6.5 The Halting Problem

Theorem 6.9 The language

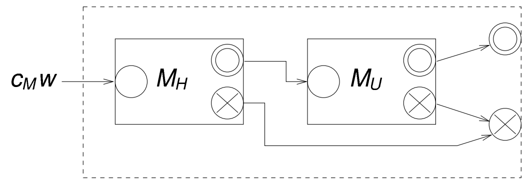
$$H = \{c_M w \mid M \text{ halts with input } w\}$$

is recursively enumerable, but not recursive.

Todistus. 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 $H = L(M_H)$ for some total Turing machine. Assume also, that M_H 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_H and M_U as follows:



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

Corollary 6.10 The language

$$\tilde{H} = \{c_M w \mid M \text{ does not halt on input } x\}$$

is not recursively enumerable. □

6.7 Rice's theorem

By Rice's theorem *all* nontrivial questions regarding the languages Turing machine recognize, i.e., regarding the I/O-mappings they compute, are undecidable.

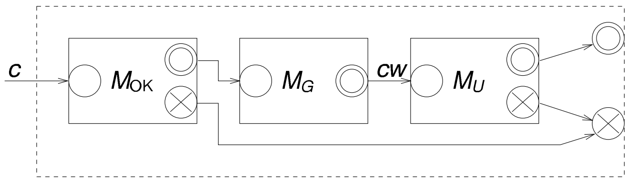
We first examine a special case, the *nonemptiness problem* of the language a Turing machine recognizes: "Does a given Turing machine accept any input string?" The formal language corresponding to the problem is

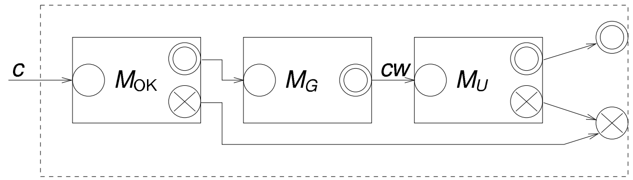
$$\mathrm{NE} = \{c \in \{0,1\}^* \mid L(M_c) \neq \emptyset\}.$$

Theorem 6.11 The language 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:



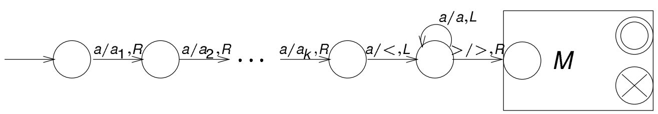


Clearly

$$
\begin{aligned}
& c \in L(M_{\mathrm{NE}}) \\
\Leftrightarrow\ & c \text{ is a legal TM-encoding and } \exists\, w \text{ s.t. } cw \in U \\
\Leftrightarrow\ & c \text{ is a legal TM-encoding and } \exists\, w \text{ s.t. } w \in L(M_c) \\
\Leftrightarrow\ & L(M_c) \neq \emptyset.
\end{aligned}
$$

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

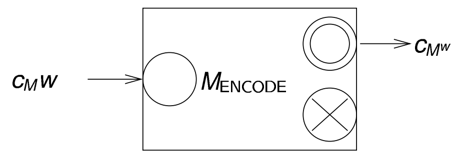
In this construction, inputs are encoded as "program constants" in the Turing machine. Let M be an arbitrary Turing machine whose behaviour under the input $w = a_1 a_2 \ldots a_k$ we wish to examine. Use M^w to denote a machine, that first replaces its input by the string w and then functions as M:



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:

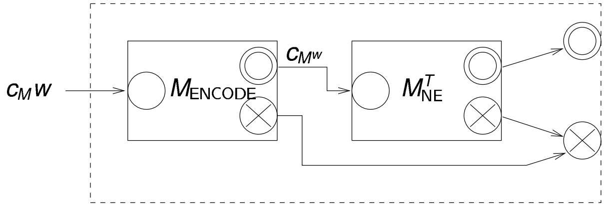
$$
L(M^w) = \begin{cases} \{0,1\}^*, & \text{if } w \in L(M); \\ \emptyset, & \text{if } w \notin L(M). \end{cases}
$$

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



(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 renumbers the states so that the resulting code represents M^w instead of M.

