Theoretical Computer Science

Pushdown Automata & Turing Machines

Technische Hochschule Rosenheim
Sommer 2022
Prof. Dr. Jochen Schmidt

Overview

- Pushdown automata
- Turing Machines
- Linear bounded automata

Pushdown Automata

Why Do We Need an Extension to Finite Automata?

Technische Hochschule **Rosenheim**

- A DFA/NFA can only recognize regular languages
  - formed by string concatenation, set union (∪) and Kleene closure (*)
- A DFA/NFA does not have memory

- DFA/NFA cannot test strings with arbitrarily deep nested brackets for correctness
  - like parentheses in arithmetical expressions like x = (((a + b) * c + (c + d) * (a + c))) * d;
  - or nested block structures in C or Java with braces {…{…}…{…}…}

⟶ Extension of NFA by adding memory in form of a stack (*Kellerspeicher*)

- a stack has a bottom and is unlimited in the other direction
- only the element at the top can be accessed directly: push/pop operations
- Pushdown Automaton (PDA, *Kellerautomat*)

Nondeterministic Pushdown Automaton (PDA)

Extension of NFA by a stack and

- a finite stack alphabet Γ
- an initial stack symbol # marking the bottom of the stack
- extension of the transition mapping
  - a transition depends on the current input symbol as well as the symbol at the top of the stack
  - in each transition
    - the top stack symbol is removed (pop)
    - none, one or multiple symbols can be written to the top of the stack (push)

- This general definition is nondeterministic – as with finite automata, we can restrict it to deterministic transitions and get a Deterministic PDA or DPDA.

PDA – Recognized Language

- Words can be accepted using
  - end states (same as for finite automata), independent of the content of the stack
  - empty stack – no end states, a word is accepted if the stack is completely empty after the input sequence has been processed (including the initial stack symbol #)
- These two options are equivalent for nondeterministic PDAs
  - They are different for DPDAs: Accepting words by end states is more powerful
- Nondeterministic PDAs recognize the so-called context-free languages
  - these are a superset of the regular languages

Example: Block Structures/Nested Braces

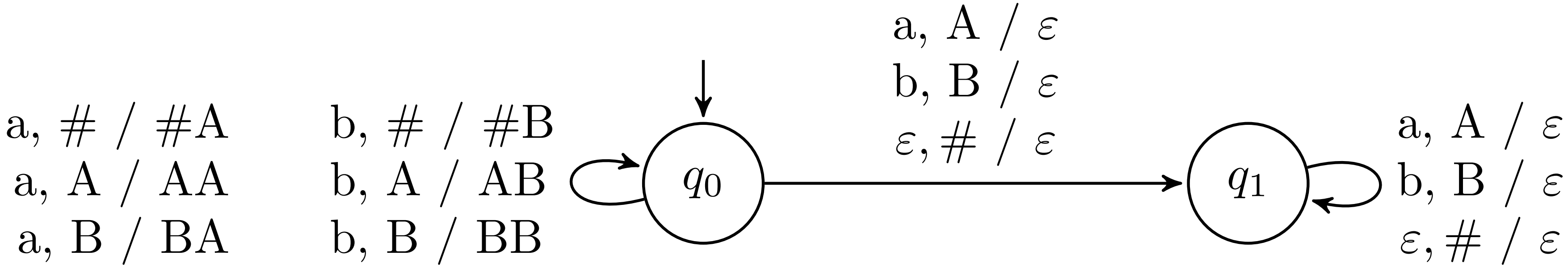
- Correct nesting: { { } { } }
- Incorrect nesting: { } } { } {   – just counting opening/closing braces will not work!
- Input alphabet $\Sigma = \{\{, \}\}$, stack alphabet $\Gamma = \{\#,\ \{\ \}$
- Accept words by empty stack
- How the PDA works:

1. Input symbol {: push it on top of stack.
2. Input symbol }:
   a) If the stack is in initial state (# on top): Error! The nesting is incorrect.
   b) If the stack is not in initial state: Pop top { from stack; each } input removes one {.
3. After all input symbols have been processed:
   a) If the stack is in initial state: The block structure is correct. Pop #, the stack is now empty, the word accepted.
   b) If the stack is not in initial state: Error! The block structure is incorrect.

