Decision problems: decidable and undecidable

We want to show that certain problems can not be solved by a computer. By problems we mean **decision problems**, *i.e.*, problems that have a yes/no answer. For example, questions like

- "Is a given regular language infinite ?",
- "Is a given grammar ambiguous ?", or
- "Does a given quadratic equation have a real number solution ?"

are decision problems.

Decision problems: decidable and undecidable

We want to show that certain problems can not be solved by a computer. By problems we mean **decision problems**, *i.e.*, problems that have a yes/no answer. For example, questions like

- "Is a given regular language infinite ?",
- "Is a given grammar ambiguous ?", or
- "Does a given quadratic equation have a real number solution ?"

are decision problems.

When the input in the problem is fixed, we get an **instance** of the problem. For example,

$$a^*bb + ba,$$
$$S \longrightarrow SS \mid a,$$
$$x^2 + 4x + 5 = 0,$$

are instances of the three decision problems above. For each instance we have a correct yes/no answer. ('yes', 'yes', 'no', respectively, in our examples.)

Instances with 'yes' answer are **positive** instances, while 'no' answer means that the instance is **negative**.

A decision problem is **decidable** if there exists an algorithm that, when given an instance as input, returns the correct yes/no answer.

The algorithm has to work for all instances correctly. The computation may take any finite amount of time: a year, billion years, gazillion years. But for every instance the computation halts at some time and gives the correct yes/no answer.

A problem is called **undecidable** if no such algorithm exists. We'll see that certain important and natural decision problems are undecidable.

It does not make sense to talk about decision problems that have only one instance: such problems are of course decidable since there is a trivial algorithm that simply writes out 'yes' or writes out 'no'. (We may not know which of these two algoritms is the correct one.)

For this reason questions like "Is $P = NP$?" or "Is the Riemann-hypothesis true" are trivial to us: they don't have different instances.

In order to use formal language theory to show decidability or undecidability of a particular decision problem, we **encode** instances to that problem as words over some alphabet.

(When typing an instance into the computer we are encoding it as a word over the binary alphabet $\{0, 1\}$. It is up to us what encoding we use. As long as two encodings can be effectively converted into each other, the problem cannot be decidable in one presentation and undecidable in the other one.)

In order to use formal language theory to show decidability or undecidability of a particular decision problem, we **encode** instances to that problem as words over some alphabet.

(When typing an instance into the computer we are encoding it as a word over the binary alphabet $\{0, 1\}$. It is up to us what encoding we use. As long as two encodings can be effectively converted into each other, the problem cannot be decidable in one presentation and undecidable in the other one.)

Example. we may decide to encode quadratic equations using alphabet

$$\{x, +, -, *, =, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

in the natural manner. Then for example a word

$$x * x + 10 * x - 100 = 0$$

is a coding of an instance to our third sample decision problem.

The **language** $L_{\mathcal{P}}$ corresponding to a particular decision problem $\mathcal{P}$ consists of words representing positive instances.

Example. The language $L_{\mathcal{P}}$ of our quadratic equation example contains all words representing quadratic equations with a real solution:

$$
\begin{aligned}
&x * x = 0, \\
&x * x - 1 = 0, \\
&x * x - 2 * x + 1 = 0, \\
&\ldots
\end{aligned}
$$

are all words in the language. $L_{\mathcal{P}}$ is of course infinite. Is it regular ? Is it context-free ? Hardly (use pumping lemma . . .). We need a larger family of languages that contains all languages $L_{\mathcal{P}}$ representing decidable problems $\mathcal{P}$.

In parts I and II we have investigated regular and context-free languages. They are two very natural language classes but – as we have seen – there are many quite simple languages outside of them, e.g. $\{a^n b^n c^n \mid n \geq 1\}$.

In part III we meet two new language families: **recursive** and **recursively enumerable** languages. Using these language families we can exhibit certain limitations on what computers can do.

There are many possible ways to define recursive and recursively enumerable languages, just like in case of regular and context-free languages. We will formally define them using accepting devices called **Turing machines**.

The **membership problem** of a language L is the following decision problem:

Membership in $L \subseteq \Sigma^*$
Instance: Word $w \in \Sigma^*$
Problem: Is $w \in L$?

Before Turing machines, here is an **informal definition of recursive languages**. Keeping this informal interpretation in mind will be very useful.

A language L is recursive

if and only if

The membership problem of L is decidable.

The **membership problem** of a language L is the following decision problem:

Membership in $L \subseteq \Sigma^*$
  Instance: Word $w \in \Sigma^*$
  Problem: Is $w \in L$?

Before Turing machines, here is an **informal definition of recursive languages**. Keeping this informal interpretation in mind will be very useful.

A language L is recursive if and only if The membership problem of L is decidable.

In other words, L is recursive if one can write a computer program that, when given any word w as input, determines (after some finite amount of time) whether w is in L or not. The program has to work correctly for all input words. We assume that the computing resources are not a problem: We have a big enough computer, so that we do not run out of memory.

All regular and context-free languages are recursive. (the CYK-algorithm can be used to determine if a given w belongs to $L(G)$.) In addition, for example following languages are recursive:

$\{a^n b^n c^n \mid n \geq 1\}$
$\{a^p \mid p \text{ is a prime number }\}$
$\{w \mid w \text{ is an encoding of a quadratic equation with a real solution }\}$

A connection between decision problems and recursive languages:

A decision problem $\mathcal{P}$ is decidable
if and only if
The language $L_{\mathcal{P}}$ is recursive.

This is trivial: The language $L_{\mathcal{P}}$ is recursive if and only if there exists an algorithm for finding out if a given word belongs to $L_{\mathcal{P}}$. But the word w belongs to $L_{\mathcal{P}}$ if and only if the answer to the instance encoded as w is positive.

The language family of recursively enumerable languages is larger than the family of recursive languages. It corresponds to the notion of a **semi-algorithm**.

The difference between a semi-algorithm and algorithm is that the semi-algorithm does not need to stop on all inputs. A semi-algorithm does stop on instances with 'yes' -answer, but on instances with 'no'-answer it may enter an infinite, non-halting computation. If the semi-algorithm gives an answer, the answer is always correct. But on some 'no'-instances it may not give any answer at all, but continue computing for ever.

A problem is called **semi-decidable** if there exists a semi-algorithm for it.

An informal definition of recursively enumerable (r.e.) languages is:

A language L is recursively enumerable if and only if The membership problem of L is semi-decidable.

So the relationship between semi-decidable problems and recursively enumerable languages is

> A decision problem $\mathcal{P}$ is semi-decidable
>
> if and only if
>
> The language $L_{\mathcal{P}}$ is recursively enumerable.

All recursive languages are of course also r.e. (Every algorithm is also a semi-algorithm.) But there are r.e. languages that are not recursive. In other words, there are decision problems that are semi-decidable but not decidable.

It is not very surprising that there are languages that are not recursive or even r.e. We can reason as follows: The number of different r.e. languages is countably infinite. This follows from the fact that the number of different algorithms (computer programs) is countably infinite.

But the number of different languages is uncountably infinite, i.e. there are **more** languages than there are r.e. languages.

What is more surprising is that we can actually find individual languages that correspond to very natural decision problems and are non-recursive.

Turing machines

In order to formally prove a language recursive, recursively enumerable or non-recursive we must have a precise definition of an algorithm. We do this using Turing machines: We say a language is **r.e.** iff it is accepted by a Turing machine, and it is **recursive** iff it is accepted by a Turing machine that halts on every input.

Turing machines may seem like very restricted models of computers, but they are in fact as powerful as any other model.

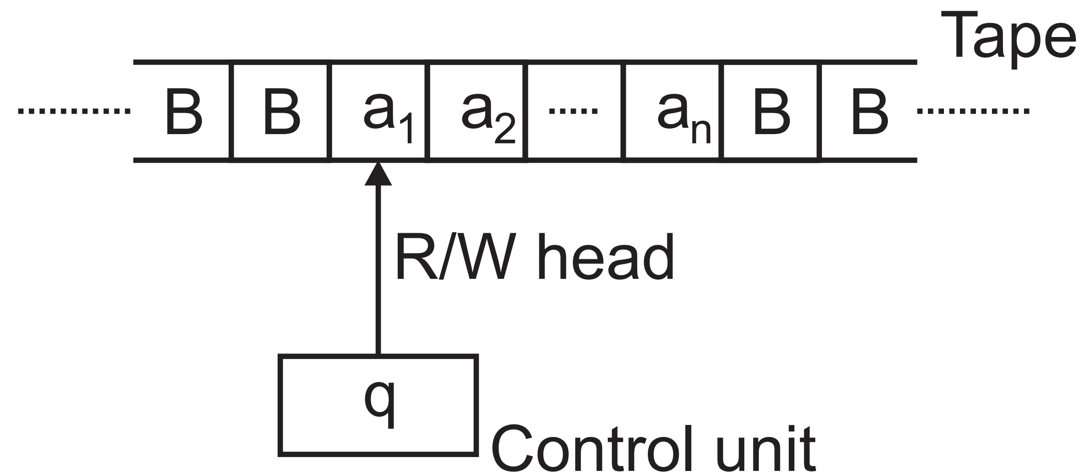
Church's thesis is the claim that Turing machines capture the full power of any effective method of computation. One cannot of course **prove** that nobody ever invents a totally different, more powerful computer architecture. But all evidence indicates that this is not possible, as long as we require that the computer executes mechanically a finitely describable program on discrete steps.

A **Turing machine (TM)** has a finite state control similar to PDA. Instead of input tape and stack, the machine has only one **infinite tape** that is both the input tape and a "work" tape.

The machine has a **read/write head** that may move left or right on the tape. Initially, the input is written on the tape. At each time step the machine reads the tape symbol under its read/write head, and depending on the tape symbol and the state of the control unit, the machine may

1. Change its state,
2. Replace the symbol on the tape under the read/write head by another tape symbol, and
3. move the R/W head left or right on the tape.

The tape is infinite to the left and to the right.



Initially the input word is written on the tape in such a way that the read-write head scans the first letter of the input. All other tape locations contain a special **blank symbol** B. The input is accepted if and only if the machine eventually enters the **final state**.

Formally, a (deterministic) Turing machine is

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$$

where

- Q is the finite **state set** of the control unit.
- Σ is the **input alphabet**.
- Γ is the **tape alphabet** containing all allowed symbols that may be on the tape. Especially $\Sigma \subset \Gamma$ since initially the input is written on the tape. We assume that $Q \cap \Gamma = \emptyset$ so that there is no confusion between states and tape letters.
- δ is a **transition function**, described below in detail.
- $q_0 \in Q$ is the **initial state**.
- $B \in \Gamma \setminus \Sigma$ is a special **blank symbol**. It is not part of the input alphabet Σ.
- $f \in Q$ is the **final state**.

The transition function δ is a (partial) mapping from the set

$$(Q \setminus \{f\}) \times \Gamma$$

into the set

$$Q \times \Gamma \times \{L, R\}.$$

The mapping is partial: it may be undefined for some arguments, in which case the machine has no possible next move, and it halts. In particular, there is no transition from the final state f.

The transition function δ is a (partial) mapping from the set

$$(Q \setminus \{f\}) \times \Gamma$$

into the set

$$Q \times \Gamma \times \{L, R\}.$$

The mapping is partial: it may be undefined for some arguments, in which case the machine has no possible next move, and it halts. In particular, there is no transition from the final state f.

$$\delta(q, X) = (p, Y, L)$$

means that in state q, scanning tape symbol X, the machine changes its state to p, replaces X by Y on the tape, and moves the R/W head one cell to the left on the tape.

$$\delta(q, X) = (p, Y, R)$$

is defined analogously (only the R/W head moves to the right.)

Initially,

- the input word is written on the tape, with all other cells containing the blank symbol B,
- the machine is in state q_0, and
- the R/W head is positioned on the leftmost letter of the input.

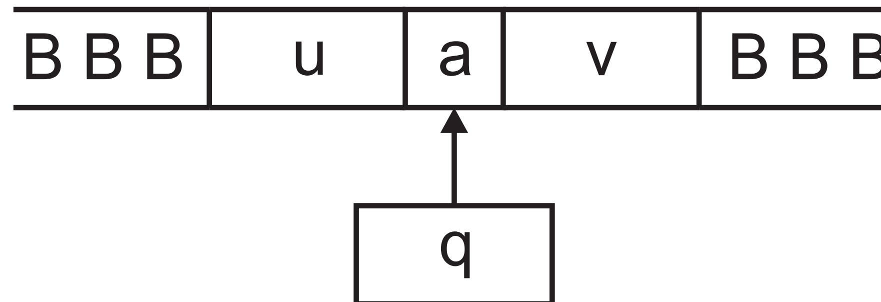
If after some moves the machine enters the final state f the input word is accepted, otherwise it is rejected.

(A word is rejected if the machine never halts, or if it halts in a non-final state when there is no next move.)

Let us define **instantaneous descriptions** (IDs) of Turing machines: An ID is a word

$$u\ q\ a\ v$$

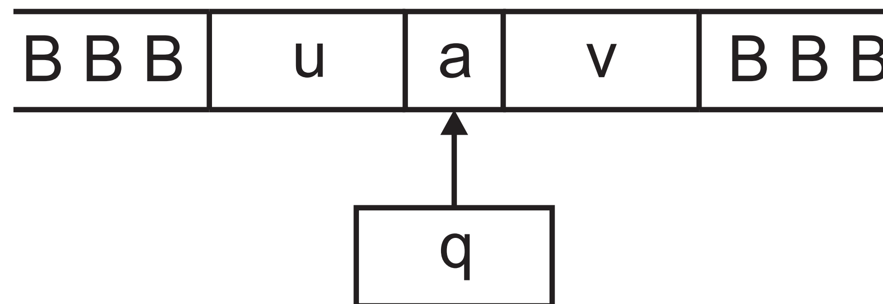
where $q \in Q$ is the state of the control unit, $u, v \in \Gamma^*$ are the content of the tape to the left and to the right of the R/W head, respectively, until the last non-blank symbols, and $a \in \Gamma$ is the symbol currently scanned by the R/W head:



Let us define **instantaneous descriptions** (IDs) of Turing machines: An ID is a word

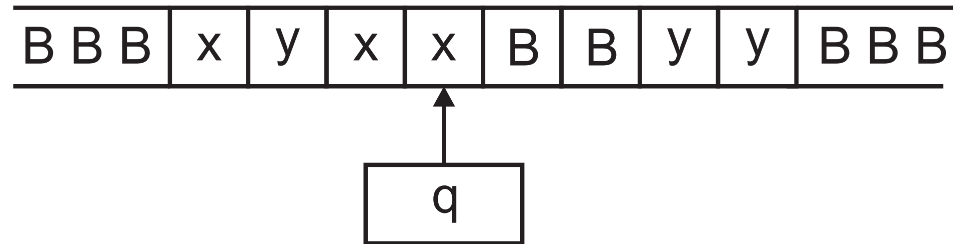
$$u\ q\ a\ v$$

where $q \in Q$ is the state of the control unit, $u, v \in \Gamma^*$ are the content of the tape to the left and to the right of the R/W head, respectively, until the last non-blank symbols, and $a \in \Gamma$ is the symbol currently scanned by the R/W head:

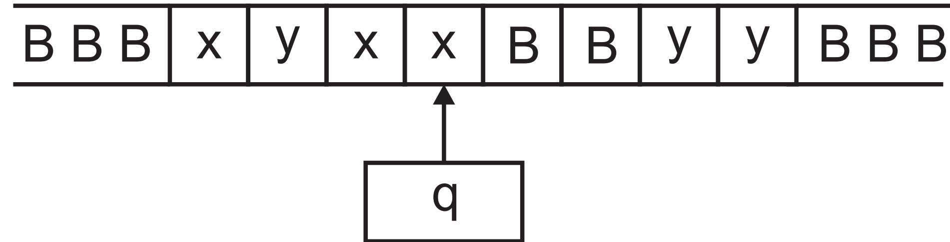


So: The first (last) letter of u (or v, respectively) is not B, because the leading and trailing B's are trimmed away from u and v, respectively. Also that the current state q can be uniquely identified from the word $u\ q\ a\ v$ because we required alphabets Q and Γ to be disjoint.

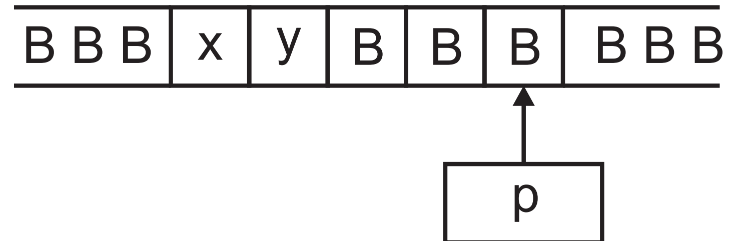
Example. $xyxqxBByy$ (where $q \in Q$) represents the situation



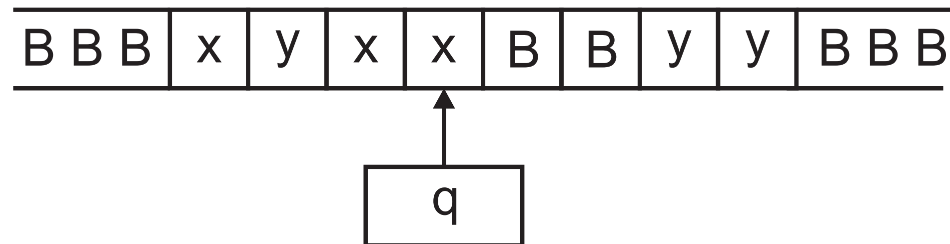
Example. $xyxqxBByy$ (where $q \in Q$) represents the situation



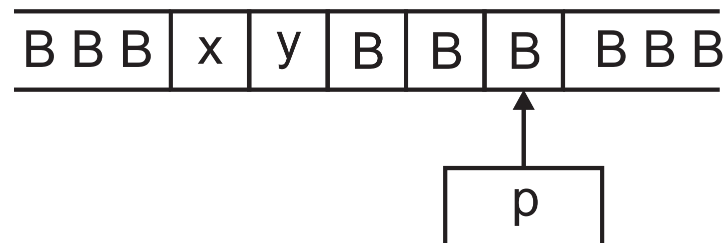
$xyBBpB$ (where $p \in Q$) represents



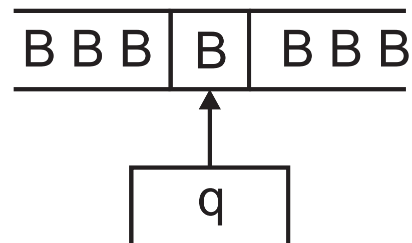
Example. $xyxqxBByy$ (where $q \in Q$) represents the situation



$xyBBpB$ (where $p \in Q$) represents



qB represents



Example. On the other hand, BqB and $xyBqBxBB$ are not valid ID's since the surrounding B's have not been removed.

Example. On the other hand, BqB and $xyBqBxBB$ are not valid ID's since the surrounding B's have not been removed.

Precisely: Valid ID's are words of the set

$$(\{\varepsilon\} \cup (\Gamma \setminus \{B\})\Gamma^*) \; Q \; \Gamma \; (\Gamma^*(\Gamma \setminus \{B\}) \cup \{\varepsilon\}).$$

Let us define **moves** of TM. Let the current ID be

$$\alpha = u \; q \; a \; v.$$

If β is the next ID we denote $\alpha \vdash \beta$ and say that β results from α by one move.

1. Assume first that

$$\delta(q, a) = (p, b, L).$$

- If $u = \varepsilon$ then the next ID is $\beta = pBbv$, except that possible trailing B's are removed from bv.
- If $u = u'c$ for some $c \in \Gamma$ then the next ID is $\beta = u'pcbv$, except that possible trailing B's are removed from bv.

Let us define **moves** of TM. Let the current ID be

$$\alpha = u\ q\ a\ v.$$

If β is the next ID we denote $\alpha \vdash \beta$ and say that β results from α by one move.

2. Assume then that

$$\delta(q, a) = (p, b, R).$$

- If $v = \varepsilon$ then the next ID is $\beta = ubpB$ except that possible leading B's are removed from ub.
- If $v \neq \varepsilon$ then the next ID is $u = ubpv$, except that possible leading B's are removed from ub.

Let us define **moves** of TM. Let the current ID be

$$\alpha = u\ q\ a\ v.$$

If β is the next ID we denote $\alpha \vdash \beta$ and say that β results from α by one move.

3. If $\delta(q, a)$ is undefined then no move from α is possible, and α is a **halting** ID. If $q = f$ then α is an **accepting** ID.

Our TM model is **deterministic**, which means that for each α there is at most one β such that $\alpha \vdash \beta$.

Our TM model is **deterministic**, which means that for each α there is at most one β such that $\alpha \vdash \beta$.

As usual, we write

$$\alpha \vdash^* \beta$$

if the TM changes α into β in any number of moves (including 0 in which case $\alpha = \beta$).

We denote

$$\alpha \vdash^+ \beta$$

if the TM changes α into β using at least one move.

We denote

$$\alpha \vdash^i \beta$$

if the TM changes α into β in exactly i moves.

