8 Turing machines and the rest

A Turing machine (TM) is a generalization of a PDA which uses a tape instead of a stack. Turing machines are an abstract version of a computer - they have been used to define formally what is computable. There are a number of alternative approaches to formalize the concept of computability (e.g. called the λ-calculus, or μ-recursive functions, ...) but they can all shown to be equivalent. That this is the case for any reasonable notion of computation is called the Church-Turing Thesis.

On the other side there is a generalization of context free grammars called phrase structure grammars or just grammars. Here we allow several symbols on the left hand side of a production, e.g. we may define the context in which a rule is applicable. Languages definable by grammars correspond precisely to the type-0-languages or the recursively enumerable languages (or semidecidable languages). Turing machines behave different from the previous machine classes we have seen: they may run forever, without stopping. To say that a language is uncomputable is applicable. Languages definable by grammars correspond precisely to the type-0-languages or the recursively enumerable languages (or semidecidable languages).

On the other side there is a generalization of context free grammars called phrase structure grammars or just grammars. Here we allow several symbols on the left hand side of a production, e.g. we may define the context in which a rule is applicable. Languages definable by grammars correspond precisely to the type-0-languages or the recursively enumerable languages (or semidecidable languages).

A Turing machine

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \]

is given by the following data:

- A finite set of states, \( Q \).
- A finite set \( \Sigma \) of symbols (the alphabet),
- A finite set \( \Gamma \) of tape symbols s.t. \( \Sigma \subseteq \Gamma \). This is the case because we use the tape also for the input.
- A transition function

\[ \delta : Q \times \Gamma \rightarrow \{ \text{stop} \} \cup Q \times \Gamma \times \{ L, R \} \]

The transition function defines how the function behaves if is in state \( q \) and the symbol on the tape is \( x \). If \( \delta(q, x) = \text{stop} \) then the machine stops otherwise if \( \delta(q, x) = (q', y, d) \) the machine gets into state \( q' \), writes \( y \) on the tape (replacing \( x \)) and moves left if \( d = L \) or right, if \( d = R \)

- An initial state \( q_0 \in Q \).
- The blank symbol \( B \in \Gamma \) but \( B \notin \Sigma \). In the beginning only a finite section of the tape containing the input is not blank.
- A set of final states \( F \subseteq Q \).

In [IMU01] the transition function is defined without the stop option as \( \delta \in Q \times \Gamma \rightarrow Q \times \Gamma \times \{ L, R \} \). However they allow \( \delta \) to be undefined which correspond to our function returning stop.

This defines deterministic Turing machines, for non-deterministic TMs we change the transition function to

\[ \delta \in Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{ L, R \}) \]

Here stop corresponds to \( \delta \) returning an empty set. As for finite automata (and unlike for PDAs) there is no difference in the strength of deterministic or non-deterministic TMs.

As for PDAs we define instantaneous descriptions ID for Turing machines. We have \( ID = I^* \times \Gamma \) where \( (\gamma, q, \gamma_i) \in I^* \) means that the TM is in state \( Q \) and left from the head the non-blank part of the tape is \( \gamma \) and starting with the head itself and all the non-blank symbols to the right is \( \gamma_i \).

We define the next state relation \( \rightarrow_M \) similar as for PDAs:

1. \( (\gamma, q, x\gamma_i) \rightarrow_M (\gamma y, q, \gamma_i) \) if \( \delta(q, x) = (q', y, R) \)
2. \( (\gamma z, q, x\gamma_i) \rightarrow_M (\gamma z y, q, \gamma_i) \) if \( \delta(q, x) = (q', y, L) \)
3. \( (\gamma, q, x) \rightarrow_M (\gamma y, q', \gamma_i) \) if \( \delta(q, B) = (q', y, R) \)
4. \( (c, q, x) \rightarrow_M (c' q, B y, \gamma_i) \) if \( \delta(q, x) = (q', y, L) \)

The cases 3. and 4. are only needed to deal with the situation if we have reached the end of the (non-blank part of) the tape.