This PDA is deterministic and has only a single state – no point in drawing a state diagram

Example: Mirroring (Palindromes)

- Input alphabet $\Sigma = \{a, b\}$, stack alphabet $\Gamma = \{A, B, \#\}$
- Recognized language: palindromes (*Palindrome*): $L = \{x_1 x_2 \ldots x_n x_n \ldots x_2 x_1 \mid x_i \in \Sigma\}$
- Accept words by empty stack
- Transition notation: a, # / #A means:
  - Do the transition if a was read as an input symbol
  - and # is on top of the stack (and will be popped).
  - Push #A on top of the stack.
  - (so, this example will actually leave # on top and in addition push A)



Example: Mirroring – Some Configurations

Some selected possible (not complete!) configurations when processing the input string aabbaa

Input	Configuration of PDA
ε	$(q_0, aabbaa, \#) \vdash (q_1, aabbaa, \varepsilon)$
a	$(q_0, aabbaa, \#) \vdash (q_0, abbaa, A\#)$
a	$(q_0, abbaa, A\#) \vdash (q_0, bbaa, AA\#)$; $(q_0, abbaa, A\#) \vdash (q_1, bbaa, \#) \vdash (q_1, bbaa, \varepsilon)$
b	$(q_0, bbaa, AA\#) \vdash (q_0, baa, BAA\#)$
b	$(q_0, baa, BAA\#) \vdash (q_0, aa, BBAA\#)$; $(q_0, baa, BAA\#) \vdash (q_1, aa, AA\#)$
a	$(q_0, aa, BBAA\#) \vdash (q_0, a, ABBAA\#)$; $(q_1, aa, AA\#) \vdash (q_1, a, A\#)$
a	$(q_0, a, ABBAA\#) \vdash (q_0, \varepsilon, AABBAA\#)$; $(q_0, a, ABBAA\#) \vdash (q_1, \varepsilon, BBAA\#)$; $(q_1, a, A\#) \vdash (q_1, \varepsilon, \#) \vdash (q_1, \varepsilon, \varepsilon)$

This is the only path that leads to accepting the input. Due to nondeterminism the PDA will automatically follow this path and not try the others.

Example: Mirroring – Remarks

- The PDA for $L = \{x_1 x_2 \dots x_n x_n \dots x_2 x_1 \mid x_i \in \Sigma\}$ is nondeterministic.
- There exists no deterministic PDA (DPDA) that recognizes this language.
- The nondeterministic behavior is necessary to “guess” the middle of the word
- Only by marking the middle of the word we can construct a DPDA, e.g.:
  $L = \left\{x_1 x_2 \dots x_n 8 x_n \dots x_2 x_1 \mid x_i \in \{\mathrm{a}, \mathrm{b}\}\right\}, \Sigma = \{\mathrm{a}, \mathrm{b}, 8\}$

PDAs – Notes

Technische Hochschule Rosenheim

- Unlike DFA/NFA, PDA and DPDA are not equivalent: PDAs are more powerful than DPDAs

- DPDAs recognize only a proper subset of context-free languages:
  the deterministic context-free languages
  - these are equivalent to the LR(k) languages (k > 0) and play an important role in compiler construction for syntax analysis

- What happens if we add more stacks?
  - A PDA with 2 stacks is computationally more powerful than a PDA with one stack
  - Adding yet more stacks may be more convenient, but does not increase computational power any further.
  - With 2 stacks a PDA is equivalent to a Turing Machine, the most powerful concept we know of.