By combining M_{ENCODE} and the hypothetical M_{NE}^T as follows we could form a total recognizer M_U^T for the language U:



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

$$c_M w \in L(M_U^T) \Leftrightarrow c_{M^w} \in L(M_{\text{NE}}^T) = \text{NE} \Leftrightarrow L(M^w) \neq \emptyset \Leftrightarrow w \in L(M).$$

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

Rice's Theorem

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

A property $\mathcal{S}$ is *decidable*, if the set

$$\text{codes}(\mathcal{S}) = \{c \mid L(M_c) \in \mathcal{S}\}$$

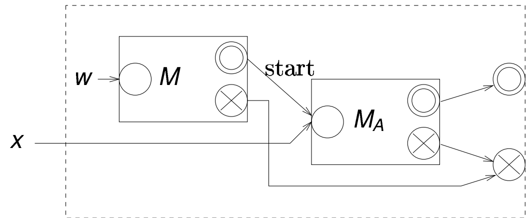
is recursive. In other words: a property is decidable, if by observing a Turing machine we can algorithmically deduce whether the machine has the desired semantic property.

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

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

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

This time, let M_{ENCODE} be a Turing machine that transforms its input $c_M w$ into an encoding of the following Turing machine M^w.



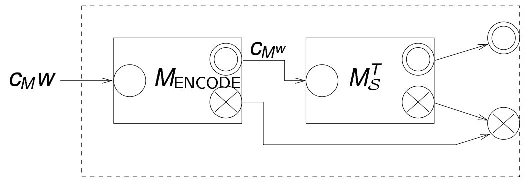
If the input is not of the correct form M_{ENCODE} halts in the rejecting state.

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:n, 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 $L(M_A) \in \mathcal{S}$ and $\emptyset \notin \mathcal{S}$, machine M^w now has property $\mathcal{S}$, if and only if $w \in L(M)$.

Assume then that $\mathcal{S}$ were decidable, i.e., that the language codes($\mathcal{S}$) would have a total recognizer $M_{\mathcal{S}}^{T}$. In style of the previous proof, a total recognizer for U could be combined from M_{ENCODE} and $M_{\mathcal{S}}^{T}$ as follows:



Clearly M_U^T is total, if $M_{\mathcal{S}}^{T}$ is, and

$$c_M w \in L(M_U^T) \Leftrightarrow c_{M^w} \in L(M_{\mathcal{S}}^T) = \mathrm{codes}(\mathcal{S}) \Leftrightarrow L(M^w) \in \mathcal{S} \Leftrightarrow w \in L(M).$$

Since U is not recursive, this is impossible, and hence $\mathcal{S}$ cannot be decidable. □

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 ϕ is valid ("logically true", provable from the axioms of predicate calculus). □

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 $(m_1,\ldots,m_n) \in \mathbf{Z}^n$, for which $P(m_1,\ldots,m_n) = 0$). The problem is undecidable already when $n = 15$ or $\deg(P) = 4$. □

The decidability of certain grammar problems, given two grammars G and G' from a certain level i of the Chomsky hierarchy, and a string w. In the table $D \sim$ "decidable", $U \sim$ "undecidable", $T \sim$ "always true".

Problem: is	Level i: 3	2	1	0
$w \in L(G)$?	D	D	D	U
$L(G) = \emptyset$?	D	D	U	U
$L(G) = \Sigma^*$?	D	U	U	U
$L(G) = L(G')$?	D	U	U	U
$L(G) \subseteq L(G')$?	D	U	U	U
$L(G) \cap L(G') = \emptyset$?	D	U	U	U
$L(G)$ regular?	T	U	U	U
$L(G) \cap L(G')$ of type i?	T	U	T	T
$\overline{L(G)}$ of type i?	T	U	T	U

