6.5 The Halting Problem

Theorem 6.9 The language

\[ H = \{c_M w \mid M \text{ halts with input } w \} \]

is recursively enumerable, but not recursive.

Tocistus. We show first that \( H \) is recursively enumerable. The universal machine \( M_U \) in the proof of Theorem 6.6 is easily modified to a machine, that on the input \( c_M w \) simulates the computation of \( M \) on the input \( w \) and accepts, if the simulated computation ever halts.

We show next that \( H \) is not recursive. Assume that \( H = L(M_U) \) for some total Turing machine. Assume also, that \( M_U \) leaves the original inputs, possibly appended with blanks, on the tape upon halting. Let \( M_U \) be the universal Turing machine in the proof of Theorem 6.6.

Now we could form a total recognizer for the language \( U \) by combining the machines \( M_U \) and \( M_U \) as follows:

By theorem 6.7 such a recognizer of \( U \) cannot exist. This contradiction shows that \( H \) cannot be recursive.

\[ \square \]

Corollary 6.10 The language

\[ \bar{H} = \{c_M w \mid M \text{ does not halt on input } x \} \]

is not recursively enumerable.

\[ \square \]

6.7 Rice’s theorem

By Rice’s theorem \textit{all} nontrivial questions regarding the languages Turing machine recognize, i.e., regarding the I/O-mappings they compute, are undecidable.

We first examine a special case, the \textit{nonemptiness problem} of the language a Turing machine recognizes: “Does a given Turing machine accept any input string?” The formal language corresponding to the problem is

\[ \text{NE} = \{c \in \{0,1\}^* \mid L(M) \neq \emptyset \} \]

Theorem 6.11 The language \( \text{NE} \) is recursively enumerable, but not recursive.
Proof. First we find that NE is recursively enumerable by designing a recognizer $M_{NE}$ for it. It is easiest to design $M_{NE}$ as nondeterministic. Let $M_{OK}$ be a Turing machine that tests whether the input is a legal encoding of a Turing machine, and let $M_G$ be a nondeterministic Turing machine, that appends an arbitrary binary string $w$ to the end of the input. The machine $M_{NE}$ can be combined from $M_{OK}$, $M_G$ and the universal machine $M_U$ as follows:

\[
\begin{align*}
M_{OK} & \quad M_G & \quad M_U
\end{align*}
\]

Clearly

\[
\begin{align*}
c \in L(M_{NE}) & \iff c \text{ is a legal TM-encoding and } \exists w \text{ s.t. } cw \in U \\
& \iff c \text{ is a legal TM-encoding and } \exists w \text{ s.t. } w \in L(M_G) \\
& \iff L(M_G) \neq \emptyset.
\end{align*}
\]

We show that NE is not recursive. Assume that NE would have a total recognizer $M_{NE}$ and use it to construct a total recognizer $M_U$ for the language $U$ (contradiction).

In this construction, inputs are encoded as “program constants” in the Turing machine. Let $M$ be an arbitrary Turing machine whose behaviour under the input $w = a_1 a_2 \ldots a_k$ we wish to examine. Use $M^w$ to denote a machine, that first replaces its input by the string $w$ and then functions as $M$:

\[
\begin{align*}
M^w : & \quad a_1 a_2 \ldots a_k \rightarrow a_1 a_2 \ldots a_k \rightarrow \ldots
\end{align*}
\]

The behaviour of $M^w$ does not depend on the actual input; rather it accepts or rejects all strings depending on whether $M$ accepts $w$:

\[
L(M^w) = \begin{cases} 
\{0, 1\}^*, & \text{ if } w \in L(M) \\
\emptyset, & \text{ if } w \notin L(M).
\end{cases}
\]

Now let $M_{ENCODE}$ be a Turing machine that receives as input the string $cw^w$, where $c$ is the encoding of an arbitrary Turing machine $M$ and $w$ is a binary string, and leaves the encoding $c_{w^w}$ of the previously described $M^w$ on the tape:

\[
\begin{align*}
M^w & \quad M_{ENCODE}
\end{align*}
\]

(If the input is not of the form $cw$ with $c$ a legal encoding of a Turing machine, $M_{ENCODE}$ halts in the rejecting state.) Thus $M_{ENCODE}$ operates on encodings of Turing machines. It adds transitions to the code of the given Turing machine $M$ and renumerates the states so that the resulting code represents $M^w$ instead of $M$. 