We say that a TM \( M \) accepts a word if it goes into an accepting state, i.e. the language of a TM is defined as

\[ L(M) = \{ w \in \Sigma^* \mid (c, q_0, w) \rightarrow_M (\gamma, q_f, \gamma_i) \land q_f \in F \} \]

i.e. the TM stops automatically if it goes into an accepting state. However, it may also stop in a non-accepting state if \( \delta \) returns stop - in this case the word is rejected.

A TM \( M \) decides a language if it accepts it and it never loops (in the negative case).

To illustrate this we define a TM \( M \) which accepts the language \( L = \{ a^n b^n c^n \mid n \in \mathbb{N} \} \) — this is a language which cannot be recognized by a PDA or be defined by a CFG.

We define \( M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \) by

- \( Q = \{ q_0, q_1, q_2, q_3, q_4, q_5 \} \)
- \( \Sigma = \{ a, b, c \} \)
- \( \Gamma = \{ \} \)

To illustrate this we define a TM \( M \) which accepts the language \( L = \{ a^n b^n c^n \mid n \in \mathbb{N} \} \) — this is a language which cannot be recognized by a PDA or be defined by a CFG.
\[ f(X; Y; Z; q_0) = (q_1; X; Y; Z) \]
\[ g(q_1; Y) = (q_2; Z; Y; R) \]
\[ h(q_2; Z) = (q_3; Z; X; R) \]
\[ i(q_3; q_0) = (q_4; X; Y; Z; R) \]
\[ j(q_4; q_1) = (q_5; X; Y; Z; R) \]
\[ k(q_5; q_2) = (q_6; X; Y; Z; R) \]

Graphically the machine can be represented by the following transition diagram, where the edges are labelled by (read-symbol,write-symbol,move-direction).

The machine replaces an \( a \) by \( X \) \((q_0)\) and then looks for the first \( b \) replaces it by \( Y \) \((q_1)\) and looks for the first \( c \) and replaces it by a \( Z \) \((q_2)\). If there are more \( c \)'s left it moves left to the next \( a \) \((q_3)\) and repeats the cycle. Otherwise it checks whether there are no more \( a \)'s and \( b \)'s left \((q_4)\) and if so goes in an accepting state \((q_5)\).

Graphically the machine can be represented by the following transition diagram, where the edges are labelled by (read-symbol,write-symbol,move-direction).

8.2 Grammars and context-sensitivity

Grammars \( G = (V, \Sigma, S, P) \) are defined as context-free grammars before with the only difference that there may be several symbols on the left-hand side of a production, i.e. \( P \subseteq (V \cup T)^+ \times (V \cup T)^+ \). Here \((V \cup T)^+\) means that at least
one symbol has to present. The relation derives $\Rightarrow_G$ (and $\Rightarrow^*_G$) is defined as before

$$\Rightarrow_G \subseteq (V \cup T)^* \times (V \cup T)^*$$

and as before the language of $G$ is defined as

$$L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^*_G w \}$$

We say that a grammar is context-sensitive (or type 1) if the left hand side of a production is at least as long as the right hand side. That is for each $\alpha \Rightarrow \beta \in P$ we have $|\alpha| \leq |\beta|$.

Here is an example of a context sensitive grammar: $G = (V, \Sigma, S, P)$ with

$$L(G) = \{ a^n b^n c^n \mid n \in \mathbb{N} \land n \geq 1 \}$$

where

- $V = \{ S, B, C \}$
- $\Sigma = \{ a, b, c \}$
- $P = \{ S \rightarrow aSBC, S \rightarrow aBC, aB \rightarrow ab, CB \rightarrow BC, bb \rightarrow bB, bc \rightarrow bC, cc \rightarrow cC \}$

We present without proof:

**Theorem 8.1** For a language $L \subseteq \Sigma^*$ the following is equivalent:

1. $L$ is accepted by a Turing machine $M$, i.e. $L = L(M)$
2. $L$ is given by a grammar $G$, i.e. $L = L(G)$

**Theorem 8.2** For a language $L \subseteq \Sigma^*$ the following is equivalent:

1. $L$ is accepted by a Turing machine $M$, i.e. $L = L(M)$ such that the length of the tape is bounded by a linear function in the length of the input, i.e. $|s| + |s| \leq f(x)$ where $f(x) = ax + b$ with $a, b \in \mathbb{N}$.
2. $L$ is given by a context sensitive grammar $G$, i.e. $L = L(G)$

### 8.3 The halting problem

Turing showed that there are languages which are accepted by a TM (i.e. type 0 languages) but which are undecidable. The technical details of this construction are quite involved but the basic idea is quite simple and is closely related to Russell’s paradox, which we have seen in MCS.