For any $w \in \Sigma^*$ we define the corresponding **initial ID**

$$\iota_w = \begin{cases} q_0 w, & \text{if } w \neq \varepsilon, \\ q_0 B, & \text{if } w = \varepsilon. \end{cases}$$

For any $w \in \Sigma^*$ we define the corresponding **initial ID**

$$\iota_w = \begin{cases} q_0 w, & \text{if } w \neq \varepsilon, \\ q_0 B, & \text{if } w = \varepsilon. \end{cases}$$

The language **recognized** (or accepted) by the TM M is

$$L(M) = \{w \mid w \in \Sigma^* \text{ and } \iota_w \vdash^* ufv \text{ for some } u, v \in \Gamma^* \}.$$

Example. If $\delta(q, B) = (p, c, L)$ then

$$\begin{array}{ll} ab\ q\ Ba & \vdash \\ aB\ q\ B & \vdash \\ q\ Bb & \vdash \end{array}$$

Example. If $\delta(q, B) = (p, c, L)$ then

$$
\begin{array}{lcl}
ab\ q\ Ba & \vdash & a\ p\ bca \\
aB\ q\ B & \vdash & a\ p\ Bc \\
q\ Bb & \vdash & p\ Bcb
\end{array}
$$

Example. If $\delta(q, B) = (p, c, L)$ then

$$ab\ q\ Ba \vdash a\ p\ bca$$
$$aB\ q\ B \vdash a\ p\ Bc$$
$$q\ Bb \vdash p\ Bcb$$

If $\delta(q, B) = (p, B, L)$ then

$$aB\ q\ B \vdash$$
$$ab\ q\ B \vdash$$

Example. If $\delta(q, B) = (p, c, L)$ then

$$
\begin{array}{ll}
ab\ q\ Ba \vdash & a\ p\ bca \\
aB\ q\ B \vdash & a\ p\ Bc \\
q\ Bb \vdash & p\ Bcb
\end{array}
$$

If $\delta(q, B) = (p, B, L)$ then

$$
\begin{array}{ll}
aB\ q\ B \vdash & a\ p\ B \\
ab\ q\ B \vdash & a\ p\ b
\end{array}
$$

If $\delta(q, B) = (p, c, R)$ then

$$aB\ q\ B \vdash$$

Example. If $\delta(q, B) = (p, c, L)$ then

$$\begin{array}{lcl} ab\ q\ Ba & \vdash & a\ p\ bca \\ aB\ q\ B & \vdash & a\ p\ Bc \\ q\ Bb & \vdash & p\ Bcb \end{array}$$

If $\delta(q, B) = (p, B, L)$ then

$$\begin{array}{lcl} aB\ q\ B & \vdash & a\ p\ B \\ ab\ q\ B & \vdash & a\ p\ b \end{array}$$

If $\delta(q, B) = (p, c, R)$ then

$$aB\ q\ B \vdash aBc\ p\ B$$

Example. Let us design a TM M that recognizes the language

$$L(M) = \{w \mid w \in (a+b)^* \text{ and } w \text{ is a palindrome } \}.$$

We have

$$M = (\{q_0, q_F, q_L, q_a, q_a', q_b, q_b'\}, \{a, b\}, \{a, b, B\}, \delta, q_0, B, q_F),$$

where δ is is defined below.

Example. Let us design a TM M that recognizes the language

$$L(M) = \{w \mid w \in (a+b)^* \text{ and } w \text{ is a palindrome } \}.$$

We have

$$M = (\{q_0, q_F, q_L, q_a, q'_a, q_b, q'_b\}, \{a, b\}, \{a, b, B\}, \delta, q_0, B, q_F),$$

where δ is is defined below.

The idea is that the machine reads the first input letter from the tape, remembers it in the control unit (q_a and q_b), finds the last letter on the tape and compares it with the first letter. If they are identical, they are erased (replaced by B), and the process is repeated. If the whole input word gets erased (or if there remains only one input letter if the original word has odd length) the word was a palindrome, and the machine accepts it.

The transition function δ is defined as follows:

$$
\begin{array}{llll}
\delta(q_0, B) & = & (q_F, B, R) & \text{Remaining input is empty} \\
\\
\delta(q_0, a) & = & (q_a, B, R) & \text{Erase first input letter } a \\
\delta(q_a, a) & = & (q_a, a, R) & \\
\delta(q_a, b) & = & (q_a, b, R) & \\
\delta(q_a, B) & = & (q'_a, B, L) & \text{End of input found} \\
\delta(q'_a, B) & = & (q_F, B, R) & \text{Remaining input was empty} \\
\delta(q'_a, a) & = & (q_L, B, L) & \text{Erase last input letter } a \\
\\
\delta(q_0, b) & = & (q_b, B, R) & \text{Erase first input letter } b \\
\delta(q_b, a) & = & (q_b, a, R) & \\
\delta(q_b, b) & = & (q_b, b, R) & \\
\delta(q_b, B) & = & (q'_b, B, L) & \text{End of input found} \\
\delta(q'_b, B) & = & (q_F, B, R) & \text{Remaining input was empty} \\
\delta(q'_b, b) & = & (q_L, B, L) & \text{Erase last input letter } b \\
\\
\delta(q_L, a) & = & (q_L, a, L) & \text{move back to beginning} \\
\delta(q_L, b) & = & (q_L, b, L) & \\
\delta(q_L, B) & = & (q_0, B, R) & \text{beginning of input found}
\end{array}
$$

The accepting computation for the word *abba*:

$q_0 \ abba \vdash$

The accepting computation for the word $abba$:

$$q_0\ abba \vdash q_a\ bba \vdash b\ q_a\ ba \vdash bb\ q_a\ a \vdash bba\ q_a\ B \vdash bb\ q'_a\ a \vdash b\ q_L\ b \vdash q_L\ bb$$

$$\vdash q_L\ Bbb \vdash\vdash q_0\ bb \vdash q_b\ b \vdash b\ q_b\ B \vdash q'_b\ b \vdash q_L\ B \vdash q_0\ B \vdash q_F\ B$$

The accepting computation for the word $abba$:

$$q_0\ abba \vdash q_a\ bba \vdash b\ q_a\ ba \vdash bb\ q_a\ a \vdash bba\ q_a\ B \vdash bb\ q_a'\ a \vdash b\ q_L\ b \vdash q_L\ bb$$

$$\vdash q_L\ Bbb \vdash\vdash q_0\ bb \vdash q_b\ b \vdash b\ q_b\ B \vdash q_b'\ b \vdash q_L\ B \vdash q_0\ B \vdash q_F\ B$$

The accepting computation for the word aba:

$$q_0\ aba \vdash$$

The accepting computation for the word $abba$:

$$q_0\ abba \vdash q_a\ bba \vdash b\ q_a\ ba \vdash bb\ q_a\ a \vdash bba\ q_a\ B \vdash bb\ q_a'\ a \vdash b\ q_L\ b \vdash q_L\ bb$$

$$\vdash q_L\ Bbb \vdash\vdash q_0\ bb \vdash q_b\ b \vdash b\ q_b\ B \vdash q_b'\ b \vdash q_L\ B \vdash q_0\ B \vdash q_F\ B$$

The accepting computation for the word aba:

$$q_0\ aba \vdash q_a\ ba \vdash b\ q_a\ a \vdash ba\ q_a\ B \vdash b\ q_a'\ a \vdash q_L\ b \vdash q_L\ Bb \vdash q_0\ b \vdash q_b\ B$$

$$\vdash q_b'\ B \vdash q_F\ B$$

The accepting computation for the word $abba$:

$$q_0\ abba \vdash q_a\ bba \vdash b\ q_a\ ba \vdash bb\ q_a\ a \vdash bba\ q_a\ B \vdash bb\ q_a'\ a \vdash b\ q_L\ b \vdash q_L\ bb$$

$$\vdash q_L\ Bbb \vdash\vdash q_0\ bb \vdash q_b\ b \vdash b\ q_b\ B \vdash q_b'\ b \vdash q_L\ B \vdash q_0\ B \vdash q_F\ B$$

The accepting computation for the word aba:

$$q_0\ aba \vdash q_a\ ba \vdash b\ q_a\ a \vdash ba\ q_a\ B \vdash b\ q_a'\ a \vdash q_L\ b \vdash q_L\ Bb \vdash q_0\ b \vdash q_b\ B$$

$$\vdash q_b'\ B \vdash q_F\ B$$

The word abb is not accepted because the computation halts in a non-accepting ID:

$$q_0\ abb \vdash$$

The accepting computation for the word $abba$:

$$q_0\ abba \vdash q_a\ bba \vdash b\ q_a\ ba \vdash bb\ q_a\ a \vdash bba\ q_a\ B \vdash bb\ q'_a\ a \vdash b\ q_L\ b \vdash q_L\ bb$$

$$\vdash q_L\ Bbb \vdash\vdash q_0\ bb \vdash q_b\ b \vdash b\ q_b\ B \vdash q'_b\ b \vdash q_L\ B \vdash q_0\ B \vdash q_F\ B$$

The accepting computation for the word aba:

$$q_0\ aba \vdash q_a\ ba \vdash b\ q_a\ a \vdash ba\ q_a\ B \vdash b\ q'_a\ a \vdash q_L\ b \vdash q_L\ Bb \vdash q_0\ b \vdash q_b\ B$$

$$\vdash q'_b\ B \vdash q_F\ B$$

The word abb is not accepted because the computation halts in a non-accepting ID:

$$q_0\ abb \vdash q_a\ bb \vdash b\ q_a\ b \vdash bb\ q_a\ B \vdash b\ q'_a\ b$$

A language is **recursively enumerable** (**r.e.**) if it is recognized by a Turing machine. A language is **recursive** if it is recognized by a Turing machine that halts on all inputs. Of course, every recursive language is also r.e.

This is our precise mathematical definition of recursive and r.e. languages.

(It coincides with the informal definition given before.)

Our sample TM halts on every input, so the palindrome language L is recursive. In fact, every context-free language is recursive. (We can implement the CYK algorithm on a Turing machine!)

There are some "programming techniques" when designing Turing machines.

1. **Storing a tape symbol in the finite control**. We can build a TM whose states are pairs $[q, X]$ where q is an original state, and X is a tape symbol. The first component is used as before, the second component can be used in remembering a particular tape symbol.

There are some “programming techniques” when designing Turing machines.

1. **Storing a tape symbol in the finite control**. We can build a TM whose states are pairs $[q, X]$ where q is an original state, and X is a tape symbol. The first component is used as before, the second component can be used in remembering a particular tape symbol.

We used this technique in the sample Turing machine that recognizes palindromes: The first letter of the input was stored in the state (state q_a or q_b), and the machine moved to the end of the input and verified that the letter there was identical to the first letter.

Example. A TM that recognizes the language

$$L = ab^* + ba^*$$

The machine reads the first symbol, remembers it in the finite control, and checks that the same symbol does not appear anywhere else in the input word.

$$\begin{array}{lcl}
\delta(q_0, a) & = & ([q, a], a, R) \\
\delta(q_0, b) & = & ([q, b], b, R) \\
\delta([q, a], b) & = & ([q, a], b, R) \\
\delta([q, b], a) & = & ([q, b], a, R) \\
\delta([q, a], B) & = & (q_F, B, R) \\
\delta([q, b], B) & = & (q_F, B, R)
\end{array}$$

(Compare this with a finite automaton that recognizes L.)

2. **Multiple tracks**. Sometimes it is useful to imagine that the tape consists of multiple “tracks”. We can store different intermediate information on different tracks:

BBB	B	a	b	a	B	B	a	B	BBB
BBB	B	B	B	B	a	a	a	a	BBB
BBB	a	a	a	a	B	B	B	B	BBB

q

The tape alphabet is then

$$\Gamma_1 \times \Gamma_2 \times \cdots \times \Gamma_n$$

where Γ_i is the alphabet of symbols on the i'th track.

3. **Checking off symbols**. This is equivalent to having a second track where we can place blank B or symbol $\surd$. The tick mark can be conveniently used in remembering which letters of the input have been already processed. It is useful in recognizing languages where we have to count letters, for example.

3. **Checking off symbols**. This is equivalent to having a second track where we can place blank B or symbol $\surd$. The tick mark can be conveniently used in remembering which letters of the input have been already processed. It is useful in recognizing languages where we have to count letters, for example.

Example. Let

$$L = \{ww \mid w \in (a+b)^*\}.$$

We first use the tick mark to find the center of the input word: Mark alternatively the first and last unmarked letters, one-by-one. The last letter to be marked is in the center. So we know where the second w should start.

Using the "Storage in the finite control" technique, check that the letters in the first half and the second half are identical.

Example. A sketch of a TM with a **three track** tape that recognizes

$$L = \{a^p \mid p \text{ is a prime number } \}$$

Initially the input is written on the first track and the other two tracks contain B's. (This means we identify a with $[a, B, B]$ and B with $[B, B, B]$.)

The machine operates as follows. It starts by placing two a's on the second track. Then it repeats the following:

1. Copy the content of the first track to the third track.
2. Subtract the number on the second track from the third track as many times as possible. If the third track becomes empty, halt in a non-accepting state. (The number on the first track was divisible by the number on the second track.)
3. Increase the number on the second track by one. If the number becomes the same as the number on the first track halt in an accepting state. Else go back to step 1.

4. **Shifting over**. This means adding an new cell at the current location of the tape. This is done by shifting all symbols one position to the right by scanning the tape from the current position to the right, remembering the content of the previous cell in the finite control, and writing it to the next cell on the right. Once the rightmost non-blank symbol is reached the machine can return to the new vacant cell that was introduced.

(In order to recognize the rightmost non-blank symbol, it is convenient to introduce an end-of-tape symbol that is written in the first cell after the last non-blank symbol.)

5. **Subroutines**. We can use subroutines in TM in the same way as they are used in normal programming languages. A subroutine uses its own set of states, including its own "initial state" q_0' and a return state q_r.

To call a subroutine, the calling TM simply changes the state to q_0', and makes sure the read-write head is positioned on the leftmost symbol of the "input" to the subroutine.

Constructing TM to perform specific tasks can be quite complicated. Even to recognize some simple languages may require many states and complicated constructions. However, TM are powerful enough to be able to simulate any computer program.

The claim that Turing machines can compute everything that is computable using any model of computation is known as **Church's thesis**. Since the thesis talks about *any* model of computation, it can never be proved. But there is strong evidence supporting the thesis: So far, TM have been able to simulate any other model of computing anyone has invented (λ-calculus, Markov-algorithms, Post systems, recursive functions etc.)

Example. Let us see how a Turing machine would simulate a conventional computer architecture. The tape contains all data the computer has in its memory.

The data can be organized for example in such a way that the word v_i in the memory location i of the computer is stored on the tape as

$$\#0^i * v_i\#$$

The contents of the registers of the CPU are stored on their own tracks on the tape.

To execute the next instruction, the TM finds the memory location addressed by the **Program Counter** register. In order to do that it goes through all memory locations one by one and – using the tick marks – counts if the address i is the same as the content of the Program Counter register. When it finds the correct memory location i, it reads the **instruction** v_i and memorizes it in the finite control. (The computer has only finitely many different instructions.)

To each instruction corresponds its own **subroutine**. To simulate the instruction, the TM can use the same tick marking to find any required memory locations, and then execute the particular task. The task may be adding the content of a memory location to some register, for example.

Adding two numbers can be easily implemented (especially if we represent all number in the unary format so that number n is represented as the word a^n).

To write a word to the memory may require shifting all cells on the right hand side of the memory location, but we know how to do that.

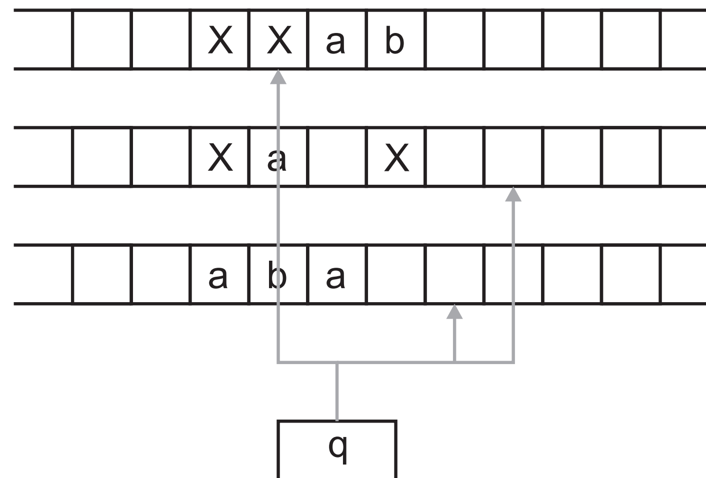
Modifications of Turing machines

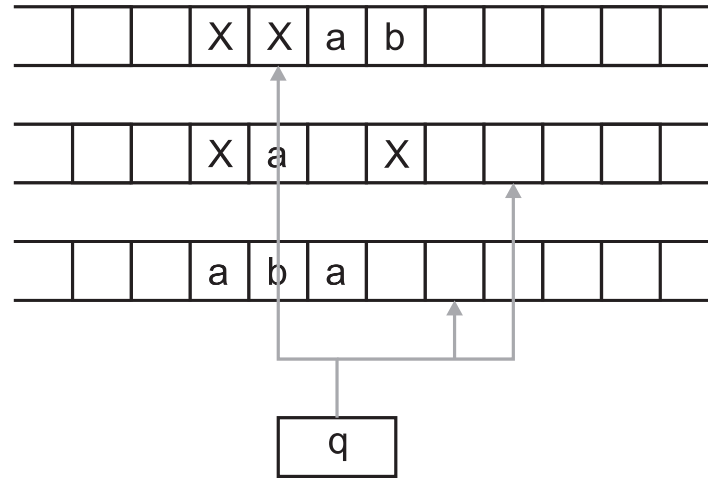
The following modifications do **not** make Turing machines more powerful. The modified TMs accept exactly the same family of r.e. languages as our basic model.

Modifications of Turing machines

The following modifications do **not** make Turing machines more powerful. The modified TMs accept exactly the same family of r.e. languages as our basic model.

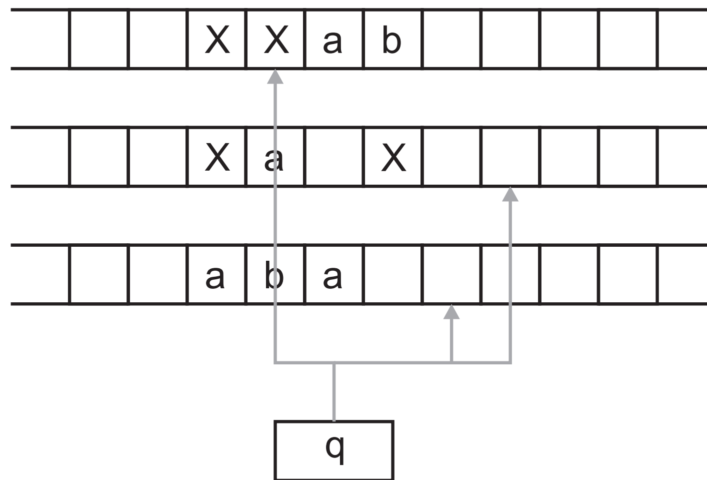
1. **Multiple tapes**. We can allow the TM to have more than one tape. Each tape has its own independent R/W head. This is different from one tape TM with multiple tracks, since the R/W heads of different tapes can now be at different positions.





Depending on the state of the finite control and the current tape symbols on all tapes the machine can

- change the state,
- overwrite the currently scanned symbols on all tapes, and
- move each R/W head to left or right independently of each other.



Formally, the transition function δ is now a (partial) function from

$$(Q \setminus \{f\}) \times \Gamma^n$$

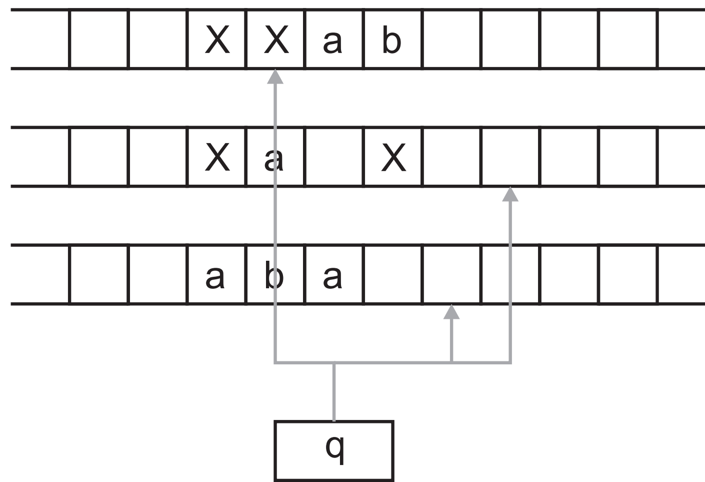
to

$$Q \times \Gamma^n \times \{L, R\}^n$$

where n is the number of tapes. A transition

$$\delta(q, X_1, X_2, \ldots, X_n) = (p, Y_1, Y_2, \ldots, Y_n, d_1, d_2, \ldots, d_n),$$

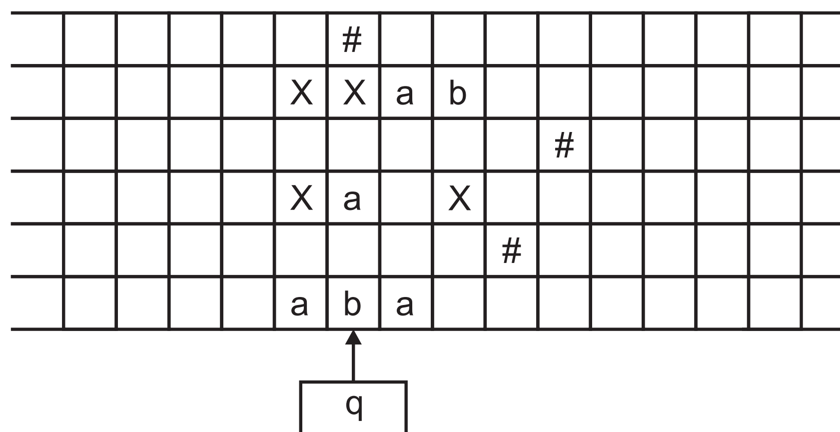
means that the machine, in state q, reading symbols X_1, $X_2, \ldots X_n$ from the n tapes, changes its state to p, writes symbols Y_1,$Y_2, \ldots Y_n$ on the tapes, and moves the first, second, third, etc. R/W head to directions indicated by d_1, $d_2, \ldots d_n$, respectively.



Initially, the input is written on tape number one, and all other tapes are blank. A word is accepted iff the machine enters a final state.

Simulating a multitape TM using a single tape.

The single tape will have two tracks for every tape of M: One track is identical to M's corresponding tape; The other one contains a single # indicating the position of the R/W head on that tape.



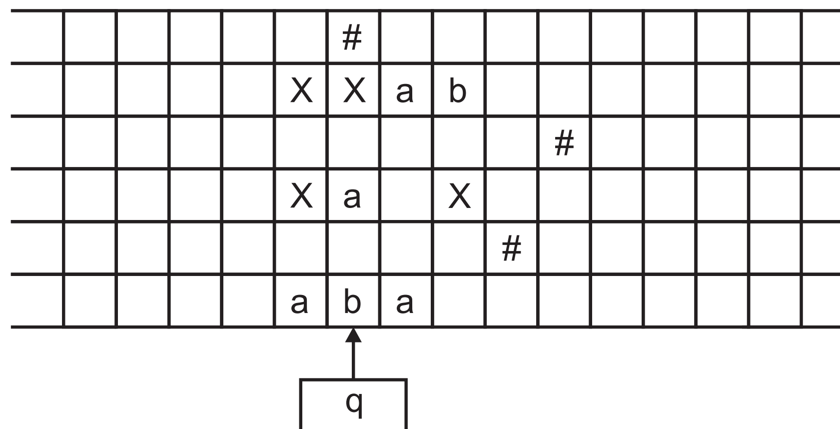
To simulate one move of the multitape machine M, we scan the tape from left to right, remembering in the finite control the tape symbols below #'s.