Turing Machines

Turing Machines – Overview

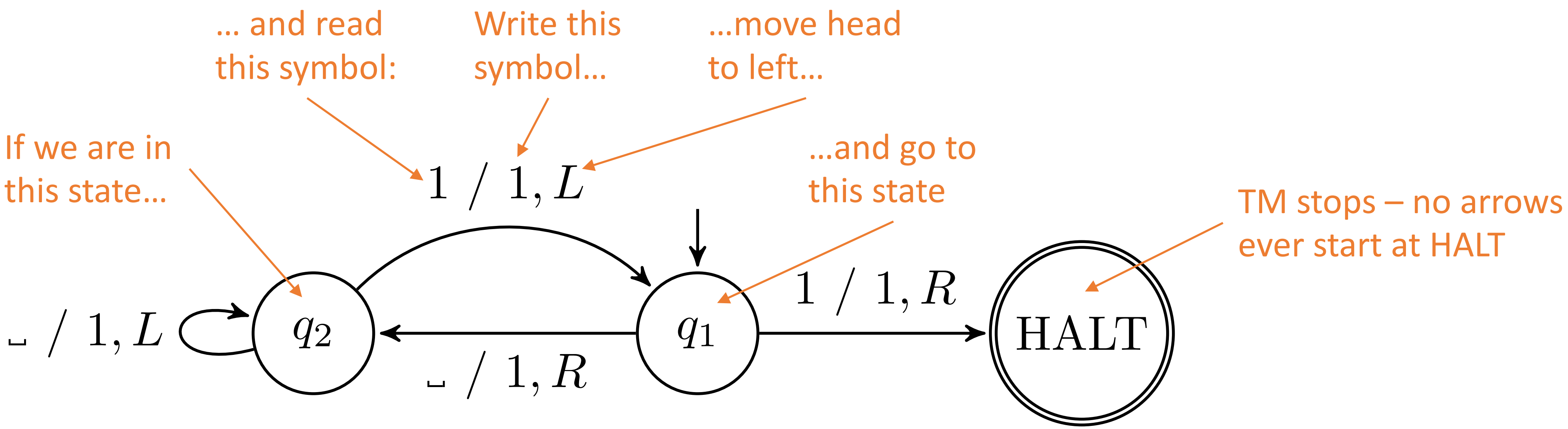
- Finite automata and pushdown automata have obvious restrictions
  - DFA/NFA: no memory at all, only states
  - PDA: memory, where access is restricted by stack-principle (push/pop)
    - can recognize languages like $L = \{a^n b^n\}$, but not $L = \{a^n b^n c^n\}$
- Turing Machine: Use memory tape (*Band*) where we can move left & right to read/write
  - we will now read our input from this tape
- Developed by Alan Turing (1912 – 1954) in the 1930s
  - the ACM Turing Award is named after him (the "Nobel Prize" of computer science)
- Anything a computer can do, a Turing Machine can do – and vice versa
  - it provides a very simple model of a universal computer and is therefore often used in theoretical studies
  - all other known concepts for formulating algorithms or describing abstract computer models can be shown to be equivalent to Turing Machines

Definition

A (deterministic) Turing machine (TM) consists of

- an infinite memory tape for input and output (*Schreib-/Lese-Band*), divided into cells,
- a read/write head that can move along the tape by single steps to left (L) and right (R),
- a finite input alphabet Σ,
- a finite tape alphabet Γ
  - Γ includes all input symbols and possibly additional ones, in particular the blank (space ⊔) with which the band is filled at the beginning
- a finite set of states Q with one initial state and at least one end state (HALT state).
- a state transition function $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$

TM as State Diagrams



Additionally required:

- Initialization of tape (typically: input string, remaining cells initialized with blank)
- Initial position of head