\[
\begin{align*}
cw^w & \quad M_{ENCODE}
\end{align*}
\]
By combining \( M_{\text{ENCOD}} \) and the hypothetical \( M_{\text{NE}}^T \) as follows we could form a total recognizer \( M_{T}^U \) for the language \( U \):

\[
\begin{aligned}
\text{\( q_0 \)} & \overset{\text{\( M_{\text{ENCOD}} \)}}{\rightarrow} \ \text{\( q_0 \)w} \overset{\text{\( M_{\text{NE}}^T \)}}{\rightarrow} \ \text{\( M_{T}^U \)w} \\
\end{aligned}
\]

The machine \( M_{T}^U \) is total, if \( M_{\text{NE}}^T \) is, and \( L(M_{T}^U) = U \), since:

\[
\text{\( q_0 \)w} \in L(M_{T}^U) \iff \text{\( q_0 \)w} \in L(M_{\text{NE}}^T) = \text{\( NE \)} \iff L(M^w) \neq \emptyset \iff w \in L(M).
\]

But \( U \) is not recursive, so such a total recognizer \( M_{T}^U \) is impossible. By this contradiction, the language \( \text{NE} \) cannot have a total recognizer \( M_{\text{NE}}^T \).

**Rice’s Theorem**

A semantic property \( S \) of Turing machines is any collection of recursively enumerable languages over the alphabet \( \{0, 1\} \); a machine \( M \) has the property \( S \), if \( L(M) \in S \). Trivial properties are \( S = \emptyset \) (a property no Turing machine has) and \( S = \text{RE} \) (a property all Turing machines have).

A property \( S \) is decidable, if the set

\[
\text{codes}(S) = \{ c \mid L(M_c) \in S \}
\]

is recursive. In other words: a property is decidable, if by observing a Turing machine we can algorithmically deduce whether the machine has the desired semantic property.

**Theorem 6.12 [Rice 1953]** All nontrivial semantic properties of Turing machines are undecidable.

**Proof.** Let \( S \) be an arbitrary nontrivial semantic property. We may assume that \( \emptyset \notin S \); in other words, that the Turing machines that recognize \( S \) do not have this property. Namely, if \( \emptyset \in S \), we can first show that the property \( S = \text{RE} - S \) is undecidable and then see that \( S \) is undecidable as well. (Since \( \text{codes}(S) = \text{codes}(S') \).)

Since \( S \) is nontrivial, there is some Turing machine \( M_A \) with property \( S \) — thus \( L(M_A) \neq \emptyset \in S \).

This time, let \( M_{\text{ENCOD}} \) be a Turing machine that transforms its input \( q_0 \)w into an encoding of the following Turing machine \( M^w \).

If the input is not of the correct form \( M_{\text{ENCOD}} \) halts in the rejecting state.

With input \( x \) the machine \( M^w \) first does as \( M \) does with input \( w \). If \( M \) accepts \( w \), \( M^w \) then does as \( M_A \) does with input \( x \). If \( M \) rejects \( w \), also \( M^w \) rejects \( x \). The machine \( M^w \) thus recognizes the language

\[
L(M^w) = \begin{cases} 
L(M_A), & \text{if } w \in L(M) \\
\emptyset, & \text{if } w \notin L(M).
\end{cases}
\]

Since by assumption \( L(M_A) \in S \) and \( \emptyset \notin S \), machine \( M^w \) now has property \( S \), if and only if \( w \in L(M) \).
Assume then that \( S \) were decidable, i.e., that the language codes(\( S \)) would have a total recognizer \( M^*_S \). In style of the previous proof, a total recognizer for \( U \) could be combined from \( M_{\text{ENCODE}} \) and \( M^*_S \) as follows:

Clearly \( M^*_U \) is total, if \( M^*_S \) is, and

\[
\omega_W \in L(M^*_U) \iff \omega_W \in L(M^*_S) \iff \text{codes}(S) \iff L(M^*), S \iff \omega \in L(M).
\]

Since \( U \) is not recursive, this is impossible, and hence \( S \) cannot be decidable. \( \Box \)

6.8 Other undecidability results

Theorem 6.13 (Undecidability of predicate calculus; Church/Turing 1936)

There is no algorithm for deciding whether a given first order predicate calculus formula \( \phi \) is valid ("logically true", provable from the axioms of predicate calculus). \( \Box \)

Theorem 6.14 ("Hilbert's 10th problem": Matiyasevich/Davis/Robinson/Putnam 1953–70)

There is no algorithm for deciding whether a given polynomial \( P(x_1, \ldots, x_n) \) with integer coefficients has integer roots (that is, tuples \( (m_1, \ldots, m_n) \in \mathbb{Z}^n \), for which \( P(m_1, \ldots, m_n) = 0 \)). The problem is undecidable already when \( n = 15 \) or \( \deg(P) = 4 \). \( \Box \)