Once we have encountered all #'s, the machine can figure out the new state p and the action taken on each tape. During another sweep over the tape, the machine can execute the instruction by writing the required symbols and moving the #'s on the tape left or right.

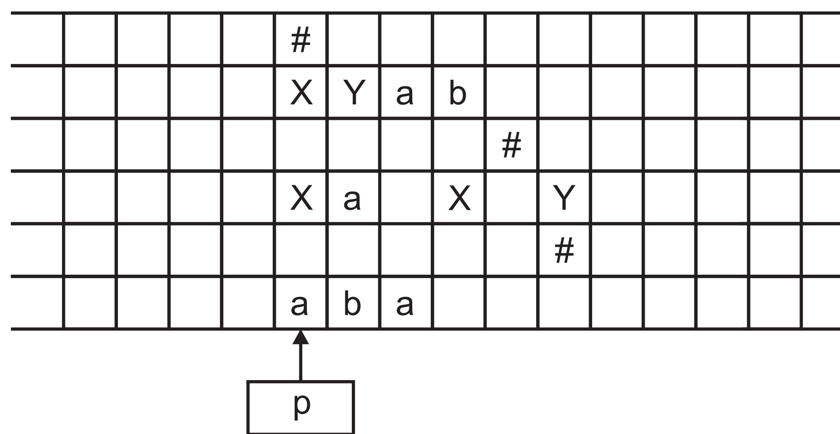
Example. If

$$\delta(q, X, B, B) = (p, Y, Y, B, L, L, R)$$

the simulating machine will turn



into



by a simulation loop that sweeps back-and-forth between the extremal #'s.

Note that simulating one step of the multitape machine requires scanning back-and-forth over the input, so the one-tape machine will be much slower. But all that matters is that the machines accept exactly same words.

3. **Nondeterministic Turing machines**. We can also introduce nondeterminism. Now δ is a function from

$$(Q \setminus \{f\}) \times \Gamma$$

to **subsets** of

$$Q \times \Gamma \times \{L, R\}.$$

When scanning tape symbol X in state q, the machine can execute any instruction from the set $\delta(q, X)$. If $\delta(q, X)$ is empty, the machine halts.

A word w is accepted if and only if there exists a computation that takes the initial ID

$$q_0 \; w$$

into a ID where the state is a final state.

(Note that there may be other computations that halt in a non-final state, or do not halt at all, but the word is accepted as long as there exists at least one accepting computation.)

Example. Recognizing the language

$$L = \{ww \mid w \in (a+b)^*\}$$

is made easier using non-determinism. We do not need to find the center point of the input word first: we may guess where the center point is. If in the end all letters have been erased we know that the guess was correct and the word is in the language.

Example. It is easy to construct a non-deterministic TM that recognizes the language

$$\{w\#u \mid w, u \in (a+b)^* \text{ and } w \text{ is a subword of } u \}.$$

The machine first makes a non-deterministic guess where w starts inside u. Then it verifies the correctness of the guess by erasing symbols in w and u, one by one. If the process ends successfully and the whole w gets erased, the word is in the language!

Simulating a non-deterministic TM M by a deterministic one.

Let r be the maximum size of the sets $\delta(q, X)$ for any q and X. In other words, there are always at most r possible choices in the nondeterministic machine.

The deterministic TM uses three tapes (which we know can be converted into a one-tape version.)

- Tape 1 contains the input, and that tape never changes.
- On tape 2 we generate words over the alphabet

  $$\{1, 2, \ldots, r\}$$

  in some predetermined order.

  (For example, start with the shortest words, and move up to longer and longer words. Words of equal length are generated in the lexicographical order. In other words, count integers 1,2,3,... in base r.)

  We use the word generated on the second tape to make a selection among non-deterministic choices for each move.

- For each word generated on tape 2 we simulate M on tape 3 using the content on tape 2 to select among possible non-deterministic choices.

More precisely: Let x be the word generated on tape 2. We copy the input from tape 1 to tape 3, and simulate the nondeterministic TM on tape 3.

At the i'th simulation step we use the i'th letter of x to choose which nondeterministic alternative to use.

In any of the following cases we give up, generate the next word on the second tape, and repeat the process:

- If the next r-ary digit on the second tape is greater than the number of choices in the current ID of the machine we are simulating, or
- If we consumed all letters from the second tape.

If on the other hand the machine halts in a final state, the simulating machine accepts the word.

The machine we constructed is deterministic. It is also equivalent to the original nondeterministic machine M:

$\Longrightarrow$ If the input w is accepted by M, there exists a sequence of choices that leads to an accepting calculation. Sooner or later that sequence of choices gets generated on tape 2. Once that happens, we get an accepting computation on tape 3, and the word w is accepted.

$\Longleftarrow$ If $w \notin L(M)$, there does not exist a sequence of choices that leads to a final state. Therefore, our deterministic machine never halts: it keeps generating longer and longer words on tape 2 never halting.

The deterministic TM in effect tries all possible non-deterministic choices one after the other until (if ever) it finds an accepting computation.

Note that the simulating deterministic machine M' is **much slower** than the original non-deterministic machine M. If M accepts word w in n moves, it may take over r^n moves by M' to accept the same word.

It is a celebrated open problem whether the deterministic machine needs to be so much slower.

Note that the simulating deterministic machine M' is **much slower** than the original non-deterministic machine M. If M accepts word w in n moves, it may take over r^n moves by M' to accept the same word.

It is a celebrated open problem whether the deterministic machine needs to be so much slower.

Denote by **P** the family of languages that are recognized by some deterministic TM in polynomial time. In other words, L is in P if there exists a Turing machine M and a polynomial p such that M recognizes L, and for every $w \in L$ the accepting computation of w by M uses at most $p(|w|)$ moves.

Analogously, let us denote by **NP** the family of languages that are recognized by some non-deterministic TM in polynomial time. (That means, for every $w \in L$ there exists an accepting computation for w that uses at most $p(|w|)$ moves.)

It is a famous open problem whether

$$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$$

i.e., whether there are languages that are recognized non-deterministically in polynomial time, but recognizing them deterministically requires super-polynomial time.

It is generally assumed that

$$P \neq NP$$

but no proof is known.

This problem has enormous practical importance because there are many important computational problems that are known to be in NP, but nobody has been able to find an efficient deterministic algorithm for them.

Closure properties

Theorem.

(1) The family of recursive languages is effectively closed under complementation. (If L is recursive, so is its complement $\Sigma^* \setminus L$.)

(2) The families of recursive and recursively enumerable languages are effectively closed under union and intersection. (If L_1 and L_2 are recursive or r.e., so are $L_1 \cup L_2$ and $L_1 \cap L_2$.)

(3) A language $L \subseteq \Sigma^*$ is recursive if and only if both L and its complement $\Sigma^* \setminus L$ are recursively enumerable.

We'll have negative results later (after we learn how to prove that a language is not recursive or r.e.) In particular: the family of r.e. language is **not** closed under complementation.

Proof.

(1) Recursive languages are closed under complementation.

Let L be a recursive language. This means it is recognized by some Turing machine

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$$

that halts on every input.

(1) Recursive languages are closed under complementation.

Let L be a recursive language. This means it is recognized by some Turing machine

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$$

that halts on every input.

Let us build a Turing machine

$$M' = (Q \cup \{f'\}, \Sigma, \Gamma, \delta', q_0, B, f')$$

that recognizes the complement of L:

M' is identical to M except that M' has a new final state f', and whenever $\delta(q, X)$ is not defined for $q \neq f$ we add the transition $\delta'(q, X) = (f', X, R)$.

- If $w \in L(M)$ then both M and M' halt in state f. That state is not the final state of M', so M' does not accept w, and $w \notin L(M')$.
- If $w \notin L(M)$ then M halts in a non-final state q. Therefore M' continues one more move and enters its final state f'. So $w \in L(M')$.

Clearly M' halts on every input.

(2) Recursive and r.e. families are closed under union and intersection.

Let L_1 and L_2 be recognized by TMs M_1 and M_2, respectively. In the recursive cases we also assume the corresponding TM halts on all inputs.

(2) Recursive and r.e. families are closed under union and intersection.

Let L_1 and L_2 be recognized by TMs M_1 and M_2, respectively. In the recursive cases we also assume the corresponding TM halts on all inputs.

Idea 1. We can construct a TM M that executes M_1 and M_1 **sequentially** on the same input w.
- **Intersection**: M accepts iff both M_1 and M_2 accept.
- **Union**: M accepts iff M_1 or M_2 accepts.

This works well for the intersections, and for the union of recursive languages.

However, this sequential approach does not work for the union of r.e. languages: It could be that M_1 does not halt with the input w while M_2 accepts w. Machine M should accept w, but the execution of M_1 never halts so M never gets into executing M_2.

(2) Recursive and r.e. families are closed under union and intersection.

Let L_1 and L_2 be recognized by TMs M_1 and M_2, respectively. In the recursive cases we also assume the corresponding TM halts on all inputs.

Idea 2. Execute M_1 and M_2 **in parallel** on the same input w. We can easily do this using two tapes: execute M_1 on tape 1 and M_2 on tape 2.

More precisely, M

- Copies the input w from tape 1 to tape 2,
- Using states in $Q_1 \times Q_2$ simulates simultaneously step-by-step TM M_1 on tape 1 and M_2 on tape 2.

More precisely, M

- Copies the input w from tape 1 to tape 2,
- Using states in $Q_1 \times Q_2$ simulates simultaneously step-by-step TM M_1 on tape 1 and M_2 on tape 2.

In the case of the union operation:

- As soon as either simulation enters its final state, M enters its final state and the input is accepted.
- If one of the simulations halts in a non-final state, that simulation enters and maintains a "wait" state while the other simulation continues.
- Once both simulations reach "wait" state then M halts in a non-final state.

Now it is clear that M accepts w iff M_1 or M_2 accepts w. Moreover, if M_1 and M_2 halt on all inputs then M has this same property.

We know how to convert the two-tape TM into a single tape TM. (And this conversion preserves the property of halting on all inputs.)

In the case of the intersection operation:

- If one of the simulations enter its final state than that simulation maintains that state while the other simulation continues.
- If one of the simulations halts in a non-final state, that simulation enters and maintains a "wait" state while the other simulation continues.
- Once both simulations halt: if both reached the final state then M enters its final state and the input is accepted. Otherwise, M halts in a non-final state.

Now it is clear that M accepts w iff M_1 and M_2 accept w. Moreover, if M_1 and M_2 halt on all inputs then M has this same property.

(3) A language $L \subseteq \Sigma^*$ is recursive if and only if both L and its complement $\Sigma^* \setminus L$ are recursively enumerable.

(3) A language $L \subseteq \Sigma^*$ is recursive if and only if both L and its complement $\Sigma^* \setminus L$ are recursively enumerable.

$\Longrightarrow$ If L is recursive then by (1) its complement $\Sigma^* \setminus L$ is also recursive. Every recursive language is also recursively enumerable.

(3) A language $L \subseteq \Sigma^*$ is recursive if and only if both L and its complement $\Sigma^* \setminus L$ are recursively enumerable.

$\Longleftarrow$ Assume that L is recognized by TM M_1, and that its complement $\Sigma^* \setminus L$ is recognized by TM M_2. We may assume that the machines do not halt unless they reach their final state. (Introduce a new infinite loop state for the cases when the machines halt in non-final states.)

As in (2) above, we construct a TM M that executes M_1 and M_2 in parallel on the same input w using two tapes. Note that precisely one of the two simulations will halt. If M_1 halts then M enters the final state, if M_2 halts then M halts in a non-final state.

Clearly M halts on every input, and it accepts w if and only if M_1 accepts w.

A little summary:

L is recursively enumerable (r.e.)

$\Longleftrightarrow$ $L = L(M)$ for a deterministic TM M

$\Longleftrightarrow$ $L = L(M)$ for a non-deterministic TM M

$\Longleftrightarrow$ $\exists$ a deterministic semi-algorithm for the decision problem
"Is a given word w in L ?"

$\Longleftrightarrow$ $\exists$ a non-deterministic semi-algorithm for the decision problem
"Is a given word w in L ?"

A non-deterministic semi-algorithm for a decision problem **guesses a certificate** x for the positivity of the given instance w, and verifies that x indeed proves that w is positive. (If w is positive then such a certificate x must exist; if w is negative then such a certificate x may not exist.)

L is recursive (Rec)

$\iff$ $L = L(M)$ for a deterministic TM M that halts with every input

$\iff$ $\exists$ a deterministic algorithm for the decision problem
"Is a given word w in L ?"

$\iff$ $L = L(M_1)$ and $\Sigma^* \setminus L = L(M_2)$ for non-deterministic TMs M_1 and M_2

$\iff$ $\exists$ non-deterministic semi-algorithms for the decision problems
"Is a given word w in L ?" and "Is a given word w in $\Sigma^* \setminus L$?"

When a decision problem P is semi-decidable we say that the **positive instances** of P are semi-decidable. When the complement of a decision problem P is semi-decidable we say that the **negative instances** of P are semi-decidable.

So a decision problem is decidable iff its positive instances are semi-decidable and its negative instances are semi-decidable.

L is recursive (Rec)

$\iff$ $L = L(M)$ for a deterministic TM M that halts with every input

$\iff$ $\exists$ a deterministic algorithm for the decision problem
"Is a given word w in L ?"

$\iff$ $L = L(M_1)$ and $\Sigma^* \setminus L = L(M_2)$ for non-deterministic TMs M_1 and M_2

$\iff$ $\exists$ non-deterministic semi-algorithms for the decision problems
"Is a given word w in L ?" and "Is a given word w in $\Sigma^* \setminus L$?"

Thus we sometimes prove the decidability of P by proving that

- there exist certificates for the positivity of positive instances, and
- there exist certificates for the negativity of negative instances.

The validity of certificates must be effectively checkable.

Example. The decision problem: ”Does a given directed graph contain a cycle?” is decidable.

Positive instances: A certificate is a sequence $v_1, v_2, \ldots, v_k$ of vertices that forms a cycle. (To verify the certificate check that for every i there is an edge $v_i \longrightarrow v_{i+1}$, and that there also is the edge $v_k \longrightarrow v_1$.)

Negative instances: A certificate is a permutation $v_1, v_2, \ldots, v_n$ of all vertices that is a topological sort. (To verify the certificate check that it is a permutation and that for all $i < j$ there is no edge $v_j \longrightarrow v_i$.)

Example. The decision problem: "Does a given directed graph contain a cycle?" is decidable.

Positive instances: A certificate is a sequence $v_1, v_2, \ldots, v_k$ of vertices that forms a cycle. (To verify the certificate check that for every i there is an edge $v_i \longrightarrow v_{i+1}$, and that there also is the edge $v_k \longrightarrow v_1$.)

Negative instances: A certificate is a permutation $v_1, v_2, \ldots, v_n$ of all vertices that is a topological sort. (To verify the certificate check that it is a permutation and that for all $i < j$ there is no edge $v_j \longrightarrow v_i$.)

(In practice one would rather use **depth-first-search** to check for the presence of cycles in a given directed graph. This algorithm is very fast: linear time in the size of the graph.)

Decision problems concerning Turing machines

In the following we consider decision problems whose instances involve Turing machines. We prove that many such problems are undecidable (even non-semi-decidable).

So there does not exist a Turing machine that recognizes the (encodings of) positive instances of these problems.

We start by fixing our encoding scheme, *i.e.*, the way we represent Turing machines as words.

Let us consider now only Turing machines with the input alphabet $\{a, b\}$.

By renaming the states and tape letters we may assume that the state set is

$$Q = \{q_1, q_2, \ldots, q_n\}$$

and that the tape alphabet is

$$\Gamma = \{X_1, X_2, \ldots, X_m\}$$

The input alphabet $\{a, b\}$ and the blank symbol B must be part of the tape alphabet, so we assume that $m \geq 3$ and

$$X_1 = a, X_2 = b, X_3 = B$$

We may also assume that q_1 is the initial state and q_n is the final state of the machine, and that $n \geq 2$.

So we restrict our instances to Turing machines $M = (Q, \Sigma, \Gamma, \delta, q_1, B, q_n)$ where

$$\begin{aligned} Q &= \{q_1, q_2, \dots, q_n\}, \\ \Sigma &= \{a, b\}, \\ \Gamma &= \{X_1, X_2, \dots, X_m\} \text{ with} \\ &\quad X_1 = a, \\ &\quad X_2 = b, \\ &\quad X_3 = B, \end{aligned}$$

and $n \geq 2$ and $m \geq 3$.

It is clear that such Turing machines recognize all recursively enumerable languages over the alphabet $\{a, b\}$, and that halting machines of this kind recognize all recursive languages over $\{a, b\}$.

Encoding of M as a word $\langle M \rangle \in \{a, b\}^*$

Let us number L and R as directions 1 and 2:

$$\begin{aligned} D_1 &= L, \\ D_2 &= R. \end{aligned}$$

An arbitrary transition

$$\delta(q_i, X_j) = (q_k, X_l, D_s)$$

by the machine M is encoded as the word

$$a^i \; b \; a^j \; b \; a^k \; b \; a^l \; b \; a^s$$

Then we encode the machine M as the word

$$\langle M \rangle = bbb \; a^n \; bb \; a^m \; bb \; \text{code}_1 \; bb \; \text{code}_2 \; bb \ldots bb \; \text{code}_r \; bbb$$

where code_1, code_2, . . . ,code_r are encodings of all the defined transitions $\delta(q_i, X_j)$, listed in the **lexicographic order** of i, j.

Example. Let

$$M = (\{q_1, q_2, q_3\}, \{a, b\}, \{a, b, B, X_4\}, \delta, q_1, B, q_3)$$

with the transitions

$$\begin{aligned}
\delta(q_1, a) &= (q_1, b, R), \\
\delta(q_1, B) &= (q_2, B, L), \\
\delta(q_2, b) &= (q_1, X_4, R), \\
\delta(q_2, B) &= (q_3, a, R), \\
\delta(q_2, X_4) &= (q_1, B, L).
\end{aligned}$$

Then

$\langle M \rangle =$

Example. Let

$$M = (\{q_1, q_2, q_3\}, \{a, b\}, \{a, b, B, X_4\}, \delta, q_1, B, q_3)$$

with the transitions

$$\begin{array}{lcl} \delta(q_1, a) & = & (q_1, b, R), \\ \delta(q_1, B) & = & (q_2, B, L), \\ \delta(q_2, b) & = & (q_1, X_4, R), \\ \delta(q_2, B) & = & (q_3, a, R), \\ \delta(q_2, X_4) & = & (q_1, B, L). \end{array}$$

Then

$$\begin{array}{lcl} \langle M \rangle & = & bbbaaabbaaaabb\ abababaabaa\ bb\ abaaabaabaaaba\ bb\ aabaababaaaabaa \\ & & bb\ aabaaabaaababaa\ bb\ aabaaaababaaaba\ bbb \end{array}$$

The set of valid encodings of Turing machines is recursive: there exists an algorithm that checks whether a given word w is an encoding of some Turing machine.

Lemma. The language

$$L_{enc} = \{\ w \in \{a,b\}^* \mid\ w = \langle M \rangle \text{ for some Turing machine } M\ \ \}$$

is recursive.

Proof.

Proof that L_{enc} is recursive:

There is a TM that verifies the following facts about the given input word w:

- Word w has the correct form

$$bbb\ a^n\ bb\ a^m\ bb\ \text{code}_1\ bb\ \text{code}_2\ bb \ldots bb\ \text{code}_r\ bbb$$

  for some $n \geq 2$ and $m \geq 3$, where each code_t is a word from $a^+ba^+ba^+ba^+ba^+$. (Such words form even a regular language.)

Proof that L_{enc} is recursive:

There is a TM that verifies the following facts about the given input word w:

- Word w has the correct form

$$bbb\ a^n\ bb\ a^m\ bb\ \text{code}_1\ bb\ \text{code}_2\ bb \ldots bb\ \text{code}_r\ bbb$$

  for some $n \geq 2$ and $m \geq 3$, where each code_t is a word from $a^+ba^+ba^+ba^+ba^+$. (Such words form even a regular language.)

- Each $\textbf{code}_t = a^i\ b\ a^j\ b\ a^k\ b\ a^l\ b\ a^s$ satisfies the conditions

$$\begin{array}{lcccl} 1 & \leq & i & \leq & n-1, \\ 1 & \leq & k & \leq & n, \\ 1 & \leq & j,l & \leq & m, \\ 1 & \leq & s & \leq & 2. \end{array}$$

  Then each code_t is a valid encoding of some transition $\delta(q_i, X_j) = (q_k, X_l, D_s)$.

Proof that L_{enc} is recursive:

There is a TM that verifies the following facts about the given input word w:

- Word w has the correct form

$$bbb\ a^n\ bb\ a^m\ bb\ \text{code}_1\ bb\ \text{code}_2\ bb \ldots bb\ \text{code}_r\ bbb$$

  for some $n \geq 2$ and $m \geq 3$, where each code_t is a word from $a^+ba^+ba^+ba^+ba^+$. (Such words form even a regular language.)

- Each $\textbf{code}_t = a^i\ b\ a^j\ b\ a^k\ b\ a^l\ b\ a^s$ satisfies the conditions

$$\begin{array}{rcl} 1 \leq & i & \leq n-1, \\ 1 \leq & k & \leq n, \\ 1 \leq & j, l & \leq m, \\ 1 \leq & s & \leq 2. \end{array}$$

  Then each code_t is a valid encoding of some transition $\delta(q_i, X_j) = (q_k, X_l, D_s)$.

- For all consecutive

$$\text{code}_t = a^i\ b\ a^j\ b\ a^k\ b\ a^l\ b\ a^s \text{ and } \text{code}_{t+1} = a^{i'}\ b\ a^{j'}\ b\ a^{k'}\ b\ a^{l'}\ b\ a^{s'}$$

  we have either $i < i'$, or both $i = i'$ and $j < j'$. This guarantees the proper lexicographic ordering of the transitions. This also guarantees that the machine is deterministic.