Transitions as Instruction Tables

- Transition function of a TM is typically described by a finite number of instructions
- Example of structure (for tape alphabet limited to $\{\sqcup, 1\}$):

symbol read from
current tape cell

direction of movement
for r/w head ($d_l \in \{R, L\}$)

instruction index $i \in \mathbb{N}$

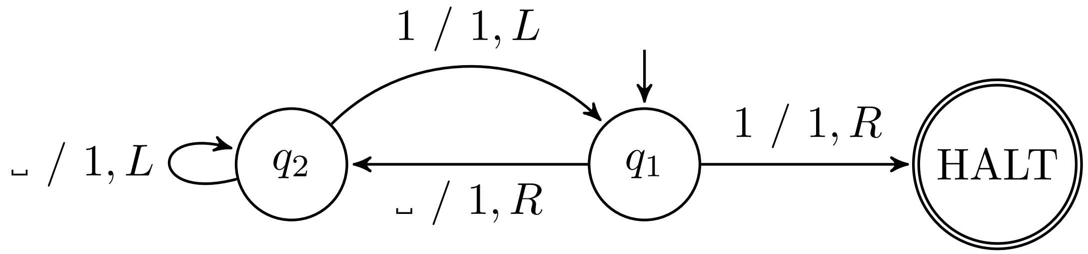
$$i\begin{cases} \sqcup & \gamma_1 & d_1 & j \\ 1 & \gamma_2 & d_2 & k \end{cases}$$

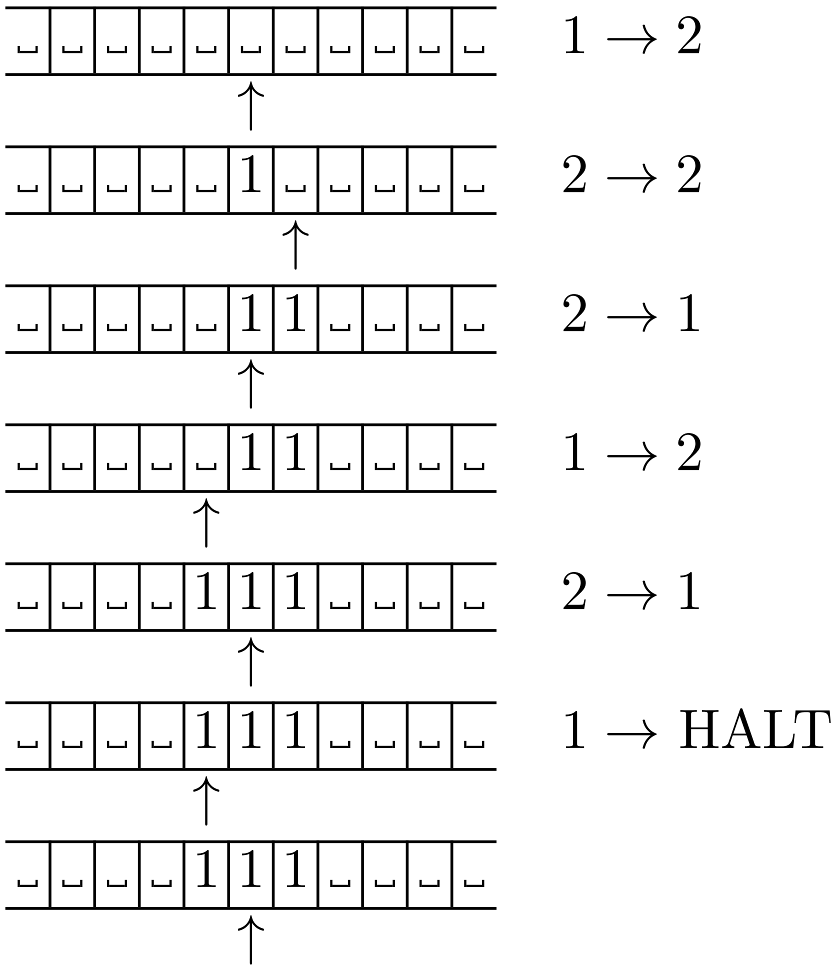
index of next instruction
or 0 for HALT

symbol to write to
current tape cell ($\gamma_l \in \Gamma$)