6.9 Recursive functions

A partial function computed by the Turing machine \( M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{ rej}}) \)

\[
f_M : \Sigma^* \rightarrow \Gamma^*
\]

is defined by:

\[
f_M(x) = \begin{cases} 
    u, & \text{if } (q_0, x) \vdash^{*} (q, uav) \text{ for some } q \in \{q_{\text{acc}}, q_{\text{ rej}}\}, av \in \Gamma^*; \\
    \text{undefined, otherwise.}
\end{cases}
\]

A partial function \( f : \Sigma^* \rightarrow A \) is partial recursive if it can be computed by some Turing machine and recursive, if it can be computed by a total Turing machine. Equivalently we could define that a partial recursive function \( f \) is recursive, if \( f(x) \) is defined for all \( x \).
Theorem 6.15
(i) A language \( A \subseteq \Sigma^* \) is recursive if and only if its characteristic function
\[
\chi_A : \Sigma^* \to \{0, 1\}, \quad \chi_A(x) = \begin{cases} 
1, & \text{if } x \in A \\
0, & \text{if } x \notin A
\end{cases}
\]
is a recursive function.
(ii) A language \( A \subseteq \Sigma^* \) is recursively enumerable if and only if \( A = \emptyset \) or there exists a recursive function \( g : \{0, 1\}^* \to \Sigma^* \), for which
\[
A = \{g(x) \mid x \in \{0, 1\}^*\}.
\]
Proof. Exercise.  \( \square \)

6.10 Recursive reduction and RE-complete languages
A formal language \( A \subseteq \Sigma^* \) can be reduced recursively to language \( B \subseteq \Gamma^* \), denoted by
\( A \leq_m B \), if there is a recursive function \( f : \Sigma^* \to \Gamma^* \), with the property:
\[
x \in A \iff f(x) \in B, \quad \text{for all } x \in \Sigma^*.
\]

Lemma 6.16 For all languages \( A, B, C \) it holds that:
(i) \( A \leq_m A \);
(ii) if \( A \leq_m B \) and \( B \leq_m C \), then \( A \leq_m C \);
(iii) if \( A \leq_m B \) and \( B \) is recursively enumerable, then \( A \) is recursively enumerable;
(iv) if \( A \leq_m B \) and \( B \) is recursive, then \( A \) is recursive.  \( \square \)

Let us denote:
\[
\begin{align*}
\text{RE} &= \{\text{the recursively enumerable languages over } \{0, 1\}\}; \\
\text{REC} &= \{\text{the recursive languages over } \{0, 1\}\}.
\end{align*}
\]
A language \( A \subseteq \{0, 1\}^* \) is RE-complete, if
(i) \( A \in \text{RE} \) and
(ii) \( B \leq_m A \) for all \( B \in \text{RE} \).

Theorem 6.17 The language \( U \) is RE-complete.
Proof. We know that \( U \in \text{RE} \). Let \( B = \text{L}(M_B) \) be an arbitrary language in \( \text{RE} \). Now \( B \) may be reduced to \( U \) by the function \( f(x) = \alpha_{M_B} x \).
This function is clearly recursive, and has the property
\[
x \in B = \text{L}(M_B) \iff f(x) = \alpha_{M_B} x \in U. \quad \square
\]

Lemma 6.18 Let \( A \) be an RE-complete language, \( B \in \text{RE} \) and \( A \leq_m B \). Then also \( B \) is RE-complete.  \( \square \)

From Rice’s Theorem it follows that among other all problems, where something is to be deduced about the language a Turing machine recognizes by looking at the code of the Turing machine are RE-complete. In general it appears that all “natural” recursively enumerable, non-recursive languages are RE-complete. However, it can be shown (proof omitted) that:

Theorem 6.19 (E. Post 1944) In \( \text{RE} \) – \( \text{REC} \) there are languages that are not RE-complete.  \( \square \)
Since the class $\text{RE}$ is not closed under complementation, it has the natural complement class $\text{co-RE} = \{ \overline{A} \mid A \in \text{RE} \}$.

By Theorem 6.3 we have $\text{RE} \cap \text{co-RE} = \text{REC}$.

In $\text{co-RE}$ the concept of a complete language can be defined similarly as in $\text{RE}$: a language $A \subseteq \{0,1\}^*$ is $\text{co-RE}$-complete, if $A \in \text{co-RE}$ and $B \leq_{m} A$ for all $B \in \text{co-RE}$. It is easy to see that a language $A$ is $\text{co-RE}$-complete, if and only if the language $\overline{A}$ is $\text{RE}$-complete (exercise).