Proof that L_{enc} is recursive:

There is a TM that verifies the following facts about the given input word w:

- Word w has the correct form

$$bbb\ a^n\ bb\ a^m\ bb\ \text{code}_1\ bb\ \text{code}_2\ bb \ldots bb\ \text{code}_r\ bbb$$

  for some $n \geq 2$ and $m \geq 3$, where each code_t is a word from $a^+ba^+ba^+ba^+ba^+$. (Such words form even a regular language.)

- Each $\mathbf{code}_t = a^i\ b\ a^j\ b\ a^k\ b\ a^l\ b\ a^s$ satisfies the conditions

$$\begin{array}{lllll} 1 & \leq & i & \leq & n-1, \\ 1 & \leq & k & \leq & n, \\ 1 & \leq & j,l & \leq & m, \\ 1 & \leq & s & \leq & 2. \end{array}$$

  Then each code_t is a valid encoding of some transition $\delta(q_i, X_j) = (q_k, X_l, D_s)$.

- For all consecutive

$$\text{code}_t = a^i\ b\ a^j\ b\ a^k\ b\ a^l\ b\ a^s \text{ and } \text{code}_{t+1} = a^{i'}\ b\ a^{j'}\ b\ a^{k'}\ b\ a^{l'}\ b\ a^{s'}$$

  we have either $i < i'$, or both $i = i'$ and $j < j'$. This guarantees the proper lexicographic ordering of the transitions. This also guarantees that the machine is deterministic.

Input word w represents a Turing machine iff all three conditions are satisfied.

We define next the **complemented diagonal language**

$$L_d = \{\langle M \rangle \mid \ M \text{ does not accept the word } \langle M \rangle \ \}.$$

(Encodings of those Turing machines that do not accept their own encoding.)

The language L_d is a well defined language over the alphabet $\{a, b\}$. It is our first example of a language that is not recursively enumerable.

Theorem (Alan Turing 1936). The complemented diagonal language

$$L_d = \{\langle M \rangle \mid \ M \text{ does not accept the word } \langle M \rangle \ \}$$

is not recursively enumerable.

Proof.

Theorem (Alan Turing 1936). The complemented diagonal language

$$L_d = \{\langle M \rangle \mid \ M \text{ does not accept the word } \langle M \rangle \ \}$$

is not recursively enumerable.

Proof. An indirect proof: Suppose that, contrary to the claim, L_d is recognized by some Turing machine M_d.

Does M_d accept the input $\langle M_d \rangle$?

Theorem (Alan Turing 1936). The complemented diagonal language

$$L_d = \{\langle M \rangle \mid \; M \text{ does not accept the word } \langle M \rangle \;\}$$

is not recursively enumerable.

Proof. An indirect proof: Suppose that, contrary to the claim, L_d is recognized by some Turing machine M_d.

Does M_d accept the input $\langle M_d \rangle$?

- If M_d does not accept $\langle M_d \rangle$ then, by the definition of the language L_d, the word $\langle M_d \rangle$ is in L_d. But M_d recognizes L_d, so M_d accepts $\langle M_d \rangle$, a contradiction.

Theorem (Alan Turing 1936). The complemented diagonal language

$$L_d = \{\langle M \rangle \mid \ M \text{ does not accept the word } \langle M \rangle \ \}$$

is not recursively enumerable.

Proof. An indirect proof: Suppose that, contrary to the claim, L_d is recognized by some Turing machine M_d.

Does M_d accept the input $\langle M_d \rangle$?

- If M_d does not accept $\langle M_d \rangle$ then, by the definition of the language L_d, the word $\langle M_d \rangle$ is in L_d. But M_d recognizes L_d, so M_d accepts $\langle M_d \rangle$, a contradiction.
- If M_d accepts $\langle M_d \rangle$ then, by the definition of the language L_d, word $\langle M_d \rangle$ is not in L_d. But M_d recognizes L_d, so M_d does not accept $\langle M_d \rangle$, a contradiction.

Theorem (Alan Turing 1936). The complemented diagonal language

$$L_d = \{\langle M\rangle \mid \; M \text{ does not accept the word } \langle M\rangle \;\}$$

is not recursively enumerable.

Proof. An indirect proof: Suppose that, contrary to the claim, L_d is recognized by some Turing machine M_d.

Does M_d accept the input $\langle M_d\rangle$?

- If M_d does not accept $\langle M_d\rangle$ then, by the definition of the language L_d, the word $\langle M_d\rangle$ is in L_d. But M_d recognizes L_d, so M_d accepts $\langle M_d\rangle$, a contradiction.
- If M_d accepts $\langle M_d\rangle$ then, by the definition of the language L_d, word $\langle M_d\rangle$ is not in L_d. But M_d recognizes L_d, so M_d does not accept $\langle M_d\rangle$, a contradiction.

In both cases we reach a contradiction. Therefore M_d cannot exist.

The proof is, in fact, the powerful **diagonal** (or self-reference) argument used in many contexts in mathematics:

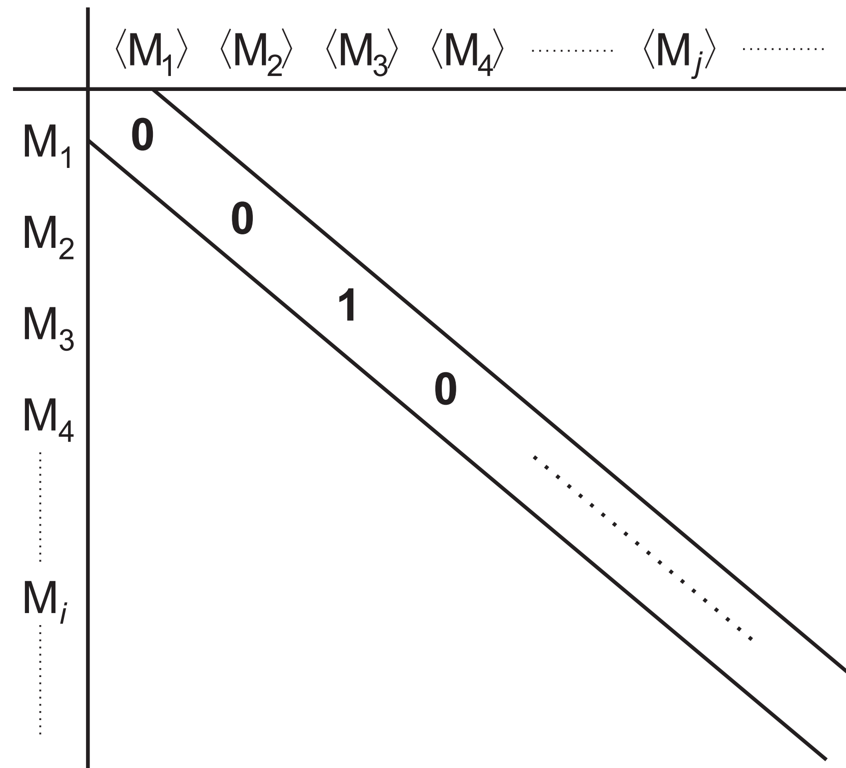
- G. Cantor (1891): the set of reals is not countable,
- K. Gödel (1931): first incompleteness theorem,
- Epimenides the Cretan: "*all Cretans are liars*"
- ...

The diagonal aspect of Turing's proof can be visualized as follows: Consider an arbitrary enumeration $M_1, M_2, \ldots$ of Turing machines with input alphabet $\{a, b\}$, and an an infinite 0/1-matrix whose rows are indexed by Turing machines M_i and whose columns are indexed by their encodings $\langle M_i \rangle$.

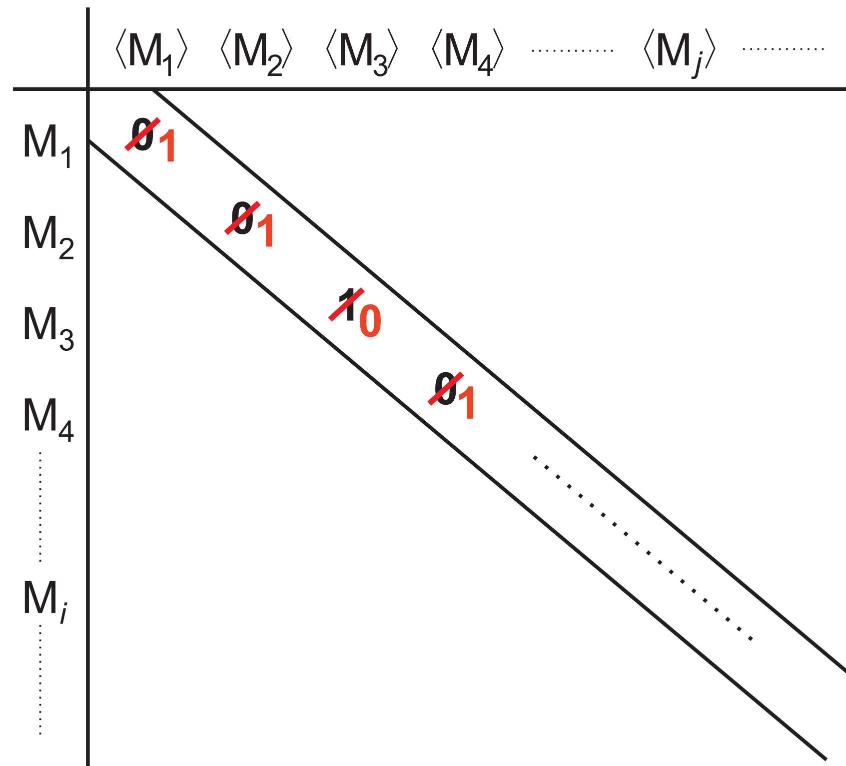
The entry $(M_i, \langle M_j \rangle)$ of the table is 1 iff M_i accepts word $\langle M_j \rangle$, so that the row indexed by M_i is the **characteristic sequence** identifying which words $\langle M_j \rangle$ are accepted by M_i:

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	...	$\langle M_j \rangle$	...
M_1							
M_2							
M_3							
M_4							
...							
M_i						1, if M_i accepts $\langle M_j \rangle$ 0, if M_i does not accept $\langle M_j \rangle$	
...							

Consider the sequence of bits along the diagonal $(M_i, \langle M_i \rangle)$, and complement this sequence by swapping each bit.



Consider the sequence of bits along the diagonal $(M_i, \langle M_i \rangle)$, and complement this sequence by swapping each bit.



The complemented diagonal cannot be identical to any row of the table. So a language whose characteristic sequence is the complemented diagonal is not recognized by any Turing machine.

Our L_d is this complemented diagonal language.

So the following decision problem is not semi-decidable (and hence undecidable):

"Given a TM M, is it true that $\langle M \rangle \notin L(M)$?"

This decision problem is rather artificial, but it serves as a **seed** from which we can conclude many other – more natural – problems to be undecidable. The method we use to obtain undecidability results is **reduction**.

As a first example of reductions, let us show that there is no algorithm to determine if a given Turing machine accepts a given input word:

Corollary. The language

$$L_u = \{ \ \langle M \rangle \# w \mid \ M \text{ accepts the input word } w \ \}$$

is not recursive. More specifically, its complement is not recursively enumerable.

Proof.

As a first example of reductions, let us show that there is no algorithm to determine if a given Turing machine accepts a given input word:

Corollary. The language

$$L_u = \{ \langle M \rangle \# w \mid \ M \text{ accepts the input word } w \}$$

is not recursive. More specifically, its complement is not recursively enumerable.

Proof. Suppose the contrary: There is a Turing machine $M_{\bar{u}}$ that recognizes the complement of L_u.

Then the following TM M_d recognizes L_d, contradicting Turing's theorem.

Machine M_d works as follows: On input w it

1. Checks whether w is a valid encoding of some Turing machine. This can be done effectively, as the language L_{enc} of valid encodings is recursive.

   If w is not a valid encoding, machine M_d halts in a non-final state.

2. If $w = \langle M \rangle$ is a valid encoding then M_d writes $w\#w = \langle M \rangle \# \langle M \rangle$ on the tape and starts $M_{\bar{u}}$ on this input. The final state is entered if and only if $M_{\bar{u}}$ accepts this word.

Such a Turing machine M_d exists if $M_{\bar{u}}$ exists. (It uses $M_{\bar{u}}$ as a subroutine.) Clearly M_d recognizes L_d. This is not possible so $M_{\bar{u}}$ cannot exist.

The reduction described a TM M_d that recognizes a known non-r.e. language L_d, using as a subroutine a hypothetical TM $M_{\bar{u}}$ that recognizes the complement of L_u.

(I did not provide a detailed transition function of M_d, but a precise enough construction of it so that it is clear that it exists if $M_{\bar{u}}$ exists.)

In the reduction I described a TM M_d that recognizes a known non-r.e. language L_d, using as a subroutine a hypothetical TM $M_{\bar{u}}$ that recognizes the complement of L_u.

(I did not provide a detailed transition function of M_d, but a precise enough construction of it so that it is clear that it exists if $M_{\bar{u}}$ exists.)

Here's a "high level" explanation of the reduction:

Suppose there is a semi-algorithm to test whether a given Turing machine does not accept a given word. This semi-algorithm then recognizes 0's in the infinite table discussed above. But it then also identifies entries 1 on the inverted diagonal, contradicting Turing's theorem.

Another (more complicated) example of a reduction.

Corollary. The language

$$L_{halt} = \{ \ \langle M \rangle \mid \ M \text{ halts when started on the blank tape } \}$$

is not recursive.

Proof.

Another (more complicated) example of a reduction.

Corollary. The language

$$L_{halt} = \{ \ \langle M \rangle \mid \ M \text{ halts when started on the blank tape } \}$$

is not recursive.

Proof. Let us do the reduction at the level of an informal algorithm, rather than explicitly constructing a Turing machine.

Suppose, on the contrary to the claim, that there exists an algorithm A_{halt} to determine if a given Turing machine halts when started on the blank tape.

Then the following algorithm $A_{\bar{u}}$ determines whether a given Turing machine M accepts a given word w, contradicting the previous corollary that states that such an algorithm does not exist. The algorithm $A_{\bar{u}}$ uses the hypothetical algorithm A_{halt} as a subroutine.

On input M and w the algorithm $A_{\bar{u}}$ **does the following**.

1. It **builds a new Turing machine** M' that works as follows:
   - First M' writes w on its tape. (This may be done using $|w|$ states.)
   - Then M' moves back to the first letter of w and enters the initial state of M.
   - From there on, transitions of machine M are used.
   - For each halting but non-final state of M the machine M' goes into a new looping state that makes the machine move to the right indefinitely without ever halting.

   Clearly, this M' **halts on the blank initial tape if and only if** M **accepts the input** w.

On input M and w the algorithm $A_{\bar{u}}$ **does the following**.

1. It **builds a new Turing machine** M' that works as follows:

   - First M' writes w on its tape. (This may be done using $|w|$ states.)
   - Then M' moves back to the first letter of w and enters the initial state of M.
   - From there on, transitions of machine M are used.
   - For each halting but non-final state of M the machine M' goes into a new looping state that makes the machine move to the right indefinitely without ever halting.

   Clearly, this M' **halts on the blank initial tape if and only if** M **accepts the input** w.

2. Algorithm $A_{\bar{u}}$ does not execute M'. Instead, it just **gives** M' **as an input to the hypothetical algorithm** A_{halt}, and returns "yes" if A_{halt} returns "yes", and returns "no" if A_{halt} returns "no".

On input M and w the algorithm $A_{\bar{u}}$ **does the following**.

1. It **builds a new Turing machine M'** that works as follows:
   - First M' writes w on its tape. (This may be done using $|w|$ states.)
   - Then M' moves back to the first letter of w and enters the initial state of M.
   - From there on, transitions of machine M are used.
   - For each halting but non-final state of M the machine M' goes into a new looping state that makes the machine move to the right indefinitely without ever halting.

   Clearly, this M' **halts on the blank initial tape if and only if M accepts the input w**.

2. Algorithm $A_{\bar{u}}$ does not execute M'. Instead, it just **gives M' as an input to the hypothetical algorithm A_{halt}**, and returns "yes" if A_{halt} returns "yes", and returns "no" if A_{halt} returns "no".

Algorithm $A_{\bar{u}}$ works correctly: it returns "yes" on input M and w if and only if M accepts w. But such algorithm cannot exist by the previous corollary. Hence, the algorithm A_{halt} cannot exist either.

A more careful analysis of the reduction above one concludes that the complement of L_{halt} is not recursively enumerable, *i.e.*, non-halting is not semi-decidable.

A more careful analysis of the reduction above one concludes that the complement of L_{halt} is not recursively enumerable, *i.e.*, non-halting is not semi-decidable.

We can also see this as follows. The language L_{halt} is recursively enumerable. (One can effectively simulate the given Turing machine on a blank tape until – if ever – it halts. The semi-algorithm answers "yes" if the simulation halts.)

Now, because L_{halt} is recursively enumerable but not recursive, its complement cannot be recursively enumerable.

Turing reductions

The reduction method is used as follows: To prove that a decision problem P is undecidable we assume that there would exist an algorithm A that solves P. Then we describe an algorithm A' that solves some known undecidable problem P', using A as a subroutine. Since such algorithm A' cannot exist we conclude that algorithm A cannot exist either.

(Notice that such undecidability proof involves designing an algorithm! – but the algorithm is for a known undecidable problem P' and it uses a hypothetical subroutine A that solves P.)

Turing reductions

The reduction method is used as follows: To prove that a decision problem P is undecidable we assume that there would exist an algorithm A that solves P. Then we describe an algorithm A' that solves some known undecidable problem P', using A as a subroutine. Since such algorithm A' cannot exist we conclude that algorithm A cannot exist either.

(Notice that such undecidability proof involves designing an algorithm! – but the algorithm is for a known undecidable problem P' and it uses a hypothetical subroutine A that solves P.)

A reduction to show that problem P is not semi-decidable works analogously: Assume that a semi-algorithm for P exists. Build a semi-algorithm A' (using A as a subroutine) for a problem P' that is known to be non-semi-decidable. As such A' cannot exist, semi-algorithm A cannot exist either.

Turing reductions

The reduction method is used as follows: To prove that a decision problem P is undecidable we assume that there would exist an algorithm A that solves P. Then we describe an algorithm A' that solves some known undecidable problem P', using A as a subroutine. Since such algorithm A' cannot exist we conclude that algorithm A cannot exist either.

(Notice that such undecidability proof involves designing an algorithm! – but the algorithm is for a known undecidable problem P' and it uses a hypothetical subroutine A that solves P.)

A reduction to show that problem P is not semi-decidable works analogously: Assume that a semi-algorithm for P exists. Build a semi-algorithm A' (using A as a subroutine) for a problem P' that is known to be non-semi-decidable. As such A' cannot exist, semi-algorithm A cannot exist either.

Such reductions are called **Turing reductions**. (Later we'll see some other, weaker types of reductions.)

Example. Yet another example of a reduction: Let us show that emptyness of a given r.e. language is undecidable. (=there is no algorithm to determine if a given TM accepts any words.)

Example. Yet another example of a reduction: Let us show that emptyness of a given r.e. language is undecidable. (=there is no algorithm to determine if a given TM accepts any words.)

Suppose the contrary: algorithm A determines if a given Turing machine accepts some word. Then we have the following algorithm A' to determine if a given Turing machine M accepts a given word w:

1. First A' builds a new Turing machine M' that erases its input, writes w on the tape, moves to the first letter of w and enters the initial state of M. From there on, the transitions of M are used. Such machine M' can be effectively constructed for any given M and w.

2. Next A' gives the machine M' it constructed as an input to the hypothetical algorithm A, and returns the answer that A provides.

Example. Yet another example of a reduction: Let us show that emptyness of a given r.e. language is undecidable. (=there is no algorithm to determine if a given TM accepts any words.)

Suppose the contrary: algorithm A determines if a given Turing machine accepts some word. Then we have the following algorithm A' to determine if a given Turing machine M accepts a given word w:

1. First A' builds a new Turing machine M' that erases its input, writes w on the tape, moves to the first letter of w and enters the initial state of M. From there on, the transitions of M are used. Such machine M' can be effectively constructed for any given M and w.

2. Next A' gives the machine M' it constructed as an input to the hypothetical algorithm A, and returns the answer that A provides.

Note that

- **if M accepts w then M' accepts all words**,
- **if M does not accept w then M' does not accept any word**.

So A returns "yes" on input M' if and only if M accepts w.

Example. Yet another example of a reduction: Let us show that emptyness of a given r.e. language is undecidable. (=there is no algorithm to determine if a given TM accepts any words.)

Suppose the contrary: algorithm A determines if a given Turing machine accepts some word. Then we have the following algorithm A' to determine if a given Turing machine M accepts a given word w:

1. First A' builds a new Turing machine M' that erases its input, writes w on the tape, moves to the first letter of w and enters the initial state of M. From there on, the transitions of M are used. Such machine M' can be effectively constructed for any given M and w.
2. Next A' gives the machine M' it constructed as an input to the hypothetical algorithm A, and returns the answer that A provides.

Note that

- **if M accepts w then M' accepts all words**,
- **if M does not accept w then M' does not accept any word**.

So A returns "yes" on input M' if and only if M accepts w.

Algorithm A' described above cannot exist (solves a known undecidable problem) so the hypothetical algorithm A does not exist.

A universal Turing machine

Recall

$$L_u = \{ \langle M \rangle \# w \mid \; M \text{ accepts the input word } w \}$$

We saw that L_u is not recursive.

However, L_u is recursively enumerable. (Informally: the semi-algorithm first checks that the given input is of the correct form $\langle M \rangle \# w$. It then simulates the Turing machine M on the input w until – if ever – M halts. If M halts in the final state then semi-algorithm answers "yes".)

A universal Turing machine

Recall

$$L_u = \{ \langle M \rangle \# w \mid M \text{ accepts the input word } w \}$$

We saw that L_u is not recursive.

However, L_u is recursively enumerable. (Informally: the semi-algorithm first checks that the given input is of the correct form $\langle M \rangle \# w$. It then simulates the Turing machine M on the input w until – if ever – M halts. If M halts in the final state then semi-algorithm answers "yes".)

Recursively enumerable languages are recognized by Turing machines, so there exists a Turing machine M_u that recognizes L_u. Such a machine M_u is called a **universal Turing machine**. It can simulate any given Turing machine M on any given input w, when given the description (=encoding $\langle M \rangle$) of the machine to be simulated.