Example: Write Three Ones

TM that writes three ones on tape initialized with blanks:





TM written using instructions:

$$1\begin{cases} \textvisiblespace\ 1\ R\ \ 2 \\ 1\ 1\ R\ \mathrm{HALT} \end{cases} \qquad 2\begin{cases} \textvisiblespace\ 1\ L\ 2 \\ 1\ 1\ L\ 1 \end{cases}$$

TM: Configuration & Recognized Language

- Configuration = current symbols on tape + current state + current head position
  - Initial/Start configuration: configuration before processing is started
  - End/Halt configuration: configuration when TM has stopped

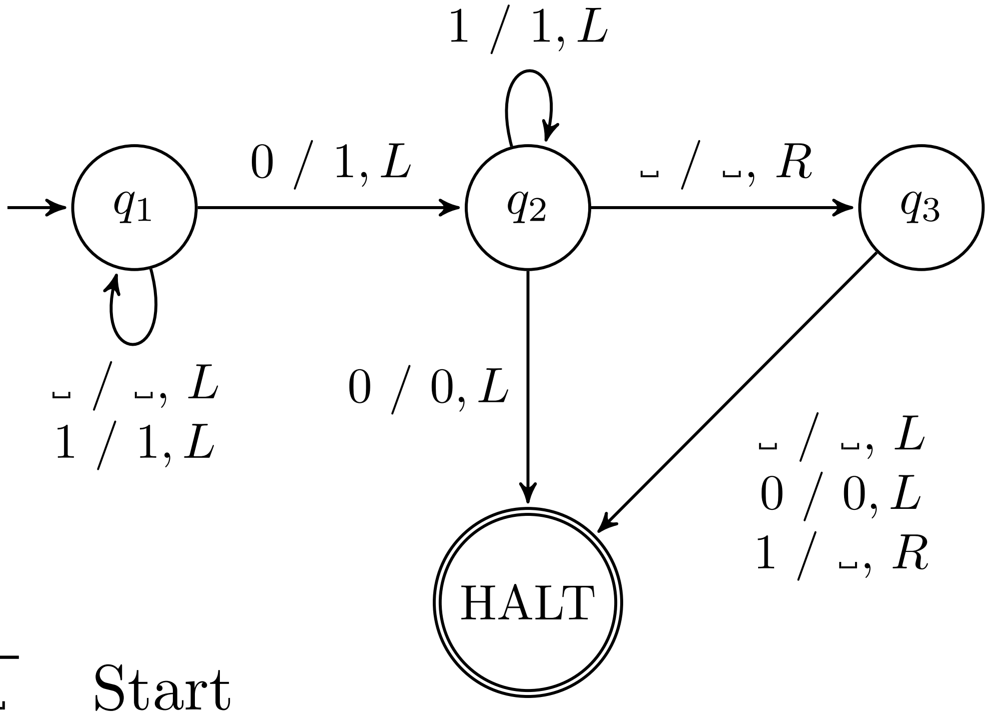
- Recognized language: Set of all input words where the TM stops in HALT state
  - TMs recognize the recursively enumerable languages

- Note:
  - A DFA/NFA/DPDA/PDA will always stop processing after it reaches the end of the input
  - A TM, however, may go into an infinite loop and never stop

Example: “Unary” Addition

TM that can add two integers in “unary” notation:

- integer x = represented by x ones, e.g., 3 = 111
- summands separated by 0, e.g., 3 + 2 = 111011
- tape alphabet $\Gamma = \{\sqcup, 0, 1\}$
- initial head position: anywhere to the right of the input string



$$1\begin{cases} \sqcup\ \sqcup\ L\ 1 \\ 0\ 1\ L\ 2 \\ 1\ 1\ L\ 1 \end{cases} \quad 2\begin{cases} \sqcup\ \sqcup\ R\ \ 3 \\ 0\ 0\ L\ \mathrm{HALT} \\ 1\ 1\ L\ \ 2 \end{cases} \quad 3\begin{cases} \sqcup\ \sqcup\ L\ \mathrm{HALT} \\ 0\ 0\ L\ \mathrm{HALT} \\ 1\ \sqcup\ R\ \mathrm{HALT} \end{cases}$$

Initial configuration

$\sqcup$	$\sqcup$	1	1	1	0	1	1	1	1	$\sqcup$	$\sqcup$	Start
										$\uparrow$		

Halt configuration

$\sqcup$	$\sqcup$	$\sqcup$	1	1	1	1	1	1	1	$\sqcup$	$\sqcup$	HALT
			$\uparrow$									

Nondeterministic TM (NTM)

- Can be defined similar to finite/pushdown automata
- The NTM will always choose a transition that leads to Halt, if it exists
- Depending on your point of view a NTM
  - can guess the correct transition without looking ahead
  - or process all possibilities in parallel and select the correct one at the end
    (this is not equivalent to parallel processing in computing – we have exponential growth)

- It can be proven that any nondeterministic TM can be converted to an equivalent deterministic TM (DTM) – a DTM is just as powerful as a NTM (but maybe slower)

What If We Restrict the Turing Machine Concept?

We can actually restrict our TM definition without loosing power. It has been proven:

- An alphabet with only two symbols (like 0, 1) is sufficient to do anything a TM can do
  - you may just need more states and more steps
- Two states are is sufficient to do anything a TM can do (initial and halt state)
  - but you may need a larger alphabet
- A tape that is infinite in both directions is not required – one-directional infinity suffices

Linear Bounded Automaton (LBA)

What if we restrict the tape of TM in both directions?

- Restrict tape length to length of input string
  - Finite memory, amount depending on input
  - We can still move left & right
- We get a Linear Bounded Automaton (LBA, *Linear beschränkter Automat*)
  - An LBA is less powerful than a TM; but still more powerful than a PDA!
- Whether nondeterministic LBAs are equivalent to deterministic LBAs is an open problem

- Recognized language: Set of all input words where the LBA stops in HALT state
  - LBAs recognize the context-sensitive languages
  - these are a proper
    - superset of the context-free languages
    - subset of the recursively enumerable languages

What If We Extend the Turing Machine Concept?

- There is no known extension that makes the TM concept more powerful
  - we have reason to believe that there is no such concept ⟶ Church-Turing thesis
  - extensions may just make processing more convenient/faster – but you cannot solve more problems
- For example, you can
  - add a neutral (N) position for head movement (i.e., the head does not move)
  - add multiple tapes with multiple r/w heads
  - let the r/w head move by more than a single cell
- There is a multitude of very diverse concepts regarding models of computation: Until now, they have all been proven to be equivalent to Turing Machines
  - in particular: models with random access memory, as in real computers

Universal Turing Machine

Technische Hochschule Rosenheim

- Universal Turing Machine (UTM) = TM that can simulate any other TM
  - A computer is basically a universal TM
  - Construction described by Alan Turing in 1936
- Therefore, any algorithm can be described as a TM and be executed by a universal TM
- A system that can simulate any TM is called Turing-complete (*Turing-vollständig*)