---

Just a couple more results from computability theory without proofs.

Proof 6.20 The language

$$\text{TOT} = \{ c \mid \text{The Turing machine } M_c \text{ halts on all inputs} \}$$

is neither in $\text{RE}$ nor in $\text{co-RE}$. 

It is said that $A,B \subseteq \{0,1\}^*$ are recursively isomorphic, if there is a recursive bijection $f : \{0,1\}^* \rightarrow \{0,1\}^*$ (then also the inverse function $f^{-1}$ must be recursive), for which

$$\forall x \in A \leftrightarrow f(x) \in B, \text{ for all } x \in \Sigma^*.$$

Theorem 6.21 (J. Myhill 1955) All $\text{RE}$-complete languages are recursively isomorphic.

---

5. UNRESTRICTED GRAMMARS

Definition 5.1 An unrestricted grammar or a string rewriting system is a 4-tuple

$$G = (V, \Sigma, P, S),$$

where

- $V$ is the alphabet of the grammar;
- $\Sigma \subseteq V$ is the set of terminals; $N = V - \Sigma$ is the set of non-terminals;
- $P \subseteq V^+ \times V^*$ is the set of rules or productions ($V^+ = V^* - \{\varepsilon\}$);
- $S \in N$ is the initial symbol.

The production $(\omega, \omega') \in P$ is usually denoted with $\omega \rightarrow \omega'$.

The string $\gamma \in V^*$ produces the string $\gamma' \in V^*$ grammar $G$, denoted by

$$\gamma \Rightarrow_0 \gamma'$$

if we may write $\gamma = \omega \beta, \gamma' = \omega' \beta'$ ($\alpha, \beta, \omega', \omega \in V^*$), and the grammar contains the production $\omega \rightarrow \omega'$.

If the grammar $G$ is clear from the context, we write $\gamma \Rightarrow \gamma'$.

The string $\gamma \in V^*$ derives the string $\gamma' \in V^*$ in grammar $G$, denoted with

$$\gamma \Rightarrow^* \gamma'$$

if there is a sequence $\gamma_0, \gamma_1, \ldots, \gamma_n (n \geq 0)$ of strings on $V$ such that

$$\gamma = \gamma_0 \Rightarrow_0 \gamma_1 \Rightarrow_0 \cdots \Rightarrow_0 \gamma_n = \gamma'.$$

Again, if $G$ is obvious from the context, we write $\gamma \Rightarrow^* \gamma'$. 
A string $\gamma \in V^*$ is a sentence derivation of $G$, if $S \Rightarrow^*_G \gamma$. A sentence derivation $x \in \Sigma^*$ that only consists of terminals is a sentence of $G$. The language $L(G)$ produced by $G$ consists of the sentences of $G$, that is

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

**Example.** An unrestricted grammar for the non-context-free language

$$\{ a^k b^k c^k \mid k \geq 0 \}.$$ 

- $S \rightarrow LT \mid \varepsilon$
- $T \rightarrow ABCT \mid ABC$
- $BA \rightarrow AB$
- $CB \rightarrow BC$
- $CA \rightarrow AC$
- $LA \rightarrow a$
- $aA \rightarrow aa$
- $aB \rightarrow ab$
- $bB \rightarrow bb$
- $bC \rightarrow bc$
- $cC \rightarrow cc$

**Example:** derivation of the sentence $aabbcc$:

$$S \Rightarrow LT \Rightarrow LABCT \Rightarrow LABCABC \Rightarrow LABACBC \Rightarrow aABBCC$$

$$\Rightarrow aaBBCC \Rightarrow aabBCC \Rightarrow aabbbcC \Rightarrow aabbbcC.$$ 

**Theorem 5.1** If a formal language $L$ can be produced by an unrestricted grammar, it can be recognized by a Turing machine.

**Proof.** Let $G = (V, \Sigma, P, S)$ be an unrestricted grammar that produces $L$. We present a two-tape nondeterministic Turing machine $M_G$ that recognizes $L$.

A copy of the input string is kept on tape 1. Tape 2 always contains a sentence production of $G$; the machine tries to change that to the input string. At start $M_G$ writes the initial symbol $S$ of the grammar onto tape 2.

The computation of $M_G$ consist of phases. At each phase the machine:

(i) nondeterministically places the tape head somewhere on tape 2;

(ii) nondeterministically chooses a production of $G$ and tries to apply it at the chosen point of tape 2 (the productions are encoded in the transition function of $M_G$);

(iii) if the left side of the production matches the symbols on the tape, $M_G$ replaces the symbols by the symbols on the right side of the production;

(iv) at the end of a phase $M_G$ compares tape 1 and tape 2; if the contents are equal, the machine accepts the input; otherwise the machine starts a new phase (goto (i)).
**Theorem 5.2** If a formal language $L$ can be recognized by a Turing machine, it can be produced by an unrestricted grammar.