So M_u is a **programmable computer**: rather than building a new TM for each new language, one can use the same TM M_u and only change the "program" $\langle M \rangle$ that describes which Turing machine should be simulated.

We can also make the following observations:

Corollary. There are recursively enumerable languages that are not recursive.

Proof. For example L_u is such a language, as is also L_{halt} and the complement of L_d.

We can also make the following observations:

Corollary. There are recursively enumerable languages that are not recursive.

Proof. For example L_u is such a language, as is also L_{halt} and the complement of L_d.

Corollary. The family of recursively enumerable languages is not closed under complementation.

Proof. Any language L that is recursively enumerable but not not recursive confirms this.

(If L and the complement of L would be recursively enumerable then L would be recursive.)

A summary of the characters of the play so far:

- $\langle M \rangle$ denotes the encoding (as a word over the alphabet $\{a, b\}$) of a Turing machine M having the input alphabet $\{a, b\}$.

- $L_{enc} = \{\langle M \rangle \mid M \text{ is a Turing machine }\}$ is the language containing all valid encodings of such Turing machines. It is recursive.

- $L_d = \{\langle M \rangle \mid \langle M \rangle \notin L(M)\}$ is the complemented diagonal language that contains encodings $\langle M \rangle$ of Turing machines M that do not accept their own encodings $\langle M \rangle$. It is not recursively enumerable.

- $L_{halt} = \{\langle M \rangle \mid \text{ Turing machine } M \text{ halts on the blank initial tape }\}$. It is r.e. but not recursive.

- $L_u = \{\langle M \rangle \# w \mid w \in L(M)\}$ is the language containing encodings of Turing machines and inputs that they accept. It is also r.e. but not recursive.

- M_u is a universal Turing machine that recognizes the language L_u, *i.e.*, $L(M_u) = L_u$.

Rice's theorem

We have seen many undecidable questions that concern Turing machines. Some questions are clearly decidable (e.g., "Does a given Turing machine have 5 states ?").

Rice's theorem states that any non-trivial question that only concerns the language that a TM recognizes, rather than the machine itself, is undecidable.

Rice's theorem

We have seen many undecidable questions that concern Turing machines. Some questions are clearly decidable (e.g., "Does a given Turing machine have 5 states ?").

Rice's theorem states that any non-trivial question that only concerns the language that a TM recognizes, rather than the machine itself, is undecidable.

More precisely: Let $\mathcal{P}$ be any family of languages. We call $\mathcal{P}$ a **non-trivial property** of Turing machines if there exist Turing machines M_1 and M_2 such that $L(M_1) \in \mathcal{P}$ and $L(M_2) \notin \mathcal{P}$.

Theorem. Let $\mathcal{P}$ be a non-trivial property of Turing machines. There is no algorithm to determine if a given Turing machine M has $L(M) \in \mathcal{P}$.

Proof. On the blackboard.

Example. Rice's theorem shows, for example, that the following questions are undecidable:

- "Does a given TM accept all input words ?"
- "Is $L(M)$ regular for a given TM M ?"
- "Does a given Turing machine accept all palindromes ?"
- $\ldots$

Example. Rice's theorem shows, for example, that the following questions are undecidable:

- "Does a given TM accept all input words ?"
- "Is $L(M)$ regular for a given TM M ?"
- "Does a given Turing machine accept all palindromes ?"
- $\ldots$

Remark. A careful analysis of the proof of Rice's theorem shows that, for a non-trivial property $\mathcal{P}$ such that $\emptyset \in \mathcal{P}$, it is non-semi-decidable for a given Turing machine M whether $L(M) \in \mathcal{P}$.

Other undecidable problems: rewrite systems

A **semi-Thue system** $T = (\Sigma, R)$ consists of

- an alphabet Σ, and
- a finite set R of **rewrite rules**

$$u \longrightarrow v$$

  where $u, v \in \Sigma^*$.

Other undecidable problems: rewrite systems

A **semi-Thue system** $T = (\Sigma, R)$ consists of

- an alphabet Σ, and
- a finite set R of **rewrite rules**

  $$u \longrightarrow v$$

  where $u, v \in \Sigma^*$.

A rewrite rule $u \longrightarrow v$ allows **derivation steps**

$$x\ u\ y \Rightarrow x\ v\ y$$

for all $x, y \in \Sigma^*$.

In other words, a derivation step consists of replacing subword u by v.

Notations $x \Rightarrow^* y$ and $x \Rightarrow^+ y$ and $x \Rightarrow^i y$ have the usual interpretation.

Example. Let $T = (\{a, b\}, R)$ where R contains three rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

Then $ababa \Rightarrow^* bab$ because

$$\underline{aba}ba \Rightarrow ba\underline{bb}a \Rightarrow b\underline{aba} \Rightarrow \underline{bb}ab \Rightarrow bab$$

(Underlining indicates the subword being rewritten.)

The **word problem** of semi-Thue systems asks for a given semi-Thue system $T = (\Delta, R)$ and given words $x, y \in \Sigma^*$ whether $x \Rightarrow^* y$ in T.

The **word problem** of semi-Thue systems asks for a given semi-Thue system $T = (\Delta, R)$ and given words $x, y \in \Sigma^*$ whether $x \Rightarrow^* y$ in T.

Example. Is there a derivation $aba \Rightarrow^* babab$ in the system $T = (\{a, b\}, R)$ with the rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

$aba \Rightarrow$

The **word problem** of semi-Thue systems asks for a given semi-Thue system $T = (\Delta, R)$ and given words $x, y \in \Sigma^*$ whether $x \Rightarrow^* y$ in T.

Example. Is there a derivation $aba \Rightarrow^* babab$ in the system $T = (\{a, b\}, R)$ with the rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

$\underline{a}ba \Rightarrow a\underline{aba} \Rightarrow ab\underline{a}b \Rightarrow \underline{aba}ab \Rightarrow babab$

Yes.

The **word problem** of semi-Thue systems asks for a given semi-Thue system $T = (\Delta, R)$ and given words $x, y \in \Sigma^*$ whether $x \Rightarrow^* y$ in T.

Example. Is there a derivation $aba \Rightarrow^* babab$ in the system $T = (\{a, b\}, R)$ with the rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

$\underline{a}ba \Rightarrow a\underline{aba} \Rightarrow ab\underline{a}b \Rightarrow \underline{aba}ab \Rightarrow babab$

Yes.

What about a derivation $babab \Rightarrow^* aba$ in the opposite direction ?

The **word problem** of semi-Thue systems asks for a given semi-Thue system $T = (\Delta, R)$ and given words $x, y \in \Sigma^*$ whether $x \Rightarrow^* y$ in T.

Example. Is there a derivation $aba \Rightarrow^* babab$ in the system $T = (\{a, b\}, R)$ with the rewrite rules

$$\begin{array}{lcl} bb & \longrightarrow & b, \\ aba & \longrightarrow & bab, \\ a & \longrightarrow & aa. \end{array}$$

$\underline{a}ba \Rightarrow a\underline{aba} \Rightarrow ab\underline{a}b \Rightarrow \underline{aba}ab \Rightarrow babab$

Yes.

What about a derivation $babab \Rightarrow^* aba$ in the opposite direction ?

No. If a word begins with the letter b then all derived words also begin with the letter b.

In general, it is difficult to solve the word problem. The problem is undecidable.

In general, it is difficult to solve the word problem. The problem is undecidable.

But we prove more. The **individual word problem** associated to a fixed semi-Thue system T and a fixed target word y asks:

"Does a given word x derive y in T ?"

The difference to the word problem is that T and x are not part of the input, but are rather **fixed**. Different choices of T and y induce different decision problems. For some T and y the problem is decidable, but we prove in the following the for a suitable choice of T and y the problem is undecidable:

Theorem. There exists a semi-Thue system T and a fixed word y such that the individual word problem "Does a given word x derive y in T ?" is undecidable.

(Note that it immediately follows that the word problem is undecidable as well.)

To prove undecidability we construct for any given Turing machine M a semi-Thue system T_M that simulates the Turing machine.

This is not difficult since the operations of a Turing machine are essentially rewrite instructions on its instantaneous descriptions. All rewriting is done locally and can hence be expressed as rewrite rules.

To prove undecidability we construct for any given Turing machine M a semi-Thue system T_M that simulates the Turing machine.

This is not difficult since the operations of a Turing machine are essentially rewrite instructions on its instantaneous descriptions. All rewriting is done locally and can hence be expressed as rewrite rules.

The only complication is caused by the **left and right ends** of the ID where the tape may need to be extended of shrank by adding or removing blank tape symbols.

Note that a rewrite rule cannot recognize the end of a word so no specific rules can be defined for the left and right end. For this reason we add to the ends of the ID new symbols [and]. A Turing machine ID α will then be represented as the word $[\alpha]$.

The system T_M is build in such a way that, for any IDs α and β of M,

$$\alpha \vdash \beta \text{ in } M \qquad \Longleftrightarrow \qquad [\alpha] \Rightarrow [\beta] \text{ in } T_M$$

In detail: Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$ be a Turing machine where $Q \cap \Gamma = \emptyset$. The corresponding semi-Thue system is $T_M = (\Delta, R)$ where

$$\Delta = Q \cup \Gamma \cup \{[,]\},$$

and R contains the productions described next.

(Left moves)

1. Every transition $\delta(q,a) = (p,b,L)$ where $b \neq B$ admits in M the moves

$$\begin{aligned} \ldots xqa\ldots &\vdash \ldots pxb\ldots \qquad \text{for all } x \in \Gamma \\ qa\ldots &\vdash pBb\ldots \end{aligned}$$

(where "$\ldots$" indicates any content that does not change.)

(Left moves)

1. Every transition $\delta(q, a) = (p, b, L)$ where $b \neq B$ admits in M the moves

$$\begin{array}{rcll} \ldots xqa\ldots & \vdash & \ldots pxb\ldots & \qquad \text{for all } x \in \Gamma \\ qa\ldots & \vdash & pBb\ldots & \end{array}$$

(where "…" indicates any content that does not change.)

Thus we add in R the productions

$$\begin{array}{rcll} xqa & \longrightarrow & pxb & \qquad \text{for all } x \in \Gamma \\ [qa & \longrightarrow & [pBb & \end{array}$$

(Left moves)

2. Every transition $\delta(q,a) = (p,b,L)$ where $b = B$ admits in M the moves

$$
\begin{array}{rcll}
\ldots xqay \ldots & \vdash & \ldots pxby \ldots & \text{for all } x, y \in \Gamma \\
\ldots xqa & \vdash & \ldots px & \text{for all } x \in \Gamma \\
qay \ldots & \vdash & pBby \ldots & \text{for all } y \in \Gamma \\
qa & \vdash & pB &
\end{array}
$$

(Left moves)

2. Every transition $\delta(q, a) = (p, b, L)$ where $b = B$ admits in M the moves

$$
\begin{array}{rcll}
\ldots xqay \ldots & \vdash & \ldots pxby \ldots & \text{for all } x, y \in \Gamma \\
\ldots xqa & \vdash & \ldots px & \text{for all } x \in \Gamma \\
qay \ldots & \vdash & pBby \ldots & \text{for all } y \in \Gamma \\
qa & \vdash & pB &
\end{array}
$$

Thus we add in R the productions

$$
\begin{array}{rcll}
xqay & \longrightarrow & pxby & \text{for all } x, y \in \Gamma \\
xqa] & \longrightarrow & px] & \text{for all } x \in \Gamma \\
[qay & \longrightarrow & [pBby & \text{for all } y \in \Gamma \\
[qa] & \longrightarrow & [pB] &
\end{array}
$$

(Right moves)

3. Every transition $\delta(q, a) = (p, b, R)$ where $b \neq B$ admits in M the moves

$$\ldots qay \ldots \ \vdash \ \ldots bpy \ldots \qquad \text{for all } y \in \Gamma$$
$$\ldots qa \vdash \ldots bpB$$

(Right moves)

3. Every transition $\delta(q, a) = (p, b, R)$ where $b \neq B$ admits in M the moves

$$\begin{array}{rcll} \ldots qay \ldots & \vdash & \ldots bpy \ldots & \quad \text{for all } y \in \Gamma \\ \ldots qa & \vdash & \ldots bpB & \end{array}$$

Thus we add in R the productions

$$\begin{array}{rcll} qay & \longrightarrow & bpy & \quad \text{for all } y \in \Gamma \\ qa] & \longrightarrow & bpB] & \end{array}$$

(Right moves)

4. Every transition $\delta(q, a) = (p, b, R)$ where $b = B$ admits in M the moves

$$\begin{array}{rcll} \ldots xqay\ldots & \vdash & \ldots xbpy\ldots & \text{for all } x, y \in \Gamma \\ \ldots xqa & \vdash & \ldots xbpB & \text{for all } x \in \Gamma \\ qay\ldots & \vdash & py\ldots & \text{for all } y \in \Gamma \\ qa & \vdash & pB & \end{array}$$

(Right moves)

4. Every transition $\delta(q, a) = (p, b, R)$ where $b = B$ admits in M the moves

$$\begin{array}{rcll} \ldots xqay \ldots & \vdash & \ldots xbpy \ldots & \text{for all } x, y \in \Gamma \\ \ldots xqa & \vdash & \ldots xbpB & \text{for all } x \in \Gamma \\ qay \ldots & \vdash & py \ldots & \text{for all } y \in \Gamma \\ qa & \vdash & pB & \end{array}$$

Thus we add in R the productions

$$\begin{array}{rcll} xqay & \longrightarrow & xbpy & \text{for all } x, y \in \Gamma \\ xqa] & \longrightarrow & xbpB] & \text{for all } x \in \Gamma \\ {[qay} & \longrightarrow & [py & \text{for all } y \in \Gamma \\ {[qa]} & \longrightarrow & [pB] & \end{array}$$

Example. Suppose M has states q and p and transitions

$$\begin{aligned} \delta(q, a) &= (q, B, R), \\ \delta(q, B) &= (p, a, R), \\ \delta(p, B) &= (q, B, L). \end{aligned}$$

A computation

$$qaaa \vdash qaa \vdash qa \vdash qB \vdash apB \vdash qa$$

has the corresponding derivation

$$[qaaa] \Rightarrow [qaa] \Rightarrow [qa] \Rightarrow [qB] \Rightarrow [apB] \Rightarrow [qa]$$

that uses (in this order) the following rewrite rules:

$$\begin{aligned} [qaa &\longrightarrow [qa \\ [qaa &\longrightarrow [qa \\ [qa] &\longrightarrow [qB] \\ qB] &\longrightarrow apB] \\ apB] &\longrightarrow qa] \end{aligned}$$

The rewrite rules simulate the TM moves in all possible situations (also regarding the ends of the ID), so it is clear that

$$\alpha \vdash \beta \text{ in } M \qquad \Longrightarrow \qquad [\alpha] \Rightarrow [\beta] \text{ in } T_M$$

The rewrite rules simulate the TM moves in all possible situations (also regarding the ends of the ID), so it is clear that

$$\alpha \vdash \beta \text{ in } M \quad \Longrightarrow \quad [\alpha] \Rightarrow [\beta] \text{ in } T_M$$

Conversely, if $\alpha = \ldots qa \ldots$ is an ID of M then a rewrite rule of T_M can be applied on $[\alpha]$ if and only if $\delta(q, a)$ is defined, and in this case the derivation step is unique. (The left-hand-side of every rewrite rule has a unique occurrence of a subword belonging to $Q\Gamma$. The symbols surrounding this qa and the value of $\delta(q, a)$ uniquely identify the applicable rewrite rule.) Thus

$$[\alpha] \Rightarrow w \text{ in } T_M \quad \Longrightarrow \quad w = [\beta] \text{ and } \alpha \vdash \beta \text{ in } M$$

The rewrite rules simulate the TM moves in all possible situations (also regarding the ends of the ID), so it is clear that

$$\alpha \vdash \beta \text{ in } M \qquad \Longrightarrow \qquad [\alpha] \Rightarrow [\beta] \text{ in } T_M$$

Conversely, if $\alpha = \ldots qa \ldots$ is an ID of M then a rewrite rule of T_M can be applied on $[\alpha]$ if and only if $\delta(q,a)$ is defined, and in this case the derivation step is unique. (The left-hand-side of every rewrite rule has a unique occurrence of a subword belonging to $Q\Gamma$. The symbols surrounding this qa and the value of $\delta(q,a)$ uniquely identify the applicable rewrite rule.) Thus

$$[\alpha] \Rightarrow w \text{ in } T_M \qquad \Longrightarrow \qquad w = [\beta] \text{ and } \alpha \vdash \beta \text{ in } M$$

In summary we have:

Lemma. Let α be an ID of the Turing machine M and let T_M be the corresponding semi-Thue system. Then, for $w \in \Delta^*$,

$$[\alpha] \Rightarrow w \qquad \text{if and only if} \qquad w = [\beta] \text{ and } \alpha \vdash \beta.$$

Let ι_w be the initial ID of M for the input w. A simple induction shows that a derivation

$$[\iota_w] \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n$$

exists in T_M if and only if

$$w_1 = [\alpha_1],\ w_2 = [\alpha_2],\ \ldots,\ w_n = [\alpha_n]$$

for IDs $\alpha_1, \alpha_2, \ldots, \alpha_n$ such that

$$\iota_w \vdash \alpha_1 \vdash \alpha_2 \vdash \cdots \vdash \alpha_n$$

Let ι_w be the initial ID of M for the input w. A simple induction shows that a derivation

$$[\iota_w] \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n$$

exists in T_M if and only if

$$w_1 = [\alpha_1],\ w_2 = [\alpha_2],\ \ldots,\ w_n = [\alpha_n]$$

for IDs $\alpha_1, \alpha_2, \ldots, \alpha_n$ such that

$$\iota_w \vdash \alpha_1 \vdash \alpha_2 \vdash \cdots \vdash \alpha_n$$

We want to reduce the problem of whether M accepts a given input w to the problem of whether $x \Rightarrow^* y$ in T_M. Thus we set $x = [\iota_w]$ and $y = [\alpha]$ for an accepting ID α. A problem remains that we do not know what the accepting ID α is.

Fortunately we may assume that the TM always erases its tape content before entering the final state f, so that the only accepting ID is fB:

Lemma. For every Turing machine M there exists a Turing machine M' such that $L(M') = L(M)$ and every accepting computation in M' ends in the instantaneous description fB.

Proof.

Now we can prove the main result:

Theorem. There exists a semi-Thue system T and a fixed word y such that the individual word problem “Does a given word x derive y in T ?” is undecidable.

Proof.

Now we can prove the main result:

Theorem. There exists a semi-Thue system T and a fixed word y such that the individual word problem "Does a given word x derive y in T ?" is undecidable.

Proof. Let M be a Turing machine such that $L(M)$ is not recursive. (For example, $L(M) = L_u$.) By the previous lemma we may assume that every accepting computation by M ends in fB.

Let $T = T_M$ be the semi-Thue system for this TM M, and let $y = [fB]$.

Now we can prove the main result:

Theorem. There exists a semi-Thue system T and a fixed word y such that the individual word problem "Does a given word x derive y in T ?" is undecidable.

Proof. Let M be a Turing machine such that $L(M)$ is not recursive. (For example, $L(M) = L_u$.) By the previous lemma we may assume that every accepting computation by M ends in fB.

Let $T = T_M$ be the semi-Thue system for this TM M, and let $y = [fB]$.

Assume, to the contrary of the claim, that there exists an algorithm A to determine if a given word x derives $[fB]$ in T_M.

Then we have an algorithm to decide if a given word w is in $L(M)$: Call A with input $x = [\iota_w]$ and return the answer that A gives.

This algorithm works: $w \in L(M) \iff \iota_w \vdash^* fB \iff [\iota_w] \Rightarrow^* [fB]$

But this algorithm cannot exist because $L(M)$ is not recursive, so algorithm A does not exist either.

Remark 1. By adding the production $[fB] \longrightarrow \varepsilon$, we can choose $y = \varepsilon$ in the theorem. We obtain a semi-Thue system in which it is undecidable whether a given word derives the empty word.

Remark 1. By adding the production $[fB] \longrightarrow \varepsilon$, we can choose $y = \varepsilon$ in the theorem. We obtain a semi-Thue system in which it is undecidable whether a given word derives the empty word.

Remark 2. It is known that there exists a semi-Thue system (Δ, R) with $|\Delta| = 2$ and $|R| = 3$ whose individual word problem is undecidable. The decidability status for semi-Thue systems with two productions is not known.

A **Thue system** $T = (\Delta, R)$ is a special kind of semi-Thue system where $u \longrightarrow v$ is a production if and only if $v \longrightarrow u$ is a production. In other words, all rewrite-rules may be applied in both directions.

We write the productions of a Thue system as $u \longleftrightarrow v$ and we denote one step derivations using $\Leftrightarrow$ instead of $\Rightarrow$. Clearly $\Leftrightarrow^*$ is an equivalence relation on Δ^*. In fact, it is a congruence of the monoid Δ^*:

Lemma. Let $T = (\Delta, R)$ be a Thue system. Then $\Leftrightarrow^*$ is an equivalence relation, and

$$\begin{cases} u_1 \Leftrightarrow^* v_1 \\ u_2 \Leftrightarrow^* v_2 \end{cases} \quad \text{imply that} \quad u_1 u_2 \Leftrightarrow^* v_1 v_2.$$

Proof.

Algebraically: The Thue system T is a finite presentation of the **quotient monoid** $\Delta^*/\Leftrightarrow^*$ whose elements are the equivalence classes.

The word problem of T is then the question of whether two given words represent the same element in the quotient monoid.

Theorem. There exists a Thue system T and a fixed word y such that the individual word problem "Does a given word x derive y in T ?" is undecidable.

Proof.

Theorem. There exists a Thue system T and a fixed word y such that the individual word problem "Does a given word x derive y in T ?" is undecidable.

Proof. We use the following two properties of the semi-Thue system T_M that we constructed based on a Turing machine M:

(i) The rewriting is deterministic: If $w \in \Delta^*$ contains a single occurrence of a letter from Q then there is at most one $z \in \Delta^*$ such that $w \Rightarrow z$.

(ii) In each production $u \longrightarrow v$ both u and v contain a single occurrence of a letter from Q.

Theorem. There exists a Thue system T and a fixed word y such that the individual word problem "Does a given word x derive y in T ?" is undecidable.

Proof. We use the following two properties of the semi-Thue system T_M that we constructed based on a Turing machine M:

(i) The rewriting is deterministic: If $w \in \Delta^*$ contains a single occurrence of a letter from Q then there is at most one $z \in \Delta^*$ such that $w \Rightarrow z$.