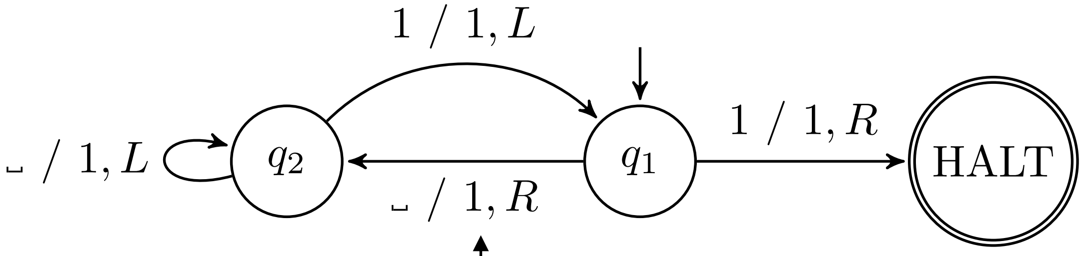
- Such a UTM surely is very large and complex? No! The smallest UTMs found have:
  - 4 states with 6 symbols and 22 instructions
  - 5 states with 5 symbols and 22 instructions
  - 15 states with 2 symbols and 29 instructions

Universal Turing Machine – Idea

- For simplification, consider the UTM having 3 tapes
  - Coding tape: Contains the encoding of the TM T to be simulated (its Gödel number)
  - Operating tape: Initially contains the input string for T, which is being processed on this tape
  - State tape: Used for storing the state the TM T would be in at each step

- Gödel number of T: Any injective mapping of TM T to natural numbers, where the inverse of the mapping can be computed
  - there are infinitely many ways of doing Gödel numbering
  - within a system the Gödel number for an instruction table is unique (otherwise: no inverse)
  - but not all natural numbers necessarily represent a TM (no one-to-one mapping)

Gödel Numbering – Simple Example



$\delta(q_1, \sqcup) = (q_2, 1, R)$

State/Symbol/Direction	Encoding $b(\cdot)$
q_1	0
q_2	00
HALT	000
1	0
$\sqcup$	00
R	0
L	00

Encode each such transition as

$$b(q_1)1b(\sqcup)1b(q_2)1b(1)1b(R) = 01001001010$$
$$b(q_2)1b(1)1b(q_1)1b(1)1b(L) = 00101010100$$

- $b(\cdot)$: mapping of states/symbols/directions to unary representation
- 1: separator

Concatenate all encoded transitions $111E_1 11E_2 11 \ldots 11E_4 111$

This is the Gödel number of TM T

$$111010010010101100101010100111 \ldots$$

Turing Machines and Real Computers

- A TM can perform any calculations that a computer can do
  - all restrictions on TM also apply to real computers
- In principle, a TM has an infinite amount of memory available, a computer does not
  - but: in finite time, a TM can only process a finite amount of data
- TM allow statements about algorithms independent of real computers
  - these will always remain true, regardless of changes in the architecture of computers
- A deterministic TM is much slower than a real computer
  - but: Time differences are bounded by polynomial factors, so this is not relevant in principle (see also: Chapter on time complexity)

Summary: Deterministic & Nondeterministic Automata

Deterministic Automaton	Nondeterministic Automaton	Are these equivalent?
DFA	NFA	yes
DPDA	PDA	no
DLBA	LBA	open problem
DTM	NTM	yes

Recognized languages:
regular ⊂ context-free ⊂ context-sensitive ⊂ recursively enumerable
regulär ⊂ *kontextfrei* ⊂ *kontextsensitiv* ⊂ *rekursiv aufzählbar*

DFA PDA LBA DTM
NFA NTM

So, we'll always use a Turing Machine, as it can do anything, right?

No: We'll try to use the simplest model available that can solve a problem

- the simpler the automaton model, the easier to handle
- computation time is typically larger for more general models (for example: see "word problem" in the next chapter)

Sources

- H. Ernst, J. Schmidt und G. Beneken: *Grundkurs Informatik*. Springer Vieweg, 7. Aufl., 2020.
- Schöning, U.: *Theoretische Informatik – kurz gefasst*. Spektrum Akad. Verlag (2008)
- Sander P., Stucky W., Herschel, R.: *Automaten, Sprachen, Berechenbarkeit*, B.G. Teubner, 1992
- D.W. Hoffmann. *Theoretische Informatik*. Hanser, 4. Aufl., 2018.