6.9 Recursive functions

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

$$f_M : \Sigma^* \to \Gamma^*$$

is defined by:

$$f_M(x) = \begin{cases} u, \text{ if } (q_0, \underline{x}) \underset{M}{\vdash}^* (q, u\underline{a}v) \text{ for some } q \in \{q_{\mathrm{acc}}, q_{\mathrm{rej}}\}, av \in \Gamma^*; \\ \text{undefined, otherwise.} \end{cases}$$

A partial function $f : \Sigma^* \to A$ 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

$$\chi_A : \Sigma^* \to \{0,1\}, \qquad \chi_A(x) = \begin{cases} 1, & \text{if } x \in A; \\ 0, & \text{if } x \notin A \end{cases}$$

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. □

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:

$$x \in A \quad \Leftrightarrow \quad f(x) \in B, \qquad \text{for all } x \in \Sigma^*.$$

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. □

Let us denote:

$$\begin{aligned} \text{RE} &= \{\text{the recursively enumerable languages over } \{0,1\}\}; \\ \text{REC} &= \{\text{the recursive languages over } \{0,1\}\}. \end{aligned}$$

A language $A \subseteq \{0,1\}^*$ is *RE-complete*, if
(i) $A \in$ RE and
(ii) $B \leq_m A$ for all $B \in$ RE.

Theorem 6.17 The language U is RE-complete.

Proof. We know that $U \in$ RE. Let $B = L(M_B)$ be an arbitrary language in RE. Now B may be reduced to U:hun by the function $f(x) = c_{M_B}x$. This function is clearly recursive, and has the property

$$x \in B = L(M_B) \quad \Leftrightarrow \quad f(x) = c_{M_B}x \in U. \quad \square$$

Lemma 6.18 Let A be an RE-complete language, $B \in$ RE and $A \leq_m B$. Then also B is RE-complete. □

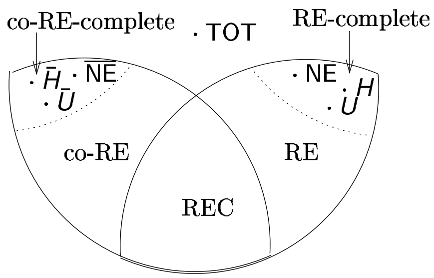
From Rice's Theorem it follows that among other all problems, where something is to be deduced about the language a Turing machine recognizes by looking at the code of the Turing machine are RE-complete. In general it appears that all "natural" recursively enumerable, non-recursive languages are RE-complete.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

Theorem 6.19 (E. Post 1944) In RE − REC there are languages that are not RE-complete. □

Since the class RE is not closed under complementation, it has the natural complement class co-RE $= \{\bar{A} \mid A \in \text{RE}\}$.

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

In co-RE the concept of a complete language can be defined similarly as in RE: a language $A \subseteq \{0,1\}^*$ is co-RE-complete, if $A \in$ co-RE and $B \leq_m A$ for all $B \in$ co-RE. It is easy to see that a language A is co-RE-complete, if and only if the language $\bar{A}$ is 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 RE nor in co-RE. $\square$

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

$$x \in A \quad \Leftrightarrow \quad f(x) \in B, \qquad \text{for all } x \in \Sigma^*.$$

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

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 $\omega \to \omega'$.

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

$$\gamma \underset{G}{\Rightarrow} \gamma'$$

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

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

$$\gamma \underset{G}{\Rightarrow}^* \gamma'$$

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

$$\gamma = \gamma_0 \underset{G}{\Rightarrow} \gamma_1 \underset{G}{\Rightarrow} \ldots \underset{G}{\Rightarrow} \gamma_n = \gamma'.$$

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 $S \underset{G}{\Rightarrow}^* \gamma$. 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

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

Example. An unrestricted grammar for the non-context-free language $\{a^k b^k c^k \mid k \geq 0\}$.