(ii) In each production $u \longrightarrow v$ both u and v contain a single occurrence of a letter from Q.

We now interpret T_M as a Thue system: each production may be applied in either direction. Denote

$u \Rightarrow v$ for derivation steps in the semi-Thue system T_M,
$u \Leftarrow v$ if $v \Rightarrow u$, (*i.e.*, for applications of reverse productions),
$u \Leftrightarrow v$ if $u \Rightarrow v$ or $u \Leftarrow v$, (*i.e.*, for derivation steps in the Thue system).

As in the proof for semi-Thue systems: Let M be a Turing machine such that $L(M)$ is not recursive, and assume w.l.o.g. that every accepting computation by M ends in fB.

Let us prove that in T_M, for every input word $w \in \Sigma^*$,

$$[\iota_w] \Rightarrow^* [fB] \qquad \text{if and only if} \qquad [\iota_w] \Leftrightarrow^* [fB]$$

As in the proof for semi-Thue systems: Let M be a Turing machine such that $L(M)$ is not recursive, and assume w.l.o.g. that every accepting computation by M ends in fB.

Let us prove that in T_M, for every input word $w \in \Sigma^*$,

$$[\iota_w] \Rightarrow^* [fB] \qquad \text{if and only if} \qquad [\iota_w] \Leftrightarrow^* [fB]$$

This is then sufficient to prove undecidability: Choose $y = [fB]$, and assume that there is an algorithm A to dcide if a given word x derives $[fB]$ in the Thue system T_M.

The following algorithm determines if a given word w is in $L(M)$: Call A with input $x = [\iota_w]$ and return the answer that A gives.

This algorithm works:

$$w \in L(M) \iff \iota_w \vdash^* fB \iff [\iota_w] \Rightarrow^* [fB] \iff [\iota_w] \Leftrightarrow^* [fB]$$

But this algorithm cannot exist because $L(M)$ is not recursive, so algorithm A does not exist either.

$$[\iota_w] \Rightarrow^* [fB] \qquad \text{if and only if} \qquad [\iota_w] \Leftrightarrow^* [fB]$$

$$[\iota_w] \Rightarrow^* [fB] \qquad \text{if and only if} \qquad [\iota_w] \Leftrightarrow^* [fB]$$

Direction $\Longrightarrow$ is trivial, so let us prove $\Longleftarrow$. Let w be such that $[\iota_w] \Leftrightarrow^* [fB]$. Consider a **shortest derivation**

$$[\iota_w] = w_0 \Leftrightarrow w_1 \Leftrightarrow w_2 \Leftrightarrow \cdots \Leftrightarrow w_n = [fB]$$

$$[\iota_w] \Rightarrow^* [fB] \qquad \text{if and only if} \qquad [\iota_w] \Leftrightarrow^* [fB]$$

Direction $\Longrightarrow$ is trivial, so let us prove $\Longleftarrow$. Let w be such that $[\iota_w] \Leftrightarrow^* [fB]$. Consider a **shortest derivation**

$$[\iota_w] = w_0 \Leftrightarrow w_1 \Leftrightarrow w_2 \Leftrightarrow \cdots \Leftrightarrow w_n = [fB]$$

It is enough to show that all derivation steps are in the forward direction $\Rightarrow$. Suppose the contrary: some derivation steps are in the reverse direction $\Leftarrow$. Let i be the largest index such that $w_{i-1} \Leftarrow w_i$. Because no forward production can be applied on $[fB]$, we have that $i < n$. Thus

$$w_{i-1} \Leftarrow w_i \Rightarrow w_{i+1}$$

$$[\iota_w] \Rightarrow^* [fB] \qquad \text{if and only if} \qquad [\iota_w] \Leftrightarrow^* [fB]$$

Direction $\Longrightarrow$ is trivial, so let us prove $\Longleftarrow$. Let w be such that $[\iota_w] \Leftrightarrow^* [fB]$. Consider a **shortest derivation**

$$[\iota_w] = w_0 \Leftrightarrow w_1 \Leftrightarrow w_2 \Leftrightarrow \cdots \Leftrightarrow w_n = [fB]$$

It is enough to show that all derivation steps are in the forward direction $\Rightarrow$. Suppose the contrary: some derivation steps are in the reverse direction $\Leftarrow$. Let i be the largest index such that $w_{i-1} \Leftarrow w_i$. Because no forward production can be applied on $[fB]$, we have that $i < n$. Thus

$$w_{i-1} \Leftarrow w_i \Rightarrow w_{i+1}$$

By the property "(ii) In each production $u \longrightarrow v$ both u and v contain a single occurrence of a letter from Q" we have that w_i contains exactly one occurrence of a state letter.

$$[\iota_w] \Rightarrow^* [fB] \qquad \text{if and only if} \qquad [\iota_w] \Leftrightarrow^* [fB]$$

Direction $\Longrightarrow$ is trivial, so let us prove $\Longleftarrow$. Let w be such that $[\iota_w] \Leftrightarrow^* [fB]$. Consider a **shortest derivation**

$$[\iota_w] = w_0 \Leftrightarrow w_1 \Leftrightarrow w_2 \Leftrightarrow \cdots \Leftrightarrow w_n = [fB]$$

It is enough to show that all derivation steps are in the forward direction $\Rightarrow$. Suppose the contrary: some derivation steps are in the reverse direction $\Leftarrow$. Let i be the largest index such that $w_{i-1} \Leftarrow w_i$. Because no forward production can be applied on $[fB]$, we have that $i < n$. Thus

$$w_{i-1} \Leftarrow w_i \Rightarrow w_{i+1}$$

By the property "(ii) In each production $u \longrightarrow v$ both u and v contain a single occurrence of a letter from Q" we have that w_i contains exactly one occurrence of a state letter.

Now the property "(i) The rewriting is deterministic: If $w \in \Delta^*$ contains a single occurrence of a letter from Q then there is at most one $z \in \Delta^*$ such that $w \Rightarrow z$." implies that $w_{i-1} = w_{i+1}$.

But this means that the derivation can be shortened by removing the unnecessary w_i and w_{i+1} from the derivation, a contradiction.

Remark 1. The theorem means that the word problem is undecidable in some **finitely presented monoids**. The result can be strengthened further: It is known that the word problem is undecidable even in some **finitely presented groups**. On the other hand, the word problem is decidable among finitely presented abelian (*i.e.*, commutative) monoids and groups.

Remark 1. The theorem means that the word problem is undecidable in some **finitely presented monoids**. The result can be strengthened further: It is known that the word problem is undecidable even in some **finitely presented groups**. On the other hand, the word problem is decidable among finitely presented abelian (*i.e.*, commutative) monoids and groups.

Remark 2. By adding $[fB] \longleftrightarrow \varepsilon$ to the production set, we can choose $u = \varepsilon$ in the theorem. In this case, it is undecidable if a given word represents the identity element of the monoid. (A small proof is needed to show that shortest derivations $[\iota_w] \Leftrightarrow^* [fB]$ do not use the new production $[fB] \longleftrightarrow \varepsilon$.)

Remark 1. The theorem means that the word problem is undecidable in some **finitely presented monoids**. The result can be strengthened further: It is known that the word problem is undecidable even in some **finitely presented groups**. On the other hand, the word problem is decidable among finitely presented abelian (*i.e.*, commutative) monoids and groups.

Remark 2. By adding $[fB] \longleftrightarrow \varepsilon$ to the production set, we can choose $u = \varepsilon$ in the theorem. In this case, it is undecidable if a given word represents the identity element of the monoid. (A small proof is needed to show that shortest derivations $[\iota_w] \Leftrightarrow^* [fB]$ do not use the new production $[fB] \longleftrightarrow \varepsilon$.)

Remark 3. There exists a Thue system (Δ, R) with $|\Delta| = 2$ and $|R| = 3$ whose word problem is undecidable.

A **type-0 grammar** (or simply a **grammar**) is a 4-tuple $G = (V, T, P, S)$ where

- V and T are disjoint finite alphabets of **variables** and **terminals**,
- $S \in V$ is the **start symbol**, and
- P is a finite set of **productions** $u \longrightarrow v$ where u and v are words over $V \cup T$ and u contains at least one variable.

The pair $(V \cup T, P)$ acts as a semi-Thue system: we denote $w \Rightarrow w'$ if w' is obtained from w by replacing a subword u by v, for some $u \longrightarrow v \in P$.

If $w \in T^*$ is terminal then no derivation is possible from w. The language

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

generated by G consists of all the terminal words that can be derived from the start symbol S.

Example. The grammar $G = (\{S, X, Y, Z\}, \{a, b, c\}, P, S)$ with productions

$$\begin{aligned} S &\longrightarrow aXbc \mid \varepsilon \\ X &\longrightarrow \varepsilon \mid aYb \\ Yb &\longrightarrow bY \\ Yc &\longrightarrow Zcc \\ bZ &\longrightarrow Zb \\ aZ &\longrightarrow aX \end{aligned}$$

generates the language

$$L(G) = \{a^n b^n c^n \mid n \geq 0\}.$$

Type-0 grammars generate precisely the family of recursively enumerable languages:

Theorem. Turing machines and type-0 grammars are effectively equivalent.

Proof.

Type-0 grammars generate precisely the family of recursively enumerable languages:

Theorem. Turing machines and type-0 grammars are effectively equivalent.

Proof. For the direction

$$\text{Grammar} \longrightarrow \text{Turing machine}$$

we rely on the fact that any semi-algorithm can be implemented on a Turing machine.

Let G be a type-0 grammar. The following (non-deterministic) semi-algorithm determines for a given word w whether $w \in L(G)$: Guess a derivation

$$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$$

and verify that the derivation is indeed a valid derivation and that $\alpha_n = w$.

Because membership in $L(G)$ is semi-decidable, the language $L(G)$ is recursively enumerable. Thus there exists a Turing machine M such that $L(M) = L(G)$. This M can also be effectively constructed.

Grammar $\longrightarrow$ Turing machine

Alternatively, a concrete non-deterministic TM M to recognize $L(G)$ could repeatedly

- scan the content of the tape from left-to-right,
- guess during the scan a position and a production $u \longrightarrow v$ to use,
- check that v appears on the tape at the guessed position,
- replace v by u (this may require shifting the suffix if $|v| \neq |u|$),
- check if the tape content is S, in which case accept (go to the final state)

Note that M is searching for a backward derivation from the input word w to the start symbol S.

Then the converse direction:

$$\text{Turing machine} \longrightarrow \text{Grammar}$$

Remark: Turing machines are accepting devices, grammars are generative. To convert a Turing machine into an equivalent grammar we therefore need to "reverse" the computation so that we start generating from the accepting ID backwards in time, until an initial ID is reached.

Turing machine $\longrightarrow$ Grammar

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$ be a given Turing machine. We may assume that every accepting computation ends in fB. Let us construct a type-0 grammar $G = (V, \Sigma, P, S)$ such that $L(G) = L(M)$.

Turing machine $\longrightarrow$ Grammar

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, f)$ be a given Turing machine. We may assume that every accepting computation ends in fB. Let us construct a type-0 grammar $G = (V, \Sigma, P, S)$ such that $L(G) = L(M)$.

Let $T_M = (\Delta, P)$ be the semi-Thue system associated to machine M. Let

$$P^R = \{v \longrightarrow u \mid u \longrightarrow v \in P\}$$

be the set of reverse productions. Then we know that with productions P^R

$$[fB] \Rightarrow^* [\iota_w] \qquad \text{if and only if} \qquad w \in L(M).$$

We also need productions that admit initializing and finalizing derivations

$$S \Rightarrow^* [fB] \qquad \text{and} \qquad [\iota_w] \Rightarrow^* w.$$

So in the grammar G we have all the productions of P^R. To initialize derivations we add a new start symbol S and the production

$$S \longrightarrow [fB]$$

And to finalize (derive from $[\iota_w]$ the word w) we add the following productions:

$$\begin{array}{rcll} [q_0 & \longrightarrow & \# & \\ \#a & \longrightarrow & a\# & \quad \text{for all } a \in \Sigma \\ \#] & \longrightarrow & \varepsilon & \\ [q_0 B] & \longrightarrow & \varepsilon & \end{array}$$

where $\#$ is a new variable.

So in the grammar G we have all the productions of P^R. To initialize derivations we add a new start symbol S and the production

$$S \longrightarrow [fB]$$

And to finalize (derive from $[\iota_w]$ the word w) we add the following productions:

$$\begin{array}{rcll} [q_0 & \longrightarrow & \# & \\ \#a & \longrightarrow & a\# & \quad \text{for all } a \in \Sigma \\ \#] & \longrightarrow & \varepsilon & \\ [q_0B] & \longrightarrow & \varepsilon & \end{array}$$

where $\#$ is a new variable.

The finalization works: The last production directly maps $[\iota_\varepsilon] \Rightarrow \varepsilon$. And for any non-empty $w = a_1 \ldots a_n$ we have the derivation

$$[\iota_w] = [q_0 a_1 \ldots a_n] \Rightarrow \#a_1 \ldots a_n] \Rightarrow^* a_1 \ldots a_n\#] \Rightarrow a_1 \ldots a_n = w$$

So in the grammar G we have all the productions of P^R. To initialize derivations we add a new start symbol S and the production

$$S \longrightarrow [fB]$$

And to finalize (derive from $[\iota_w]$ the word w) we add the following productions:

$$\begin{array}{rcll} [q_0 & \longrightarrow & \# & \\ \#a & \longrightarrow & a\# & \quad \text{for all } a \in \Sigma \\ \#] & \longrightarrow & \varepsilon & \\ [q_0 B] & \longrightarrow & \varepsilon & \end{array}$$

where $\#$ is a new variable.

The finalization works: The last production directly maps $[\iota_\varepsilon] \Rightarrow \varepsilon$. And for any non-empty $w = a_1 \ldots a_n$ we have the derivation

$$[\iota_w] = [q_0 a_1 \ldots a_n] \Rightarrow \#a_1 \ldots a_n] \Rightarrow^* a_1 \ldots a_n\#] \Rightarrow a_1 \ldots a_n = w$$

The symbols of our grammar are thus the elements of

$$Q \cup \Gamma \cup \{[,], \#, S\}$$

Out of these, the symbols in $\Sigma \subseteq \Gamma$ are terminals, all others are variables. Note that every production contains variables on the left-hand-side, as required.

The grammar G we constructed satisfies $L(M) = L(G)$:

"$\subseteq$" If $w \in L(M)$ then $[fB] \Rightarrow^* [\iota_w]$ using P^R. Hence, G admits the derivation

$$S \Rightarrow [fB] \Rightarrow^* [\iota_w] \Rightarrow^* w$$

and so $w \in L(G)$.

The grammar G we constructed satisfies $L(M) = L(G)$:

"$\supseteq$" Let $w \in L(G)$, and consider its derivation $S \Rightarrow^* w$.

The grammar G we constructed satisfies $L(M) = L(G)$:

"$\supseteq$" Let $w \in L(G)$, and consider its derivation $S \Rightarrow^* w$.

All derivations begin $S \Rightarrow [fB]$. With the productions P^R one can then only reach words $[zqy]$ where $q \in Q$. In order to derive a word without a state symbol, a production $[q_0 B] \longrightarrow \varepsilon$ or $[q_0 \longrightarrow \#$ needs to be used.

The grammar G we constructed satisfies $L(M) = L(G)$:

"$\supseteq$" Let $w \in L(G)$, and consider its derivation $S \Rightarrow^* w$.

All derivations begin $S \Rightarrow [fB]$. With the productions P^R one can then only reach words $[zqy]$ where $q \in Q$. In order to derive a word without a state symbol, a production $[q_0B] \longrightarrow \varepsilon$ or $[q_0 \longrightarrow \#$ needs to be used.

- One can use $[q_0B] \longrightarrow \varepsilon$ only if $[q_0B] = [\iota_\varepsilon]$ was reached from $[fB]$ using P^R. Then $\varepsilon \in L(M)$, and the derived word $w = \varepsilon$.

The grammar G we constructed satisfies $L(M) = L(G)$:

"$\supseteq$" Let $w \in L(G)$, and consider its derivation $S \Rightarrow^* w$.

All derivations begin $S \Rightarrow [fB]$. With the productions P^R one can then only reach words $[zqy]$ where $q \in Q$. In order to derive a word without a state symbol, a production $[q_0B] \longrightarrow \varepsilon$ or $[q_0 \longrightarrow \#$ needs to be used.

- One can use $[q_0B] \longrightarrow \varepsilon$ only if $[q_0B] = [\iota_\varepsilon]$ was reached from $[fB]$ using P^R. Then $\varepsilon \in L(M)$, and the derived word $w = \varepsilon$.
- One can use $[q_0 \longrightarrow \#$ only if $[q_0y] = [\iota_y]$ was derived using P^R, for some y. The terminal derivation must continue as

$$[q_0y] \Rightarrow \#y] \Rightarrow^* y\#] \Rightarrow y$$

so that the derived word $w = y$. As $[\iota_w]$ was derived from $[fB]$ using P^R, we have $w \in L(M)$.

The grammar G we constructed satisfies $L(M) = L(G)$:

"$\supseteq$" Let $w \in L(G)$, and consider its derivation $S \Rightarrow^* w$.

All derivations begin $S \Rightarrow [fB]$. With the productions P^R one can then only reach words $[zqy]$ where $q \in Q$. In order to derive a word without a state symbol, a production $[q_0B] \longrightarrow \varepsilon$ or $[q_0 \longrightarrow \#$ needs to be used.

- One can use $[q_0B] \longrightarrow \varepsilon$ only if $[q_0B] = [\iota_\varepsilon]$ was reached from $[fB]$ using P^R. Then $\varepsilon \in L(M)$, and the derived word $w = \varepsilon$.
- One can use $[q_0 \longrightarrow \#$ only if $[q_0y] = [\iota_y]$ was derived using P^R, for some y. The terminal derivation must continue as

  $$[q_0y] \Rightarrow \#y] \Rightarrow^* y\#] \Rightarrow y$$

  so that the derived word $w = y$. As $[\iota_w]$ was derived from $[fB]$ using P^R, we have $w \in L(M)$.

In all cases $w \in L(M)$. In fact, we saw that all terminal derivations have the structure

$$S \Rightarrow [fB] \Rightarrow^* [\iota_w] \Rightarrow^* w.$$

Post correspondence problem (PCP)

An instance to PCP consists of two lists of words over some alphabet Σ:

$$\begin{array}{lcl} L_1 & : & w_1,\ w_2,\ \ldots\ w_k \\ L_2 & : & x_1,\ x_2,\ \ldots\ x_k \end{array}$$

Both lists contain equally many words.

A solution to the instance is any non-empty string $i_1 i_2 \ldots i_m$ of indices from $\{1, 2, \ldots, k\}$ such that

$$w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}$$

In other words, we concatenate corresponding words w_i and x_i to form two words. We have a solution if the concatenated w_i's form the same word as the corresponding concatenated x_i's.

The PCP asks whether a given instance has a solution or not. It turns out that PCP is undecidable.

Example 1. Consider the following two lists:

$$\begin{array}{lllll} L_1 & : & aa, & bb, & abb \\ L_2 & : & aab, & ba, & b \end{array}$$

Example 1. Consider the following two lists:

$$\begin{aligned} L_1 &: aa, \quad bb, \quad abb \\ L_2 &: aab, \quad ba, \quad b \end{aligned}$$

This instance has solution 1213 because

$$\begin{aligned} w_1 w_2 w_1 w_3 &= aa\ bb\ aa\ abb \\ x_1 x_2 x_1 x_3 &= aab\ ba\ aab\ b \end{aligned}$$

are identical.

Example 2. Consider the PCP instance

$$\begin{aligned} L_1 &: \ aab, \quad a \\ L_2 &: \ aa, \quad baa \end{aligned}$$

Example 2. Consider the PCP instance

$$\begin{array}{lcll} L_1 & : & aab, & a \\ L_2 & : & aa, & baa \end{array}$$

Lets try to find a solution:

aab
aa

Index word: 1

(First index cannot be 2 as it would fail in the first letter.)

Example 2. Consider the PCP instance

$$\begin{aligned} L_1 &: aab, \quad a \\ L_2 &: aa, \quad baa \end{aligned}$$

Lets try to find a solution:

$aaba$
$aabaa$

Index word: 12

(Second index cannot be 1 as it would fail in the third letter.)

Example 2. Consider the PCP instance

$$\begin{array}{lcll} L_1 & : & aab, & a \\ L_2 & : & aa, & baa \end{array}$$

Lets try to find a solution:

$aaba\textcolor{red}{a}$
$aabaa\textcolor{blue}{baa}$

Index word: 122

(Third index cannot be 1 as it would fail in the 7'th letter.)

Example 2. Consider the PCP instance

$$\begin{array}{lcll} L_1 & : & aab, & a \\ L_2 & : & aa, & baa \end{array}$$

Lets try to find a solution:

$aabaa$
$aabaabaa$

Index word: 122 ?

(Both 1 and 2 fail as the fourth index.)

This PCP instance does not have a solution.

Example 3. The PCP instance

$$\begin{aligned} L_1 &= a, \quad ba \\ L_2 &= ab, \quad ab \end{aligned}$$

Example 3. The PCP instance

$$\begin{aligned} L_1 &= a, \quad ba \\ L_2 &= ab, \quad ab \end{aligned}$$

Lets try to find a solution:

a
ab

Index word: 1

Example 3. The PCP instance

$$\begin{aligned} L_1 &= a, \quad ba \\ L_2 &= ab, \quad ab \end{aligned}$$

Lets try to find a solution:

aba
$abab$

Index word: 12

Example 3. The PCP instance

$$
\begin{aligned}
L_1 &= a, \quad ba \\
L_2 &= ab, \quad ab
\end{aligned}
$$

Lets try to find a solution:

$ababa$
$ababab$

Index word: 122

Example 3. The PCP instance

$$\begin{aligned} L_1 &= a, \quad ba \\ L_2 &= ab, \quad ab \end{aligned}$$

Lets try to find a solution:

$ababa\mathbf{ba}$
$ababab\mathbf{ab}$

Index word: $1222\ldots$

The first list can never catch-up the missing b, so a solution does not exist. (Note: we are asking for finite solutions!)

PCP can be expressed in terms of **homomorphisms**: The problem asks for two given homomorphisms

$$h_1, h_2 : \Delta^* \longrightarrow \Sigma^*$$

whether there exists a non-empty word u such that $h_1(u) = h_2(u)$.

