are intuitively computable are exactly the functions which are Turing computable.

Instances of this thesis: all known models of computation

- Turing machines
- Recursive functions
- λ-functions
- all known programming languages (imperative, functional, logic)

provide the same notion of computability

Alonzo Church

Alonzo Church (1903-1995)

- studied in Princeton; PhD in Princeton
- Postdoc in Göttingen
- Professor: Princeton and UCLA
- Layed the foundations of theoretical computer science (e.g. introduced the λ-calculus)
- One of the most important computer scientists

Alonzo Church

PhD Students:

- **Peter Andrews**: automated reasoning
- **Martin Davis**: Davis-Putnam procedure (automated reasoning)
- **Leon Henkin**: (Standard) proof of completeness of predicate logic
- **Stephen Kleene**: Regular expressions
- **Dana Scott**: Denotational Semantics, Automata theory
- **Raymond Smullyan**: Tableau calculi
- **Alan Turing**: Turing machines, Undecidability of the halting problem
- ... and many others

Contents

- Recapitulation: Turing machines and Turing computability
- Recursive functions
- Register machines (LOOP, WHILE, GOTO)
- The Church-Turing Thesis
- **Computability and (Un-)decidability**
- Complexity

Computability and (Un-)decidability

Known undecidable problems (Theoretical Computer Science I)

- The halting problem for Turing machines

Computability and (Un-)decidability

Known undecidable problems (Theoretical Computer Science I)

- The halting problem for Turing machines

Consequences:

- All problems about programs (TM) which are non-trivial (in a certain sense) are undecidable (Theorem of Rice)
- Identify undecidable problems outside the world of Turing machines
  - Validity/Satisfiability in First-Order Logic
  - The Post Correspondence Problem

  $\mapsto$ undecidability results in formal languages
- These results show that Hilbert's Program is not realisable.

Computability and (Un-)decidability

The Theorem of Rice (informal)

For each non-trivial property P of (partial) functions:

It is undecidable, whether the function computed by a Turing machine has property P.

Computability and (Un-)decidability

The Theorem of Rice (informal)

For each non-trivial property P of (partial) functions:

It is undecidable, whether the function computed by a Turing machine has property P.

Variant 2

For each non-trivial property P of languages of type 0:

It is undecidable, whether the language accepted by a Turing machine has property P.

Computability and (Un-)decidability

The Theorem of Rice (informal)

For each non-trivial property P of (partial) functions:

It is undecidable, whether the function computed by a Turing machine has property P.

Generalization:

The same holds for other computability models:

- algorithms
- Java programs
- λ expressions
- recursive functions
- etc.

Henry Gordon Rice

Henry Gordon Rice (1920-2003)

best known as the author of Rice's theorem, which he proved in his doctoral dissertation of 1951 at Syracuse University.

Contents

- Recapitulation: Turing machines and Turing computability
- Register machines (LOOP, WHILE, GOTO)
- Recursive functions
- The Church-Turing Thesis
- Computability and (Un-)decidability
- **Complexity**

Complexity

- Complexity classes; Relationships between complexity classes (P, NP, PSPACE)
- How to show that a given problem is in a certain class?
  Reduction to known problems (e.g. SAT)
- Complete and hard problems
- Closure properties for complexity classes
- Examples

Stephen Cook

Stephen Arthur Cook (born 1939)

- Major contributions to complexity theory.
  Considered one of the forefathers of computational complexity theory.
- 1971 'The Complexity of Theorem Proving Procedures'
  Formalized the notions of polynomial-time reduction and NP-completeness, and proved the existence of an NP-complete problem by showing that the Boolean satisfiability problem (SAT) is NP-complete.
- Currently University Professor at the University of Toronto
- 1982: Turing award for his contributions to complexity theory.

Contents

- Recapitulation: Turing machines and Turing computability
- Register machines (LOOP, WHILE, GOTO)
- Recursive functions
- The Church-Turing Thesis
- Computability and (Un-)decidability
- Complexity