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 ones which may be accepted by a Turing machine and those are called *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 accepted by a Turing machine means that the TM will stop in an accepting state for each word which is in the language. However, if the word is not in the language the Turing machine may stop in a non-accepting state or loop forever. In this case we can never be sure whether the given word is in the language - i.e. the Turing machine doesn't decide the word problem.
We say a language is *recursive* (or *decidable*), if there is a TM which will always stop. There are *type-0-languages* which are not recursive — the most famous one is the *halting problem*. This is the language of encodings of Turing machines which will always stop.
There is no type of grammars which captures all recursive languages (and for theoretical reasons there cannot be one). However there is a subset of recursive languages which are called *context-sensitive languages* which are given by *context-sensitive grammars*, these are those grammars where the left hand side of a production is always shorter than the right hand side. Context sensitive languages on the other hand correspond to linear bounded TMs, these are those TMs which use only a tape whose length can be given by a linear function over the length of the input.

8.1 What is a Turing machine?

A Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ is given by the following data

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

$$\delta \in Q \times \Gamma \to \{\text{stop}\} \cup Q \times \Gamma \times \{\text{L}, \text{R}\}$$

  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 machines gets into state q', writes y on the tape (replacing x) and moves left if $d = \text{L}$ or right, if $d = \text{R}$

- 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 [HMU01] the transition function is defined without the stop option as $\delta \in Q \times \Gamma \to Q \times \Gamma \times \{\text{L}, \text{R}\}$. However they allow δ 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

$$\delta \in Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{\text{L}, \text{R}\})$$

Here stop corresponds to δ 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 $\text{ID} = \Gamma^* \times Q \times \Gamma^*$ where $(\gamma_l, q, \gamma_r) \in \text{ID}$ means that the TM is in state Q and left from the head the non-blank part of the tape is γ_l and starting with the head itself and all the non-blank symbols to the right is γ_r.
We define the next state relation $\vdash_M$ similar as for PDAs:

1. $(\gamma_l, q, x\gamma_r) \vdash_M (\gamma_l y, q', \gamma_r)$ if $\delta(q, x) = (q', y, R)$
2. $(\gamma_l z, q, x\gamma_r) \vdash_M (\gamma_l, q', zy\gamma_r)$ if $\delta(q, x) = (q', y, L)$
3. $(\gamma_l, q, \epsilon) \vdash_M (\gamma_l y, q', \gamma_r)$ if $\delta(q, B) = (q', y, R)$
4. $(\epsilon, q, x\gamma_r) \vdash_M (\gamma_l, q', By\gamma_r)$ 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

$$L(M) = \{w \in \Sigma^* \mid (\epsilon, q_0, w) \vdash_M^* (\gamma_l, q', \gamma_r) \land q' \in F\}$$

I.e. the TM stops automatically if it goes into an accepting state. However, it may also stop in a non-accepting state if δ 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

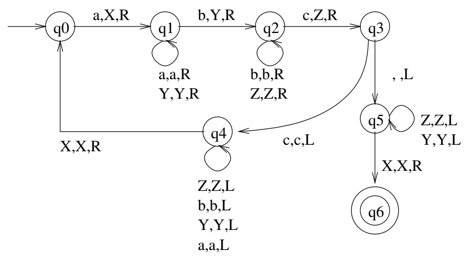
- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$

- $\Gamma = \Sigma \cup \{X, Y, Z, \sqcup\}$
- δ is given by

$$
\begin{array}{rcll}
\delta(q_0, \sqcup) & = & (\sqcup, q_6, \mathrm{R}) & \\
\delta(q_0, \mathtt{a}) & = & (X, q_1, \mathrm{R}) & \\
\delta(q_1, \mathtt{a}) & = & (\mathtt{a}, q_1, \mathrm{R}) & \\
\delta(q_1, Y) & = & (Y, q_1, \mathrm{R}) & \\
\delta(q_1, \mathtt{b}) & = & (Y, q_2, \mathrm{R}) & \\
\delta(q_2, \mathtt{b}) & = & (\mathtt{b}, q_2, \mathrm{R}) & \\
\delta(q_2, Z) & = & (Z, q_2, \mathrm{R}) & \\
\delta(q_2, \mathtt{c}) & = & (Z, q_3, \mathrm{R}) & \\
\delta(q_3, \sqcup) & = & (\sqcup, q_5, \mathrm{L}) & \\
\delta(q_3, \mathtt{c}) & = & (\mathtt{c}, q_4, \mathrm{L}) & \\
\delta(q_4, Z) & = & (Z, q_4, \mathrm{L}) & \\
\delta(q_4, \mathtt{b}) & = & (\mathtt{b}, q_4, \mathrm{L}) & \\
\delta(q_4, Y) & = & (Y, q_4, \mathrm{L}) & \\
\delta(q_4, \mathtt{a}) & = & (\mathtt{a}, q_4, \mathrm{L}) & \\
\delta(q_4, X) & = & (X, q_0, \mathrm{R}) & \\
\delta(q_5, Z) & = & (Z, q_5, \mathrm{L}) & \\
\delta(q_5, Y) & = & (Y, q_4, \mathrm{L}) & \\
\delta(q_5, X) & = & (X, q_6, \mathrm{R}) & \\
\delta(q, x) & = & \text{stop} & \text{everywhere else}
\end{array}
$$