*Proof idea.* Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{halt})$ be a standard model Turing machine that recognizes $L$. Let us form the unrestricted grammar $G_M$ that produces $L$ as follows:

Among others, symbols to represent each state $q \in Q$ of $M$ are included in $G_M$. The configuration $(q, uav)$ of $M$ is represented as the string $[uqv]$. From the transition function of $M$ productions of $G_M$ are formed so that

$$[uqv] \Rightarrow [Uq'd'v'] \text{ iff } (q, uav) \Rightarrow_M (q', Ud'v').$$

Thus $M$ accepts $x$ if and only if

$$[q_0x] \Rightarrow^* [uq_{accept}v]$$

for some $u, v \in \Sigma^*$.

---

Let us define $G = (V, \Sigma, P, S)$ where

$$V = \Gamma \cup Q \cup \{S, T, [\cdot], E_L, E_R\} \cup \{A_a | a \in \Sigma\},$$

and the productions $P$ contain the following three groups of productions:

1. Producing the initial configuration

   $$S \rightarrow T[q_0]$$

   $$T \rightarrow \varepsilon$$

   $$A_a[q_0] \rightarrow \{q_aA_a \quad (a \in \Sigma)$$

   $$A_a[q_a] \rightarrow \{q_aA_a \quad (a \in \Sigma)$$

   $$A_a \rightarrow a \quad (a \in \Sigma)$$

2. Simulating steps of the computation of $M$ ($a, b \in \Gamma, c \in \Gamma \cup \{[\cdot]\}$):

   **Transitions:**

   $$\delta(q, a) = (q', b, R) \quad qa \rightarrow bq'$$

   $$\delta(q, a) = (q', b, L) \quad caa \rightarrow q'ab$$

   $$\delta(q, >) = (q', >, R) \quad q' \rightarrow [q']$$

   $$\delta(q, <) = (q', b, R) \quad q' \rightarrow bq'$$

   $$\delta(q, >) = (q', b, L) \quad ca \rightarrow q'ab$$

   $$\delta(q, <) = (q', <, L) \quad ca \rightarrow q'c$$

**Productions:**
3. Cleaning up the final configuration:

\[ q_{acc} \rightarrow E_L E_R \]
\[ q_{acc} \rightarrow E_R \]
\[ aE_L \rightarrow E_L \quad (a \in \Gamma) \]
\[ [E_L \rightarrow \varepsilon \quad (a \in \Gamma) \]
\[ E_R a \rightarrow E_R \quad (a \in \Gamma) \]
\[ E_R ] \rightarrow \varepsilon \]

Context sensitive grammars

An unrestricted grammar is context sensitive, if all its productions are of the form \( \omega \rightarrow \omega' \), where \(|\omega'| \geq |\omega|\), or possibly \( S \rightarrow \varepsilon \), where \( S \) is the initial symbol.

Additionally, if the grammar contains the rule \( S \rightarrow \varepsilon \), then the initial symbol \( S \) must not appear on the right hand side of any production.

A formal language \( L \) is context sensitive, if it can be produced by a context sensitive grammar.

Normal form: Every context sensitive language can be produced by a grammar, whose productions are of the form \( S \rightarrow \varepsilon \) and \( \alpha A \beta \rightarrow \alpha \omega \beta \), where \( A \) is a non-terminal and \( \omega \neq \varepsilon \). (The rule \( A \rightarrow \omega \) applied "in the context" \( \alpha \_ \beta \).)

Theorem 5.3 A formal language \( L \) is context sensitive if and only if it can be recognized by a nondeterministic Turing machine that needs no more work space than the length of the input string — that is, by a machine that has no transitions of the form \( \delta(q, <) = (q', b, \Delta) \), where \( b \neq < \).

Machines such as those in Theorem 5.3 are called linearly bounded automata.

An open problem: ("LBA ?= DLBA"): is nondeterminism necessary in Theorem 5.3?

The Chomsky hierarchy

A grouping of grammars, the languages produced by them, and the corresponding automata:

**Class 0**: unrestricted grammars / recursively enumerable languages / Turing machines.

**Class 1**: context sensitive grammars / context sensitive languages / linearly bounded automata.

**Class 2**: context-free grammars / context-free languages / pushdown automata.

**Class 3**: right and left linear (regular) grammars / regular languages / finite state automata.