Vertiefung Theoretische Informatik

Advanced Topics in Theoretical Computer Science

Viorica Sofronie-Stokkermans

Universität Koblenz-Landau

e-mail: sofronie@uni-koblenz.de

Wintersemester 2012-2013

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 K. Erk and L. Priese and on slides by Christoph Kreitz)

Many thanks!

Literatur

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.

Organisation

Lecture: Viorica Sofronie-Stokkermans

`sofronie@uni-koblenz.de`

Sprechstunde: to be announced

Organisation

Lecture: Viorica Sofronie-Stokkermans

`sofronie@uni-koblenz.de`

Sprechstunde: to be announced

Exercise: Markus Bender

`bender@uni-koblenz.de`

Introduction

- Details about the lecture
- Motivation
- Contents

Lecture

- Webseite: `www.uni-koblenz.de/∼sofronie/lecture-TI2-2012`
- Time and place:
  - **Lecture** Now: Tuesdays, 14:00-16:00
  - **Exercises** Tuesdays: 16:00-18:00

Lecture

- Webseite: `www.uni-koblenz.de/~sofronie/lecture-TI2-2012`
- Time and place:
  - **Lecture** Now: Tuesdays, 14:00-16:00
  - **Exercises** Tuesdays: 16:00-18:00

Conflict with the AI lecture

New time/place

Lecture

Exercises:

- Will appear weekly on the website
- Will be discussed in the next exercise session
- You can solve them (possibly also in groups of up to 3 students) and hand in the solutions

Lecture

Exercises:

- Will appear weekly on the website
- Will be discussed in the next exercise session
- You can solve them (possibly also in groups of up to 3 students) and hand in the solutions ... but you do not *have* to hand them in

Lecture

Exams:

- **Klausur:** end of the lecture time.
  Criterion for passing: 50% of the total number of points
- **Nachklausur:** end of the semester (from all the material)
  Criterion for passing: 50% of the total number of points in the Nachklausur.

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!

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: e.g. Büchi Automata

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: e.g. Büchi Automata

Computability/Turing Machines: Idea

What is a problem?

Informally:

for certain inputs

certain outputs must be produced.

More precise definition:

Input: Word over alphabet Σ

Output: Word 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:

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

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:

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

and are also called decision problems.

Example:

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

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

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

```
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:

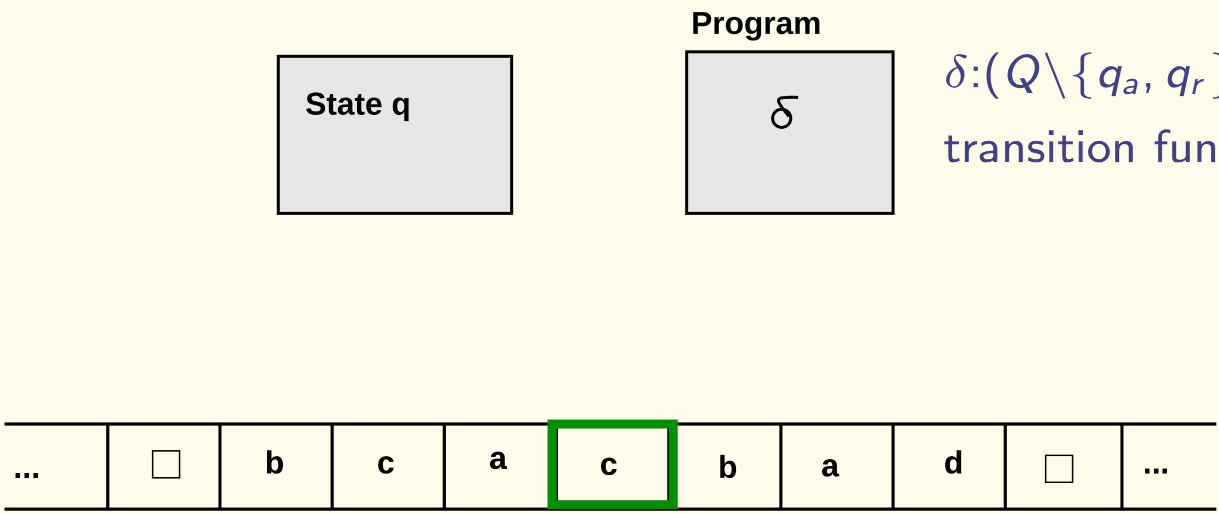
> " ... an unlimited memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed.
>
> At any moment there is one symbol in the machine; it is called the scanned symbol.
>
> The machine can alter the scanned symbol and its behavior is in 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."

Deterministic Turing machines

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; $\Box \in \Gamma \setminus \Sigma$, blank symbol

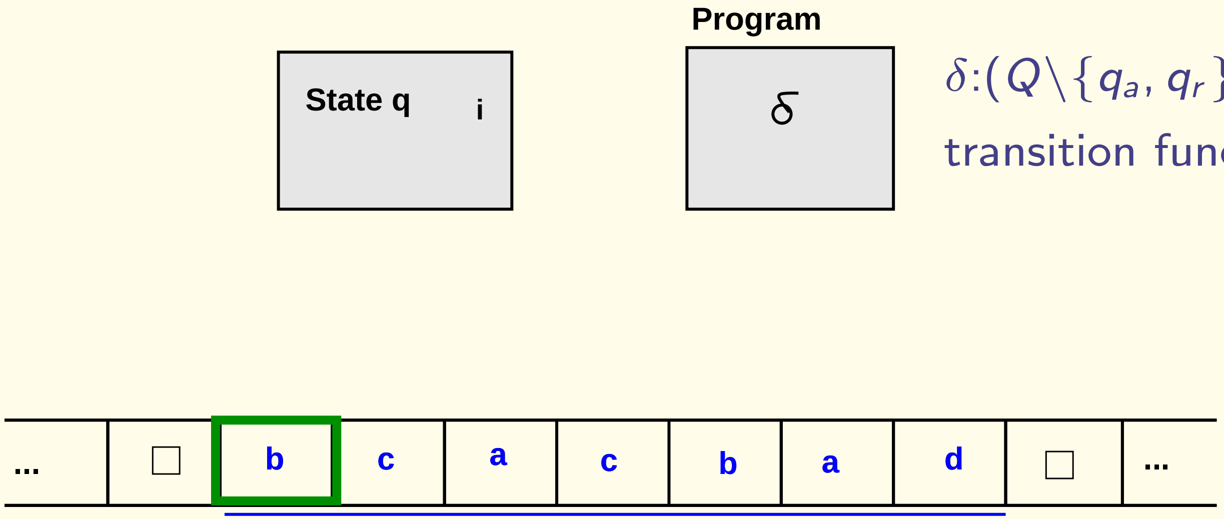
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

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



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

transition function

read/write-head

Tape (unlimited both ways)

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

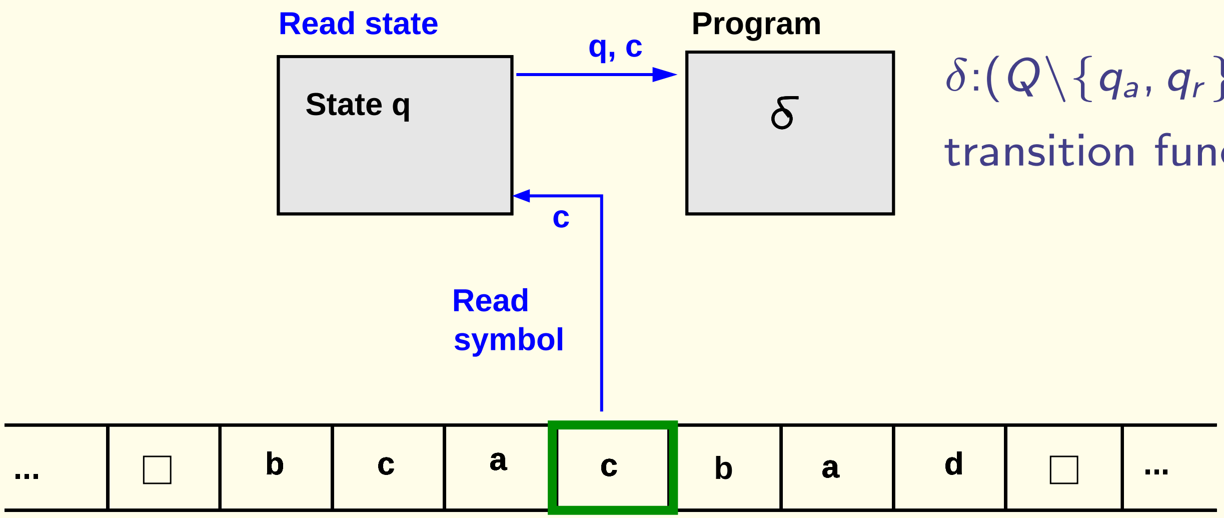
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

Turing machines: 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



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

transition function

read/write-head

Tape (unlimited both ways)

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

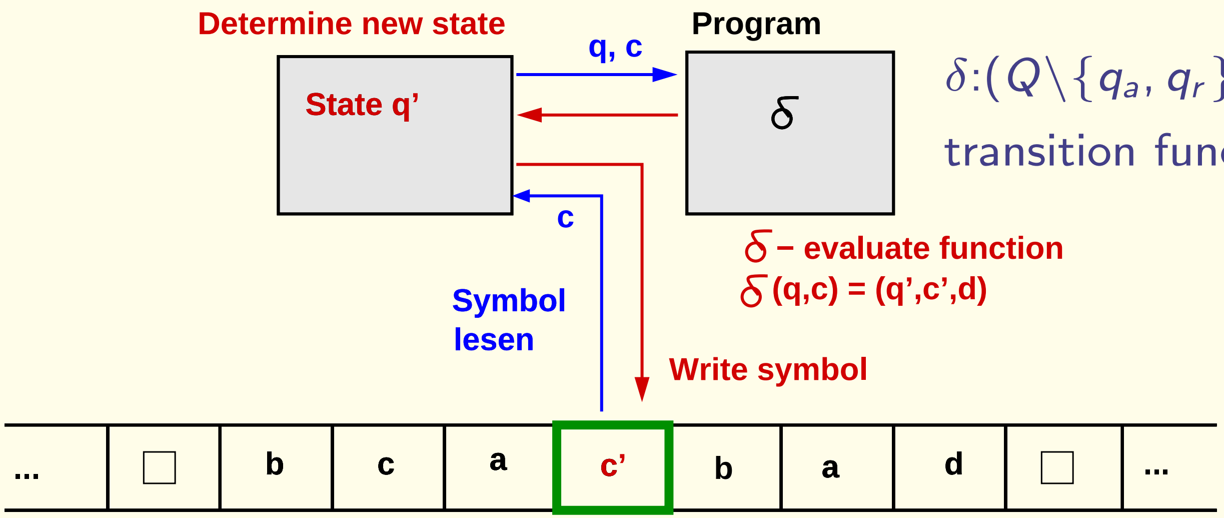
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

Turing machines: 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



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