- $q_0 = q_0$
- $B = \sqcup$
- $F = \{q_6\}$

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_4) and repeats the cycle. Otherwise it checks whether there are no `a`s and `b`s left (q_5) and if so goes in an accepting state (q_6). Graphically the machine can be represented by the following transition diagram, where the edges are labelled by (read-symbol,write-symbol,move-direction):



E.g. consider the sequence of IDs on `aabbcc`:

$$
\begin{array}{rl}
(\epsilon, q_0, \mathtt{aabbcc}) & \vdash (\mathtt{X}, q_1, \mathtt{abbcc}) \\
& \vdash (\mathtt{Xa}, q_1, \mathtt{bbcc}) \\
& \vdash (\mathtt{XaY}, q_2, \mathtt{bcc}) \\
& \vdash (\mathtt{XaYb}, q_2, \mathtt{cc}) \\
& \vdash (\mathtt{XaYbZ}, q_3, \mathtt{c}) \\
& \vdash (\mathtt{XaYb}, q_4, \mathtt{Zc}) \\
& \vdash (\mathtt{XaY}, q_4, \mathtt{bZc}) \\
& \vdash (\mathtt{Xa}, q_4, \mathtt{YbZc}) \\
& \vdash (\mathtt{X}, q_4, \mathtt{aYbZc}) \\
& \vdash (\epsilon, q_4, \mathtt{XaYbZc}) \\
& \vdash (\mathtt{X}, q_0, \mathtt{aYbZc}) \\
& \vdash (\mathtt{XX}, q_1, \mathtt{YbZc}) \\
& \vdash (\mathtt{XXY}, q_1, \mathtt{bZc}) \\
& \vdash (\mathtt{XXYY}, q_2, \mathtt{Zc}) \\
& \vdash (\mathtt{XXYYZ}, q_2, \mathtt{c}) \\
& \vdash (\mathtt{XXYYZZ}, q_2, \epsilon) \\
& \vdash (\mathtt{XXYYZ}, q_5, \mathtt{Z}) \\
& \vdash (\mathtt{XXYY}, q_5, \mathtt{ZZ}) \\
& \vdash (\mathtt{XXY}, q_5, \mathtt{YZZ}) \\
& \vdash (\mathtt{XX}, q_5, \mathtt{YYZZ}) \\
& \vdash (\mathtt{X}, q_5, \mathtt{XYYZZ}) \\
& \vdash (\epsilon, q_6, \mathtt{XXYYZZ})
\end{array}
$$

We see that M accepts `aabbcc`. Since M never loops it does actually decide L.

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

$$\Rightarrow_G \subseteq (V \cup T)^* \times (V \cup T)^*$$
$$\alpha\beta\gamma \Rightarrow_G \alpha\beta'\gamma :\iff \beta \to \beta' \in P$$

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 $\alpha \to \beta \in P$ 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

- $V = \{S, B, C\}$
- $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$
-

$$\begin{aligned} P = \{ & S \to aSBC \\ & S \to aBC \\ & aB \to ab \\ & CB \to BC \\ & bB \to bb \\ & bC \to bc \\ & cC \to cc\} \end{aligned}$$

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. $|\gamma_l| + |\gamma_r| \leq f(x)$ 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.

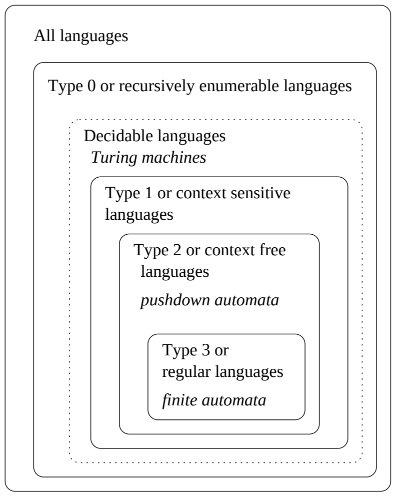
Let's fix a simple alphabet $\Sigma = \{0, 1\}$. As computer scientist we are well aware that everything can be coded up in bits and hence we accept that there is an encoding of TMs in binary. I.e. given a TM M we write $\lceil M \rceil \in \{0, 1\}^*$ for its binary encoding. We assume that the encoding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

$$L_{\text{halt}} = \{\lceil M \rceil w \mid M \text{ holds on input } w.\}$$

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 $\lceil F \rceil$? Let us assume it terminates, then H applied to $\lceil F \rceil \lceil F \rceil$ returns yes and hence we must conclude that F on $\lceil F \rceil$ loops??? On the other hand if F with input $\lceil F \rceil$ loops then H applied to $\lceil F \rceil \lceil F \rceil$ will stop and reject and hence we have to conclude that F on $\lceil F \rceil$ will stop?????
This is a contradiction and hence we must conclude that our assumption that there is a TM H which decides L_{halt} is false. We say L_{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 sublasses of grammars or equivalently by different types of automata which recognize them


All languages
Type 0 or recursively enumerable languages
Decidable languages
Turing machines
Type 1 or context sensitive languages
Type 2 or context free languages
pushdown automata
Type 3 or regular languages
finite automata

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). Correspondigly 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 sise: first introduce context-free grammars and then grammars where all productions are of the form $A \to aB$ or $A \to a$, 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 herarchy 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).