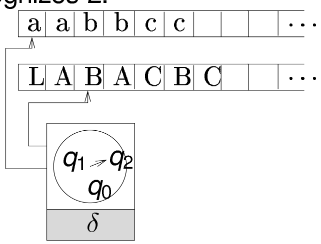
$$
\begin{array}{rcl@{\qquad\qquad}rcl}
S & \to & LT \mid \varepsilon & & & \\
T & \to & ABCT \mid ABC & aA & \to & aa \\
BA & \to & AB & aB & \to & ab \\
CB & \to & BC & bB & \to & bb \\
CA & \to & AC & bC & \to & bc \\
LA & \to & a & cC & \to & cc.
\end{array}
$$

Example: derivation of the sentence *aabbcc*:

$$
\begin{array}{llllllll}
\underline{S} & \Rightarrow & L\underline{T} \Rightarrow LABC\underline{T} & \Rightarrow & LABC\underline{CA}BC & \Rightarrow & LAB\underline{A}CBC \\
& \Rightarrow & LAAB\underline{CB}C & \Rightarrow & \underline{LA}ABBCC & \Rightarrow & \underline{aA}BBCC \\
& \Rightarrow & a\underline{aB}BCC & \Rightarrow & aa\underline{bB}CC & \Rightarrow & aab\underline{bC}C \\
& \Rightarrow & aabb\underline{cC} & \Rightarrow & aabbcc. & &
\end{array}
$$

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 $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ 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 $(q, u\underline{a}v)$ of M is represented as the string $[uqav]$. From the transition function of M productions of G_M are formed so that

$$[uqav] \underset{G_M}{\Rightarrow} [u'q'a'v'] \quad \text{iff} \quad (q, uav) \underset{M}{\vdash} (q', u'a'v').$$

Thus M accepts x if and only if

$$[q_0x] \underset{G_M}{\Rightarrow^*} [uq_{acc}v]$$

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

All in all there will be three groups of productions in G_M:

1. Productions with which any string of the form $x[q_0x]$ can be derived from S, where $x \in \Sigma^*$ and [, q_0 ja] are non-terminals of G_M.

2. Productions with which the string $[uq_{acc}v]$ can be derived from $[q_0x]$ if and only if M accepts x.

3. Productions with which a string of the form $[uq_{acc}v]$ can be transformed to the empty string.

Producing a string x that is in $L(M)$ proceeds then as follows:

$$S \overset{(1)}{\Rightarrow^*} x[q_0x] \overset{(2)}{\Rightarrow^*} x[uq_{acc}v] \overset{(3)}{\Rightarrow^*} x.$$

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

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

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

1. Producing the initial configuration

$$\begin{array}{lcll} S & \to & T[q_0] & \\ T & \to & \varepsilon & \\ T & \to & aTA_a & (a \in \Sigma) \\ A_a[q_0 & \to & [q_0A_a & (a \in \Sigma) \\ A_ab & \to & bA_a & (a, b \in \Sigma) \\ A_a] & \to & a] & (a \in \Sigma) \end{array}$$

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

Transitions:	*Productions:*
$\delta(q, a) = (q', b, R)$	$qa \to bq'$
$\delta(q, a) = (q', b, L)$	$cqa \to q'cb$
$\delta(q, >) = (q', >, R)$	$q[\to [q'$
$\delta(q, <) = (q', b, R)$	$q] \to bq']$
$\delta(q, <) = (q', b, L)$	$cq] \to q'cb]$
$\delta(q, <) = (q', <, L)$	$cq] \to q'c]$

3. Cleaning up the final configuration:

$$\begin{array}{lcll} q_{\text{acc}} & \to & E_L E_R & \\ q_{\text{acc}}[& \to & E_R & \\ aE_L & \to & E_L & (a \in \Gamma) \\ [E_L & \to & \varepsilon & \\ E_R a & \to & E_R & (a \in \Gamma) \\ E_R] & \to & \varepsilon & \end{array}$$

□

Context sensitive grammars

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

Additionally, if the grammar contains the rule $S \to \varepsilon$, 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 $S \to \varepsilon$ and $\alpha A \beta \to \alpha \omega \beta$, where A is a non-terminal and $\omega \neq \varepsilon$. (The rule $A \to \omega$ applied "in the context" α_β.)

Theorem 5.3 A formal language L is context sensitive if and only if it can be recognized by a nondetermistic 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.