transition function

read/write-head

Tape (unlimited both ways)

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

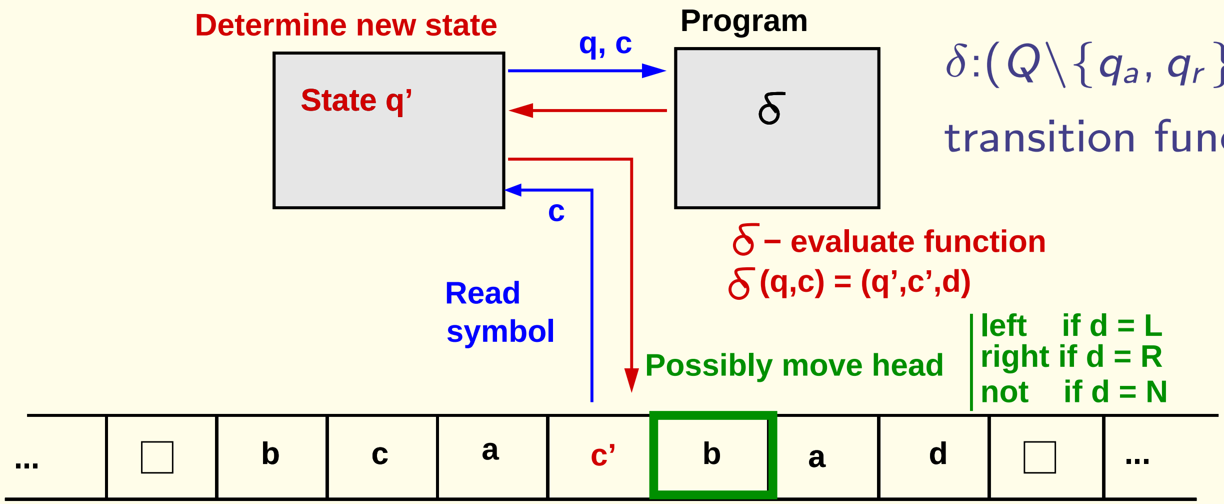
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

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



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

transition function

read/write-head

Tape (unlimited both ways)

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

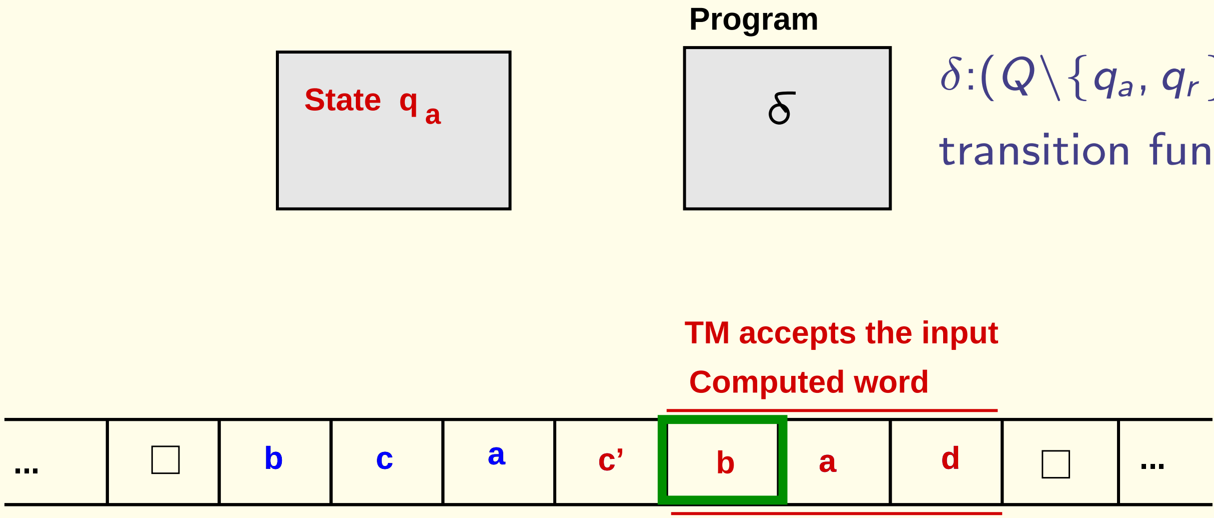
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

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



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

transition function

read/write-head

Tape (unlimited both ways)

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

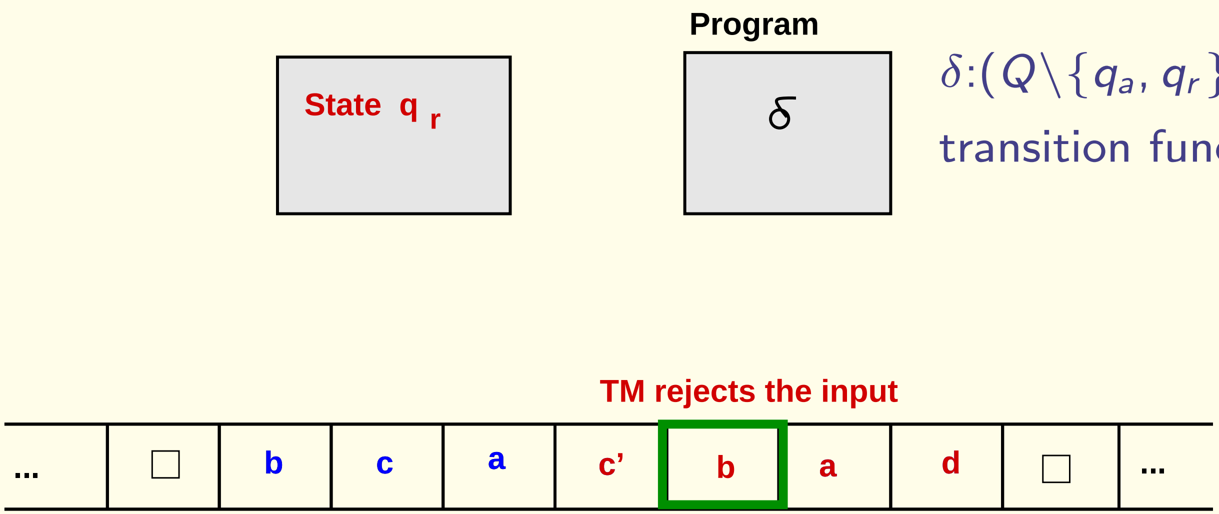
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

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



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

transition function

read/write-head

Tape (unlimited both ways)

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

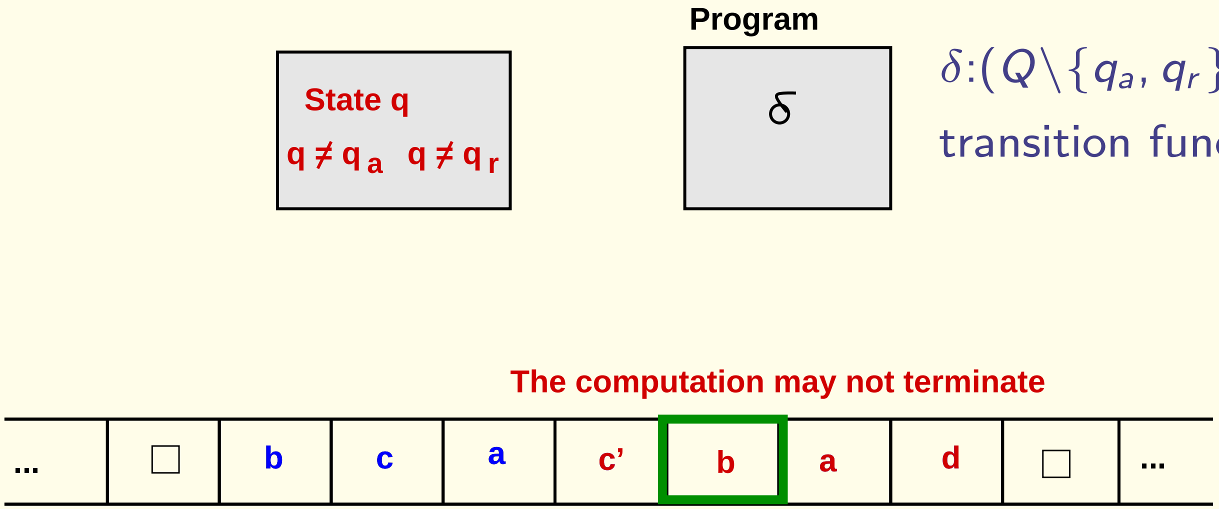
$\Gamma \supset \Sigma$, finite, non-empty tape alphabet

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-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 terminated 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 $w \in L = P^{-1}(\text{Yes})$.

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: e.g. Büchi Automata

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