The connection to the list formulation

$$\begin{array}{lllll} L_1 & : & w_1, & w_2, & \ldots & w_k \\ L_2 & : & x_1, & x_2, & \ldots & x_k \end{array}$$

is as follows:

$$\Delta = \{1, 2, \ldots, k\}$$
$$h_1(i) = w_i \text{ for all } i \in \Delta$$
$$h_2(i) = x_i \text{ for all } i \in \Delta$$

For any index sequence $u \in \Delta^+$ then $h_1(u)$ and $h_2(u)$ are the concatenated words of w_i and x_i as given by the index sequence u.

Example. Recall our first example

$$\begin{array}{ll} L_1 & : \; aa, \quad bb, \quad abb \\ L_2 & : \; aab, \quad ba, \quad b \end{array}$$

that has the solution 1213.

As a pair of homomorphisms: $h_1, h_2 : \{1,2,3\}^* \longrightarrow \{a,b\}^*$ where

$$h_1 : \begin{array}{l} 1 \mapsto aa \\ 2 \mapsto bb \\ 3 \mapsto abb \end{array} \qquad\qquad h_2 : \begin{array}{l} 1 \mapsto aab \\ 2 \mapsto ba \\ 3 \mapsto b \end{array}$$

The solution 1213 means that $h_1(1213) = h_2(1213)$.

Example. Our second example

$$\begin{array}{lcll} L_1 & : & aab, & a \\ L_2 & : & aa, & baa \end{array}$$

in terms of homomorphisms: $h_1, h_2 : \{1,2\}^* \longrightarrow \{a,b\}^*$ where

$$h_1 : \begin{array}{lcl} 1 & \mapsto & aab \\ 2 & \mapsto & a \end{array} \qquad\qquad h_2 : \begin{array}{lcl} 1 & \mapsto & aa \\ 2 & \mapsto & baa \end{array}$$

The instance has no solution, meaning that $h_1(w) \neq h_2(w)$ for all $w \neq \varepsilon$.

Theorem. The Post correspondence problem is undecidable.

Proof. Reduction from the word problem of semi-Thue systems.

Theorem. The Post correspondence problem is undecidable.

Proof. Reduction from the word problem of semi-Thue systems.

It is enough to show how to effectively construct, for a given semi-Thue system $T = (\Sigma, R)$ and source and target words x and y, an equivalent PCP instance L_1, L_2. The instance is equivalent in the sense that it has a solution if and only if $x \Rightarrow^* y$ in T.

Theorem. The Post correspondence problem is undecidable.

Proof. Reduction from the word problem of semi-Thue systems.

It is enough to show how to effectively construct, for a given semi-Thue system $T = (\Sigma, R)$ and source and target words x and y, an equivalent PCP instance L_1, L_2. The instance is equivalent in the sense that it has a solution if and only if $x \Rightarrow^* y$ in T.

Moreover, we may assume w.l.o.g. that $x, y \neq \varepsilon$, and also that $u, v \neq \varepsilon$ in each rewrite rule $u \longrightarrow v$ of R.
(We can make this assumption since the system T_M and the source $[\iota_w]$ and the target $[fB]$ in our proof of undecidability of the word problem have this property. So the word problem is undecidable in such restricted cases.)

Theorem. The Post correspondence problem is undecidable.

Proof. Reduction from the word problem of semi-Thue systems.

It is enough to show how to effectively construct, for a given semi-Thue system $T = (\Sigma, R)$ and source and target words x and y, an equivalent PCP instance L_1, L_2. The instance is equivalent in the sense that it has a solution if and only if $x \Rightarrow^* y$ in T.

Moreover, we may assume w.l.o.g. that $x, y \neq \varepsilon$, and also that $u, v \neq \varepsilon$ in each rewrite rule $u \longrightarrow v$ of R.
(We can make this assumption since the system T_M and the source $[\iota_w]$ and the target $[fB]$ in our proof of undecidability of the word problem have this property. So the word problem is undecidable in such restricted cases.)

Let $\Sigma' = \{a' \mid a \in \Sigma\}$ be a marked, disjoint copy Σ. As usual, denote for every $u \in \Sigma^*$ by u' the word obtained by marking each letter of u.

Let $\#$ be a new marker symbol. The alphabet of the PCP instance is

$$\Sigma \cup \Sigma' \cup \{\#\}$$

The following corresponding pairs of words are placed into lists L_1 and L_2:

L_1	L_2	
$\#x$	$\#$	
$\#$	$y\#$	
a	a'	for every $a \in \Sigma$
a'	a	for every $a \in \Sigma$
v'	u	for every $u \longrightarrow v \in R$

The following corresponding pairs of words are placed into lists L_1 and L_2:

L_1	L_2	
$\#x$	$\#$	
$\#$	$y\#$	
a	a'	for every $a \in \Sigma$
a'	a	for every $a \in \Sigma$
v'	u	for every $u \longrightarrow v \in R$

Lets try to find a solution:

$\#x$ $\qquad\qquad\qquad$ $\#$

$\#$ $\qquad\qquad\qquad$ $y\#$

The only pair in which one of the words is the prefix of the other one is $(\#x, \#)$. Therefore a solution must start with this pair.

Similarly, the only pair in which one word is the suffix of the other one is $(\#, y\#)$ so any solution must end in this pair.

The following corresponding pairs of words are placed into lists L_1 and L_2:

L_1	L_2	
$\#x$	$\#$	
$\#$	$y\#$	
a	a'	for every $a \in \Sigma$
a'	a	for every $a \in \Sigma$
v'	u	for every $u \longrightarrow v \in R$

Lets try to find a solution:

$\#x \; x'_1$
$\#x$

The second word has to catch-up x. The only pairs that can match letters of x are (a', a) for $a \in \Sigma$ and (v', u) for $u \longrightarrow v \in R$. Matching x with such pairs creates a new marked word x'_1 after x in the first word.

Note that x_1 can be obtained from x in the semi-Thue system in some number of derivation steps:

$$x \Rightarrow^* x_1$$

The following corresponding pairs of words are placed into lists L_1 and L_2:

L_1	L_2	
$\#x$	$\#$	
$\#$	$y\#$	
a	a'	for every $a \in \Sigma$
a'	a	for every $a \in \Sigma$
v'	u	for every $u \longrightarrow v \in R$

Lets try to find a solution:

$\#x\ x_1'\ x_1$
$\#x\ x_1'$

Next the second list has to catch-up the primed word x_1'. The only pairs that contain primed letters in the second components are (a, a') for $a \in \Sigma$. Such pairs create a new unprimed copy of x_1 in the first word.

The following corresponding pairs of words are placed into lists L_1 and L_2:

L_1	L_2	
$\#x$	$\#$	
$\#$	$y\#$	
a	a'	for every $a \in \Sigma$
a'	a	for every $a \in \Sigma$
v'	u	for every $u \longrightarrow v \in R$

Lets try to find a solution:

$\#x\ x_1'\ x_1\ x_2'$
$\#x\ x_1'\ x_1$

Now the process is repeated on x_1 instead of x. We create a new word x_2 and

$$x \Rightarrow^* x_1 \Rightarrow^* x_2$$

The following corresponding pairs of words are placed into lists L_1 and L_2:

L_1	L_2	
$\#x$	$\#$	
$\#$	$y\#$	
a	a'	for every $a \in \Sigma$
a'	a	for every $a \in \Sigma$
v'	u	for every $u \longrightarrow v \in R$

Lets try to find a solution:

$\#x\ x_1'\ x_1\ x_2'\ x_2$
$\#x\ x_1'\ x_1\ x_2'$

Now the process is repeated on x_1 instead of x. We create a new word x_2 and

$$x \Rightarrow^* x_1 \Rightarrow^* x_2$$

The following corresponding pairs of words are placed into lists L_1 and L_2:

L_1	L_2	
$\#x$	$\#$	
$\#$	$y\#$	
a	a'	for every $a \in \Sigma$
a'	a	for every $a \in \Sigma$
v'	u	for every $u \longrightarrow v \in R$

Lets try to find a solution:

$$\#x\ x_1'\ x_1\ x_2'\ x_2\ x_3'\ x_3 \ldots\ x_n'\ x_n$$
$$\#x\ x_1'\ x_1\ x_2'\ x_2\ x_3'\ x_3 \ldots\ x_n'$$

Continuing likewise, we are forced to create matching words

$$x \Rightarrow^* x_1 \Rightarrow^* x_2 \Rightarrow^* x_3 \Rightarrow^* \cdots \Rightarrow^* x_n$$

The following corresponding pairs of words are placed into lists L_1 and L_2:

L_1	L_2	
$\#x$	$\#$	
$\#$	$y\#$	
a	a'	for every $a \in \Sigma$
a'	a	for every $a \in \Sigma$
v'	u	for every $u \longrightarrow v \in R$

Lets try to find a solution:

$\#x\ x_1'\ x_1\ x_2'\ x_2\ x_3'\ x_3 \ldots\ x_n'\ x_n$
$\#x\ x_1'\ x_1\ x_2'\ x_2\ x_3'\ x_3 \ldots\ x_n'$

Continuing likewise, we are forced to create matching words

$$x \Rightarrow^* x_1 \Rightarrow^* x_2 \Rightarrow^* x_3 \Rightarrow^* \cdots \Rightarrow^* x_n$$

The second list can catch-up the first list only if for some n we have $x_n = y$. Then the pair $(\#, y\#)$ can be used to close the words.

The following corresponding pairs of words are placed into lists L_1 and L_2:

L_1	L_2	
$\#x$	$\#$	
$\#$	$y\#$	
a	a'	for every $a \in \Sigma$
a'	a	for every $a \in \Sigma$
v'	u	for every $u \longrightarrow v \in R$

Lets try to find a solution:

$$\#x\ x_1'\ x_1\ x_2'\ x_2\ x_3'\ x_3 \ldots\ x_n'\ x_n\#$$
$$\#x\ x_1'\ x_1\ x_2'\ x_2\ x_3'\ x_3 \ldots\ x_n'\ y\ \#$$

Continuing likewise, we are forced to create matching words

$$x \Rightarrow^* x_1 \Rightarrow^* x_2 \Rightarrow^* x_3 \Rightarrow^* \cdots \Rightarrow^* x_n$$

The second list can catch-up the first list only if for some n we have $x_n = y$. Then the pair $(\#, y\#)$ can be used to close the words.

Conclusion: the PCP instance has a solution if and only if $x \Rightarrow^* y$ in the semi-Thue system T.

Example. Consider our earlier rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

and words $x = ababa$ and $y = bab$.

Example. Consider our earlier rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

and words $x = ababa$ and $y = bab$.

The corresponding PCP instance contains 9 pairs:

L_1 :	$\#ababa$	$\#$	a	b	a'	b'	b'	$b'a'b'$	$a'a'$
L_2 :	$\#$	$bab\#$	a'	b'	a	b	bb	aba	a

The word problem has a solution, so the PCP has one too.

Example. Consider our earlier rewrite rules

$$\begin{aligned} bb &\longrightarrow b, \\ aba &\longrightarrow bab, \\ a &\longrightarrow aa. \end{aligned}$$

and words $x = ababa$ and $y = bab$.

The corresponding PCP instance contains 9 pairs:

L_1 :	$\#ababa$	$\#$	a	b	a'	b'	b'	$b'a'b'$	$a'a'$
L_2 :	$\#$	$bab\#$	a'	b'	a	b	bb	aba	a

The word problem has a solution, so the PCP has one too.

$\#ababa$	$b'a'b'$	b'	a'	b	a	b	b	a	b'	a'	b'	a'	b	a	b	a	b'	$b'a'b'$	b	b	a	b	b'	a'	b'	b	a	b	$\#$
$\#$	aba	b	a	b'	a'	b'	b'	a'	b	a	bb	a	b'	a'	b'	a'	b	aba	b'	b'	a'	b'	bb	a	b	b'	a'	b'	$bab\#$

Remark 1. The PCP is undecidable even if the words w_i and x_i are over the **binary alphabet** $\{a, b\}$. This is because any alphabet Σ can be encoded in the binary alphabet using an injective homomorphism $h\colon \Sigma^* \longrightarrow \{a, b\}^*$. (For example, we can use distinct binary words of length $\lceil \log_2 |\Sigma| \rceil$ to encode the letters of Σ.) Because of the injectivity of h, the instance $h_1, h_2 \colon \Delta^* \longrightarrow \Sigma^*$ is equivalent to the binary instance $h \circ h_1, h \circ h_2 \colon \Delta^* \longrightarrow \{a, b\}^*$.

Remark 1. The PCP is undecidable even if the words w_i and x_i are over the **binary alphabet** $\{a, b\}$. This is because any alphabet Σ can be encoded in the binary alphabet using an injective homomorphism $h : \Sigma^* \longrightarrow \{a, b\}^*$. (For example, we can use distinct binary words of length $\lceil \log_2 |\Sigma| \rceil$ to encode the letters of Σ.) Because of the injectivity of h, the instance $h_1, h_2 : \Delta^* \longrightarrow \Sigma^*$ is equivalent to the binary instance $h \circ h_1, h \circ h_2 : \Delta^* \longrightarrow \{a, b\}^*$.

Remark 2. Our construction converts a semi-Thue system (Σ, R) into a PCP instance containing $2|\Sigma| + |R| + 2$ pairs of words. Over the binary alphabet $|\Sigma| = 2$ this means $|R| + 6$ pairs. There is a smarter reduction that provides only $|R| + 4$ pairs of words.

Since there is a semi-Thue system with $|R| = 3$ rules whose individual word problem is undecidable, we obtain that PCP is undecidable among lists with 7 pairs of words. With a different approach this number can be reduced to 5.

On the other hand, PCP is known to be decidable among instances with 2 pairs of words. It is presently not known whether PCP is decidable or undecidable for 3 or 4 pairs of words.

Undecidable problems about context-free grammars

PCP can be further reduced to questions concerning context-free grammars and languages. Let

$$w_1, w_2, \ldots, w_k$$

be a list of k words over an alphabet Σ. Assume that

$$\{1, 2, \ldots k, \#, \$\} \cap \Sigma = \emptyset.$$

We associate to such a list the language

$$L(w_1, \ldots, w_k) = \{ \ i_1 i_2 \ldots i_n \# w_{i_n}^R w_{i_{n-1}}^R \ldots w_{i_1}^R \$ \mid n \geq 1\}$$

over the alphabet $\Delta = \Sigma \cup \{1, 2, \ldots, k\} \cup \{\#, \$\}$.

Undecidable problems about context-free grammars

PCP can be further reduced to questions concerning context-free grammars and languages. Let

$$w_1, w_2, \ldots, w_k$$

be a list of k words over an alphabet Σ. Assume that

$$\{1, 2, \ldots k, \#, \$\} \cap \Sigma = \emptyset.$$

We associate to such a list the language

$$L(w_1, \ldots, w_k) = \{\ i_1 i_2 \ldots i_n \# w_{i_n}^R w_{i_{n-1}}^R \ldots w_{i_1}^R \$ \mid n \geq 1\}$$

over the alphabet $\Delta = \Sigma \cup \{1, 2, \ldots, k\} \cup \{\#, \$\}$.

Using the homomorphism notation with $h : \{1, 2, \ldots, k\} \longrightarrow \Sigma^*$ that maps $h(i) = w_i$ for all $i \in \{1, 2, \ldots, k\}$, the language $L(w_1, \ldots, w_k)$ consists of all words $u \ \# \ h(u)^R \ \$$ for all non-empty $u \in \{1, 2, \ldots k\}^+$.

Undecidable problems about context-free grammars

PCP can be further reduced to questions concerning context-free grammars and languages. Let

$$w_1, w_2, \ldots, w_k$$

be a list of k words over an alphabet Σ. Assume that

$$\{1, 2, \ldots k, \#, \$\} \cap \Sigma = \emptyset.$$

We associate to such a list the language

$$L(w_1, \ldots, w_k) = \{\ i_1 i_2 \ldots i_n \# w_{i_n}^R w_{i_{n-1}}^R \ldots w_{i_1}^R \$ \mid n \geq 1\}$$

over the alphabet $\Delta = \Sigma \cup \{1, 2, \ldots, k\} \cup \{\#, \$\}$.

Using the homomorphism notation with $h : \{1, 2, \ldots, k\} \longrightarrow \Sigma^*$ that maps $h(i) = w_i$ for all $i \in \{1, 2, \ldots, k\}$, the language $L(w_1, \ldots, w_k)$ consists of all words $u \ \# \ h(u)^R \ \$$ for all non-empty $u \in \{1, 2, \ldots k\}^+$.

Remark. The markers # and \$ have no role in our first applications. We could just as well use a simpler language that contains the words $u \ h(u)^R$ for all non-empty index words u. The markers become relevant later.

$$L(w_1, \ldots, w_k) = \{\ i_1 i_2 \ldots i_n \# w_{i_n}^R w_{i_{n-1}}^R \ldots w_{i_1}^R \$ \mid n \geq 1\}$$

The language $L(w_1, \ldots, w_k)$ is context-free: it is generated by the context-free grammar $G = (\{S, A\}, \Delta, P, S)$ with the productions

$$S \longrightarrow A\$$$

to create the end marker \$,

$$A \longrightarrow 1Aw_1^R \mid 2Aw_2^R \mid \ \ldots \ \mid kAw_k^R$$

to create matching index/word pairs, and

$$A \longrightarrow 1\#w_1^R \mid 2\#w_2^R \mid \ \ldots \ \mid k\#w_k^R$$

to terminate (and to guarantee that at least one index/word pair is used).

Remark. The grammar is actually **linear** and **unambiguous**.

Example. The language $L(ab, b, baa)$ corresponding to the word list ab, a, baa is generated by the productions

$$\begin{array}{lcl} S & \longrightarrow & A\$ \\ A & \longrightarrow & 1Aba \mid 2Ab \mid \mid 3Aaab \\ A & \longrightarrow & 1\#ba \mid 2\#b \mid \mid 3\#aab \end{array}$$

For example, the word $123\#aabbba\$$ is generated (uniquely) as follows:

Example. The language $L(ab, b, baa)$ corresponding to the word list ab, a, baa is generated by the productions

$$\begin{array}{lcl} S & \longrightarrow & A\$ \\ A & \longrightarrow & 1Aba \mid 2Ab \mid \mid 3Aaab \\ A & \longrightarrow & 1\#ba \mid 2\#b \mid \mid 3\#aab \end{array}$$

For example, the word $123\#aabbba\$$ is generated (uniquely) as follows:

$$S \Rightarrow A\$ \Rightarrow 1Aba\$ \Rightarrow 12Abba\$ \Rightarrow 123\#aabbba\$$$

Example. The language $L(ab, b, baa)$ corresponding to the word list ab, a, baa is generated by the productions

$$\begin{array}{lcl} S & \longrightarrow & A\$ \\ A & \longrightarrow & 1Aba \mid 2Ab \mid \mid 3Aaab \\ A & \longrightarrow & 1\#ba \mid 2\#b \mid \mid 3\#aab \end{array}$$

For example, the word $123\#aabbba\$$ is generated (uniquely) as follows:

$$S \Rightarrow A\$ \Rightarrow 1Aba\$ \Rightarrow 12Abba\$ \Rightarrow 123\#aabbba\$$$

Remark. In $L(w_1, \ldots, w_k)$ we have the words before and after the center marker $\#$ in reverse orders only because this admits context-free generation of the language. If the word after the marker is not reversed there would not exist a context-free grammar that generates the language.

Theorem. It is undecidable for given context-free languages L_1 and L_2 whether $L_1 \cap L_2 = \emptyset$.

Proof.

Theorem. It is undecidable for given context-free languages L_1 and L_2 whether $L_1 \cap L_2 = \emptyset$.

Proof. We reduce the Post correspondence problem. Let

$$\begin{array}{lcl} L_1 & : & w_1,\ w_2,\ \dots\ w_k \\ L_2 & : & x_1,\ x_2,\ \dots\ x_k \end{array}$$

be a given instance to PCP. We effectively construct the context-free languages

$$L_1 = L(w_1, \dots w_k)$$
$$L_2 = L(x_1, \dots x_k)$$

corresponding to the two lists.

Theorem. It is undecidable for given context-free languages L_1 and L_2 whether $L_1 \cap L_2 = \emptyset$.

Proof. We reduce the Post correspondence problem. Let

$$\begin{array}{lcl} L_1 & : & w_1,\ w_2,\ \ldots\ w_k \\ L_2 & : & x_1,\ x_2,\ \ldots\ x_k \end{array}$$

be a given instance to PCP. We effectively construct the context-free languages

$$L_1 = L(w_1, \ldots w_k)$$
$$L_2 = L(x_1, \ldots x_k)$$

corresponding to the two lists.

Crucially,

$L_1 \cap L_2 \neq \emptyset$ if and only if the PCP instance has a solution

Indeed, a word

$$i_1 i_2 \ldots i_n \# w^R \$$$

is in both $L(w_1, \ldots w_k)$ and $L(x_1, \ldots x_k)$ if and only if $n \geq 1$ and

$$w_{i_1} w_{i_2} \ldots w_{i_n} = w = x_{i_1} x_{i_2} \ldots x_{i_n}.$$

Theorem. It is undecidable for given context-free languages L_1 and L_2 whether $L_1 \cap L_2 = \emptyset$.

Proof. We reduce the Post correspondence problem. Let

$$\begin{array}{lllll} L_1 : & w_1, & w_2, & \ldots & w_k \\ L_2 : & x_1, & x_2, & \ldots & x_k \end{array}$$

be a given instance to PCP. We effectively construct the context-free languages

$$L_1 = L(w_1, \ldots w_k)$$
$$L_2 = L(x_1, \ldots x_k)$$

corresponding to the two lists.

Crucially,

$L_1 \cap L_2 \neq \emptyset$ if and only if the PCP instance has a solution

Indeed, a word

$$i_1 i_2 \ldots i_n \# w^R \$$$

is in both $L(w_1, \ldots w_k)$ and $L(x_1, \ldots x_k)$ if and only if $n \geq 1$ and

$$w_{i_1} w_{i_2} \ldots w_{i_n} = w = x_{i_1} x_{i_2} \ldots x_{i_n}.$$

So an algorithm A that decides whether $L_1 \cap L_2 = \emptyset$ can be used to solve the PCP, a contradiction.