Let’s fix a simple alphabet $\Sigma = \{ 0, 1 \}$. As computer scientist we are well aware that everything can be coded up in bits and hence we accept that there is an encoding of TMs in binary. I.e. given a TM $M$ we write $[M] \in \{ 0, 1 \}^*$ for its binary encoding. We assume that the encoding contains its length s.t. we know when subsequent input on the tape starts.

Now we define the following language

$$L_{\text{halt}} = \{ [M]w \mid M \text{ halts on input } w \}$$

It is easy (although the details are quite daunting) to define a TM which accepts this language: we just simulate $M$ and accept if $M$ stops.

However, Turing showed that there is no TM which decides this language. To see this let us assume that there is a TM $H$ which decides $L$. Now using $H$ we construct a new TM $F$ which is a bit obnoxious: $F$ on input $x$ runs $H$ on $xx$. If $H$ says yes then $F$ goes into a loop otherwise ($H$ says no) $F$ stops.

The question is what happens if I run $F$ on $[F]$?? Let us assume it terminates, then $H$ applied to $[F][F]$ returns yes and hence we must conclude that $F$ on $[F]$ loops?? On the other hand if $F$ with input $[F]$ loops then $H$ applied to $[F][F]$ will stop and reject and hence we have to conclude that $F$ on $[F]$ will stop????

This is a contradiction and hence we must conclude that our assumption that there is a TM $H$ which decides $L$ is false. We say $L_{\text{halt}}$ is undecidable.

We haven shown that a Turing machine cannot decide whether a program (for a Turing machine) halts. Maybe we could find a more powerful programming language which overcomes this problem? It turns out that all computational formalisms (i.e. programming languages) which can actually be implemented are equal in power and can be simulated by each other — this observation is called the Church-Turing thesis because it was first formulated by Alonzo Church and Alan Turing in the 30ies.

### 8.4 Back to Chomsky

At the end of the course we should have another look at the Chomsky hierarchy, which classifies languages based on subclasses of grammars or equivalently by different types of automata which recognize them.
We have worked our way from the bottom to the top of the hierarchy: starting with finite automata, i.e. computation with fixed amount of memory via pushdown automata (finite automata with a stack) to Turing machines (finite automata with a tape). Correspondingly we have introduced different grammatical formalisms: regular expressions, context-free grammars and grammars. Note that at each level there are languages which are on the next level but not on the previous: \( \{a^n b^n \mid n \in \mathbb{N}\} \) is level 2 but not level 3; \( \{a^n b^n c^n \} \) is level 1 but not level 2 and the Halting problem is level 0 but not level 1.

We could have gone the other way: starting with Turing machines and grammars and then introduce restrictions on them. I.e. Turing machines which only use their tapes as a stack, and Turing machines which never use the tape apart for reading the input. Again correspondingly we can define restrictions on the grammar size: first introduce context-free grammars and then grammars where all productions are of the form \( A \rightarrow aB \) or \( A \rightarrow a \), with \( A, B \) non-terminal symbols and \( a, b \) are terminals. These grammars correspond precisely to regular expressions (I leave this as an exercise).

I believe that Chomsky introduced his hierarchy as a classification of grammars and that the relation to automata was only observed a bit later. This is maybe the reason why he introduced the Type-1 level, which is not so interesting from an automata point of view (unless you are into computational complexity, i.e. resource use - here linear use of memory). It is also the reason why on the other hand the decidable languages do not constitute a level: there is no corresponding grammatical formalism (we can even prove this).