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 $f : \mathbb{N}^k \to \mathbb{N}$, $k \geq 0$)

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

Contents

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
- Other computation models: e.g. Büchi Automata

Computability and (Un-)decidability

Known undecidable problems (Theoretical Computer Science I)

- The halting problem for Turing machines
- Equivalence problem

Computability and (Un-)decidability

Known undecidable problems (Theoretical Computer Science I)

- The halting problem for Turing machines
- Equivalence problem

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
- 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.

Generalization:

The same holds for other computability models:

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

Henry Gordon Rice

Henry Gordon Rice (born 1920)

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**
- Other computation models: e.g. Büchi Automata

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
- **Other computation models: e.g. Büchi Automata**

Büchi Automata

ω-Automata

An ω-automaton (or stream automaton) is a variation of finite automaton that runs on infinite, rather than finite, strings as input.

Since ω-automata do not stop, they have a variety of acceptance conditions rather than simply a set of accepting states.

Classes of ω-automata include the Büchi automata, Rabin automata, Streett automata, parity automata and Muller automata, each deterministic or non-deterministic (differ only in terms of acceptance condition).

They all recognize precisely the regular ω-languages except for the deterministic Büchi automata, which is strictly weaker than all the others.

Büchi Automata

Büchi automaton

Accepts an infinite input sequence iff there exists a run of the automaton that visits (at least) one of the final states infinitely often.

Büchi automata are often used in Model checking as an automata-theoretic version of a formula in linear temporal logic.

Model checking of finite state systems can often be translated into various operations on Büchi automata.

Other models of computation

- The λ-calculus

Presented in the lecture "Programming language theory"

Brief idea in what follows

The λ-calculus

Lambda calculus (also written as λ-calculus) is a formal system in mathematical logic for expressing computation by way of variable binding and substitution.

It was first formulated by Alonzo Church as a way to formalize mathematics through the notion of functions, in contrast to the field of set theory.

Example

The identity function $id(x) = x$: input x; returns x

$sqsum(x, y) = x \cdot x + y \cdot y$: input (x, y); returns $x^2 + y^2$.

Example

The identity function $id(x) = x$: input x; returns x

$sqsum(x, y) = x \cdot x + y \cdot y$: input (x, y); returns $x^2 + y^2$.

Observations:

1. functions need not be explicitly named.

   $sqsum(x, y) = x{\cdot}x{+}y{\cdot}y$ can be rewritten as $(x, y) \mapsto x{\cdot}x{+}y{\cdot}y$

   $id(x) = x$ can be rewritten as $x \mapsto x$

2. The specific choice of name for a function's arguments is irrelevant.

   $x \mapsto x$ and $y \mapsto y$ express the same function: the identity.

3. Any function that requires two inputs, for instance *sqsum* can be reworked into an equivalent function that accepts a single input, and as output returns another function, that in turn accepts a single input.

   $x \mapsto (y \mapsto x{\cdot}x{+}y{\cdot}y)$

   (currying; can be generalized to functions with arbitrary number of arguments)

Example

The identity function $id(x) = x$: input x; returns x

$sqsum(x, y) = x \cdot x + y \cdot y$: input (x, y); returns $x^2 + y^2$.

Observations:

1. functions need not be explicitly named.

   $sqsum(x, y) = x{\cdot}x{+}y{\cdot}y$ can be rewritten as $\lambda x, y.(x{\cdot}x{+}y{\cdot}y)$

   $id(x) = x$ can be rewritten as $\lambda x.x$

2. The specific choice of name for a function's arguments is irrelevant.

   $\lambda x.x$ and $\lambda y.y$ express the same function: the identity.

3. Any function that requires two inputs, for instance *sqsum* can be reworked into an equivalent function that accepts a single input, and as output returns another function, that in turn accepts a single input.

   $\lambda x, y.(x{\cdot}x{+}y{\cdot}y)$ is the same as
   $\lambda x.\lambda y.(x{\cdot}x{+}y{\cdot}y)$

   (currying; can be generalized to functions with arbitrary number of arguments)

The λ-calculus

Lambda calculus has played an important role in the development of the theory of programming languages. The most prominent counterparts to lambda calculus in computer science are functional programming languages, which essentially implement the calculus (augmented with some constants and datatypes).