Remark. Because the grammars for the languages $L(w_1, \ldots, w_n)$ and $L(x_1, \ldots, x_n)$ are **linear** and **unambiguous**, the decision problem is undecidable even when restricted to such instances. So the following problem is undecidable:

Is $L(G_1) \cap L(G_2) = \emptyset$ for given linear, unambiguous context-free grammars G_1 and G_2 ?

Remark. Because the grammars for the languages $L(w_1, \ldots, w_n)$ and $L(x_1, \ldots, x_n)$ are **linear** and **unambiguous**, the decision problem is undecidable even when restricted to such instances. So the following problem is undecidable:

Is $L(G_1) \cap L(G_2) = \emptyset$ for given linear, unambiguous context-free grammars G_1 and G_2 ?

This remark immediately gives us the following theorem.

Theorem. It is undecidable whether a given linear context-free grammar G is unambiguous.

Theorem. It is undecidable whether a given linear context-free grammar G is unambiguous.

Proof.

Theorem. It is undecidable whether a given linear context-free grammar G is unambiguous.

Proof. We reduce the undecidable problem "Is $L(G_1) \cap L(G_2) = \emptyset$ for given linear, unambiguous context-free grammars G_1 and G_2 ?"

Let $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$ be two given linear, unambiguous grammars. Rename the variables so that V_1 and V_2 are disjoint and do not contain variable S. Construct a new linear grammar

$$G = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P, S)$$

where P contains all productions from P_1 and P_2, and the additional initialization productions

$$S \longrightarrow S_1 \mid S_2.$$

(Recall, this is our construction for the union of two context-free languages.)

Theorem. It is undecidable whether a given linear context-free grammar G is unambiguous.

Proof. We reduce the undecidable problem "Is $L(G_1) \cap L(G_2) = \emptyset$ for given linear, unambiguous context-free grammars G_1 and G_2 ?"

Let $G_1 = (V_1, T_1, P_1, S_1)$ and $G_2 = (V_2, T_2, P_2, S_2)$ be two given linear, unambiguous grammars. Rename the variables so that V_1 and V_2 are disjoint and do not contain variable S. Construct a new linear grammar

$$G = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P, S)$$

where P contains all productions from P_1 and P_2, and the additional initialization productions

$$S \longrightarrow S_1 \mid S_2.$$

(Recall, this is our construction for the union of two context-free languages.)

$$L(G_1) \cap L(G_2) = \emptyset \text{ if and only if } G \text{ is unambiguous}$$

- Every $u \in L(G_1) \cap L(G_2)$ has two leftmost derivations in G that start $S \Rightarrow S_1$ and $S \Rightarrow S_2$ and continue with derivations according to G_1 and G_2.
- Conversely, if $L(G_1) \cap L(G_2) = \emptyset$ then every generated word has only one leftmost derivation because G_1 and G_2 are unambiguous.)

Our next undecidability result states that there is no algorithm to determine if a given context-free grammar generates all words over its terminal alphabet Σ:

Given a context-free grammar G, is $L(G) = \Sigma^*$

In this proof we take advantage of the markers # and \$ in the language

$$L(w_1, \ldots, w_k) = \{\, i_1 i_2 \ldots i_n \# w_{i_n}^R w_{i_{n-1}}^R \ldots w_{i_1}^R \$ \mid n \geq 1 \}.$$

With the help of the markers we can namely prove that the **complement** of $L(w_1, \ldots, w_k)$ is also (effectively) context-free.

Lemma. The language $L(w_1, \ldots, w_k)$ is (effectively) recognized by a deterministic PDA without ε-transitions, using the final state acceptance mode. The PDA never empties its stack.

Proof.

Lemma. The language $L(w_1, \ldots, w_k)$ is (effectively) recognized by a deterministic PDA without ε-transitions, using the final state acceptance mode. The PDA never empties its stack.

Proof. We construct such a PDA

$$A = (Q, \Delta, \Gamma, \delta, q_0, Z_0, \{q_F\})$$

where $Q = \{q_0, q_1, q_2, q_F\}$ and $\Gamma = \Sigma \cup \{Z_0\}$. The transitions are

$$\begin{array}{lcll}
\delta(q_0, i, Z_0) & = & \{(q_1, w_i^R Z_0)\} & \text{for all } i \in \{1, 2, \ldots, k\} \\
\delta(q_1, i, Z) & = & \{(q_1, w_i^R Z)\} & \text{for all } Z \in \Gamma \text{ and } i \in \{1, 2, \ldots, k\} \\
\delta(q_1, \#, Z) & = & \{(q_2, Z)\} & \text{for all } Z \in \Gamma \\
\delta(q_2, a, a) & = & \{(q_2, \varepsilon)\} & \text{for all } a \in \Sigma \\
\delta(q_2, \$, Z_0) & = & \{(q_F, Z_0)\} &
\end{array}$$

Idea: A reads in state q_1 indices $i \in \{1, 2, \ldots, k\}$ and pushes, for each index i, the corresponding word w_i^R into the stack. Once the marker $\#$ is encountered the machine changes into state q_2 and starts matching input letters with the symbols in the stack. The word is accepted if and only if the stack contains Z_0 when the last input letter is \$ is being read.

Lemma. The language $L(w_1, \ldots, w_k)$ is (effectively) recognized by a deterministic PDA without ε-transitions, using the final state acceptance mode. The PDA never empties its stack.

Corollary. The complement of the language $L(w_1, \ldots, w_k)$ is (effectively) context-free.

Proof.

Lemma. The language $L(w_1, \ldots, w_k)$ is (effectively) recognized by a deterministic PDA without ε-transitions, using the final state acceptance mode. The PDA never empties its stack.

Corollary. The complement of the language $L(w_1, \ldots, w_k)$ is (effectively) context-free.

Proof. Let $A = (Q, \Delta, \Gamma, \delta, q_0, Z_0, \{q_F\})$ be the PDA constructed in the previous proof. Let us add to A a new state f and non-ε-transitions

$$\delta(q, a, Z) = (f, Z)$$

whenever $\delta(q, a, Z)$ is undefined in A. (Here $a \neq \varepsilon$ is a letter.)

Lemma. The language $L(w_1, \ldots, w_k)$ is (effectively) recognized by a deterministic PDA without ε-transitions, using the final state acceptance mode. The PDA never empties its stack.

Corollary. The complement of the language $L(w_1, \ldots, w_k)$ is (effectively) context-free.

Proof. Let $A = (Q, \Delta, \Gamma, \delta, q_0, Z_0, \{q_F\})$ be the PDA constructed in the previous proof. Let us add to A a new state f and non-ε-transitions

$$\delta(q, a, Z) = (f, Z)$$

whenever $\delta(q, a, Z)$ is undefined in A. (Here $a \neq \varepsilon$ is a letter.)

- The new PDA is still deterministic without ε-transitions,
- the transition function is now complete, and
- the stack becomes never empty because Z_0 is never removed from the stack.

These properties imply that the PDA does not halt until the entire input word has been read. For every input word w there corresponds a **unique state** q and stack content α such that

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$$

Lemma. The language $L(w_1, \ldots, w_k)$ is (effectively) recognized by a deterministic PDA without ε-transitions, using the final state acceptance mode. The PDA never empties its stack.

Corollary. The complement of the language $L(w_1, \ldots, w_k)$ is (effectively) context-free.

Proof. Let $A = (Q, \Delta, \Gamma, \delta, q_0, Z_0, \{q_F\})$ be the PDA constructed in the previous proof. Let us add to A a new state f and non-ε-transitions

$$\delta(q, a, Z) = (f, Z)$$

whenever $\delta(q, a, Z)$ is undefined in A. (Here $a \neq \varepsilon$ is a letter.)

- The new PDA is still deterministic without ε-transitions,
- the transition function is now complete, and
- the stack becomes never empty because Z_0 is never removed from the stack.

These properties imply that the PDA does not halt until the entire input word has been read. For every input word w there corresponds a **unique state** q and stack content α such that

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$$

If we **swap the final and the non-final states** so that the new final states are all states except q_F then the new PDA recognizes the complement of $L(w_1, \ldots, w_k)$.

Remark. In fact, the family of deterministic context-free languages is closed under complementation.

However, possible ε-transitions in a deterministic PDA cause technical difficulties in the proof. (After reading the input the PDA can still continue to other states using ε-transitions. Thus the final state reached at the end is not unique.)

In our case, the complementation is easy because the PDA has no ε-transitions: the PDA reaches a unique state at the end, so it is enough to swap the accepting and non-accepting states.

Theorem. It is undecidable if $L = \Sigma^*$ for a given context-free language $L \subseteq \Sigma^*$.

Proof.

Theorem. It is undecidable if $L = \Sigma^*$ for a given context-free language $L \subseteq \Sigma^*$.

Proof. We reduce the Post correspondence problem. For any given instance

$$\begin{array}{llllll} L_1 & : & w_1, & w_2, & \ldots & w_k \\ L_2 & : & x_1, & x_2, & \ldots & x_k \end{array}$$

of PCP we effectively construct the complements L_1 and L_2 of $L(w_1, \ldots, w_k)$ and $L(x_1, \ldots, x_k)$, respectively.

Then

$$L_1 \cup L_2 = \Sigma^* \iff L(w_1, \ldots, w_k) \cap L(x_1, \ldots, x_k) = \emptyset$$

$$\iff \text{the PCP instance } L_1, L_2 \text{ does not have a solution}$$

We can effectively construct the union $L_1 \cup L_2$, so the result follows from the undecidability of PCP.

Corollary. Let L_1 and L_2 be given context-free languages and let R be a given regular language. The following questions are undecidable:

(a) Is $L_1 = L_2$?

(b) Is $L_2 \subseteq L_1$?

(c) Is $L_1 = R$?

(c) Is $R \subseteq L_1$?

The question whether $L_1 \subseteq R$ is, however, decidable.

Proof.

Mortality of matrix products

Let $\{M_1, M_2, \ldots, M_k\}$ be a finite set of $n \times n$ matrices with integer entries. We say that the matrix set is **mortal** if some product of the matrices from the set is the zero matrix.

Example. Consider

$$M_1 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \qquad M_2 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

Example. Consider

$$M_1 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \qquad M_2 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

The product $M_2M_1M_2$ is the zero matrix so the matrix set $\{M_1, M_2\}$ is mortal.

Example. Consider the following two 2×2 matrices:

$$M_1 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \qquad M_2 = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$$

Example. Consider the following two 2×2 matrices:

$$M_1 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \qquad M_2 = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$$

Matrix set $\{M_1, M_2\}$ is not mortal: Because $\det(M_1)$ and $\det(M_2)$ are non-zero, the determinant of every product is non-zero.

The **matrix mortality problem** asks whether a given finite set of $n \times n$ integer matrices is mortal. In the following we show that this problem is undecidable, even among 3×3 integer matrices.

Let us associate to each word $w = a_1 a_2 \ldots a_m$ over the alphabet $\{1, 2, 3\}$ the integer

$$\sigma(w) = a_m + 4a_{m-1} + \cdots + 4^{m-1} a_1$$

So $\sigma(w) =$ the number that w represents in base four.

The function $\sigma : \{1, 2, 3\}^* \longrightarrow \mathbb{N}$ is **injective** (because digit 0 is not used).

For any words u and v we have

$$\sigma(uv) = \sigma(v) + 4^{|v|} \sigma(u).$$

Let us associate to each word $w = a_1 a_2 \ldots a_m$ over the alphabet $\{1, 2, 3\}$ the integer

$$\sigma(w) = a_m + 4a_{m-1} + \cdots + 4^{m-1} a_1$$

So $\sigma(w)$ = the number that w represents in base four.

The function $\sigma : \{1, 2, 3\}^* \longrightarrow \mathbb{N}$ is **injective** (because digit 0 is not used).

For any words u and v we have

$$\sigma(uv) = \sigma(v) + 4^{|v|}\sigma(u).$$

Associate to each word w the 2×2 integer matrix

$$M_w = \begin{pmatrix} 4^{|w|} & 0 \\ \sigma(w) & 1 \end{pmatrix}$$

If u and v are any two words then

$$M_u M_v = \begin{pmatrix} 4^{|u|} & 0 \\ \sigma(u) & 1 \end{pmatrix} \begin{pmatrix} 4^{|v|} & 0 \\ \sigma(v) & 1 \end{pmatrix} = \begin{pmatrix} 4^{|uv|} & 0 \\ \sigma(uv) & 1 \end{pmatrix} = M_{uv}$$

so that the mapping $w \mapsto M_w$ is a monoid morphism from words to the multiplicative monoid of 2×2 matrices. The morphism is injective because $w \mapsto \sigma(w)$ is injective.

By “fusing” together the matrices M_u and M_v of two words u and v we obtain a single 3×3 integer matrix

$$M_{u,v} = \begin{pmatrix} 4^{|u|} & 0 & 0 \\ 0 & 4^{|v|} & 0 \\ \sigma(u) & \sigma(v) & 1 \end{pmatrix}.$$

corresponding to the pair u, v. It has a similar morphism property:

Lemma. For all $u, v, x, y \in \{1, 2, 3\}^*$ holds that $M_{u,v} M_{x,y} = M_{ux,vy}$.

Proof. A direct calculation (on the blackboard).

We add in the mix the following special 3×3 matrix

$$A = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix}.$$

Lemma. The matrix A has following properties:

- $A^2 = A$ (so A is **idempotent**).
- More generally,
$$AM_{u,v}A = (4^{|u|} + \sigma(u) - \sigma(v))A$$
for all words u, v. (The first bullet point is the special case with $u = v = \varepsilon$.)
- $AM_{u,v}A = 0$ if and only if $v = 1u$.

Proof. A direct calculation (on the blackboard).

Theorem. It is undecidable whether a given finite set of 3×3 integer matrices is mortal.

Proof.

Theorem. It is undecidable whether a given finite set of 3×3 integer matrices is mortal.

Proof. Reduction from the PCP over the two letter alphabet $\Delta = \{2, 3\}$.

Recall the properties

- $M_{u,v}M_{x,y} = M_{ux,vy}$
- $A^2 = A$
- $AM_{u,v}A = 0$ if and only if $v = 1u$

Theorem. It is undecidable whether a given finite set of 3×3 integer matrices is mortal.

Proof. Reduction from the PCP over the two letter alphabet $\Delta = \{2, 3\}$.

Recall the properties

- $M_{u,v} M_{x,y} = M_{ux,vy}$
- $A^2 = A$
- $A M_{u,v} A = 0$ if and only if $v = 1u$

For a given PCP instance

$$\begin{array}{lcllll} L_1 & : & w_1, & w_2, & \ldots & w_k \\ L_2 & : & x_1, & x_2, & \ldots & x_k \end{array}$$

over $\Delta = \{2, 3\}$ we construct a set of $2k+1$ integer matrices of size 3×3:

$$A \quad \text{and} \quad M_i = M_{w_i, x_i} \quad \text{and} \quad M'_i = M_{w_i, 1x_i} \quad \text{for } i \in \{1, \ldots, k\}$$

Theorem. It is undecidable whether a given finite set of 3×3 integer matrices is mortal.

Proof. Reduction from the PCP over the two letter alphabet $\Delta = \{2, 3\}$.

Recall the properties

- $M_{u,v} M_{x,y} = M_{ux,vy}$
- $A^2 = A$
- $A M_{u,v} A = 0$ if and only if $v = 1u$

For a given PCP instance

$$\begin{array}{ll} L_1 : & w_1, \ w_2, \ \ldots \ w_k \\ L_2 : & x_1, \ x_2, \ \ldots \ x_k \end{array}$$

over $\Delta = \{2, 3\}$ we construct a set of $2k + 1$ integer matrices of size 3×3:

$$A \quad \text{and} \quad M_i = M_{w_i, x_i} \quad \text{and} \quad M_i' = M_{w_i, 1x_i} \quad \text{for } i \in \{1, \ldots, k\}$$

It is enough to show (on the blackboard) that

the matrix set is mortal $\Longleftrightarrow$ the PCP instance has a solution

Remark. The proof converts a PCP instance of size k into a set of $2k+1$ matrices. Since the PCP is undecidable among $k=5$ pairs of words, we see that the mortality problem is undecidable among sets of 11 integer matrices of size 3×3.

A more careful analysis of the proof yields the undecidability of the mortality problem for sets of $k+1=6$ matrices of size 3×3.

Remark. The proof converts a PCP instance of size k into a set of $2k+1$ matrices. Since the PCP is undecidable among $k=5$ pairs of words, we see that the mortality problem is undecidable among sets of 11 integer matrices of size 3×3.

A more careful analysis of the proof yields the undecidability of the mortality problem for sets of $k+1=6$ matrices of size 3×3.

To see this: The PCP instance with k pairs of words has a solution iff for some i the following set of $k+1$ integer matrices of size 3×3 is mortal:

$$\{AM'_i, M_1, M_2, \ldots, M_k\}$$

Indeed:

- if this set is mortal then the set constructed in the proof is also mortal and so the PCP instance has a solution.
- Conversely, if the PCP instance has a solution $i_1 \ldots i_m$ then

$$AM'_{i_1}M_{i_2}\ldots M_{i_m}AM'_{i_1} = 0\cdot M'_{i_1} = 0$$

Remark. The proof converts a PCP instance of size k into a set of $2k+1$ matrices. Since the PCP is undecidable among $k=5$ pairs of words, we see that the mortality problem is undecidable among sets of 11 integer matrices of size 3×3.

A more careful analysis of the proof yields the undecidability of the mortality problem for sets of $k+1=6$ matrices of size 3×3.

To see this: The PCP instance with k pairs of words has a solution iff for some i the following set of $k+1$ integer matrices of size 3×3 is mortal:

$$\{AM'_i, M_1, M_2, \ldots, M_k\}$$

Indeed:

- if this set is mortal then the set constructed in the proof is also mortal and so the PCP instance has a solution.
- Conversely, if the PCP instance has a solution $i_1 \ldots i_m$ then

$$AM'_{i_1} M_{i_2} \ldots M_{i_m} AM'_{i_1} = 0 \cdot M'_{i_1} = 0$$

An algorithm to solve the PCP instance thus constructs these sets for all $i \in \{1, \ldots, k\}$ and asks a hypothetical algorithm for mortality whether any of these sets is mortal.

Remark. Mortality is undecidable among sets of two 15×15 matrices, among sets of three 9×9 matrices and among four 5×5 matrices.

It is not known whether the mortality problem is decidable among sets of 2×2 matrices. For a set of two 2×2 matrices the problem is decidable.

Decidability of a related **Skolem-Pisot problem** is open:

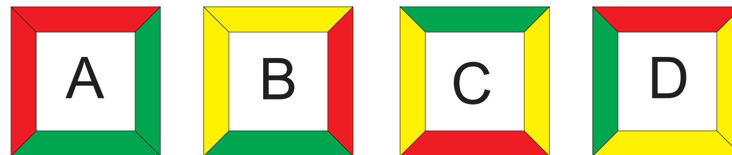
Given a single $n \times n$ integer matrix M, does there exists $k \geq 1$ such that the element in the upper right corner of M^k is zero ?

This is known to be decidable for matrices of size $n \leq 5$, but the decidability status is not known for larger values of n.

Tiling problems

Wang tiles are unit square tiles with colored edges. Each tile can be represented as a 4-tuple (N, E, S, W) where N, E, S and W are the colors of the north, east, south and west sides of the square.

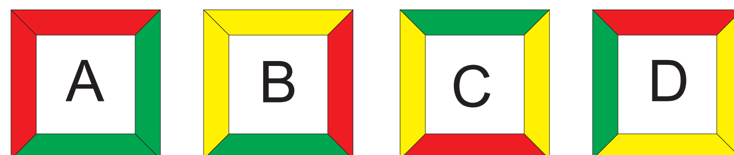
A **Wang tile set** is a **finite** set of Wang tiles.



Tiling problems

Wang tiles are unit square tiles with colored edges. Each tile can be represented as a 4-tuple (N, E, S, W) where N, E, S and W are the colors of the north, east, south and west sides of the square.

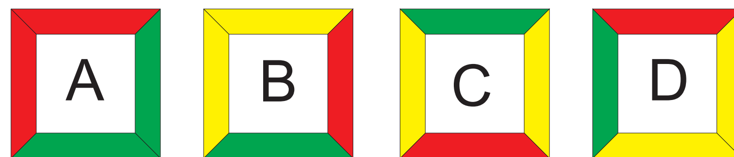
A **Wang tile set** is a **finite** set of Wang tiles.



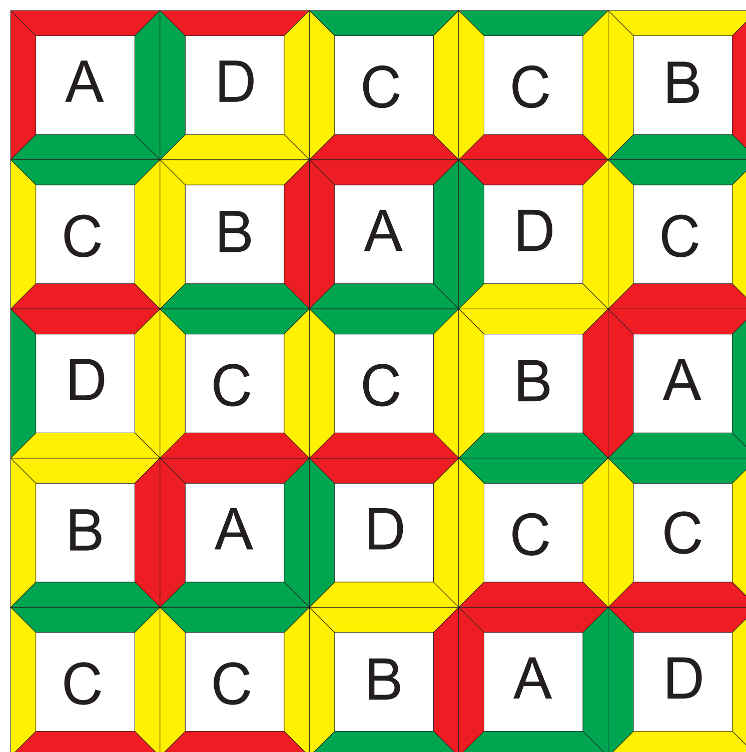
A **tiling** by a Wang tile set T is an assignment $t : \mathbb{Z}^2 \longrightarrow T$ of copies of the tiles in T on the plane in such a way that the adjacent edges of neighboring tiles have the same color.

Note that the tiles may not be rotated.

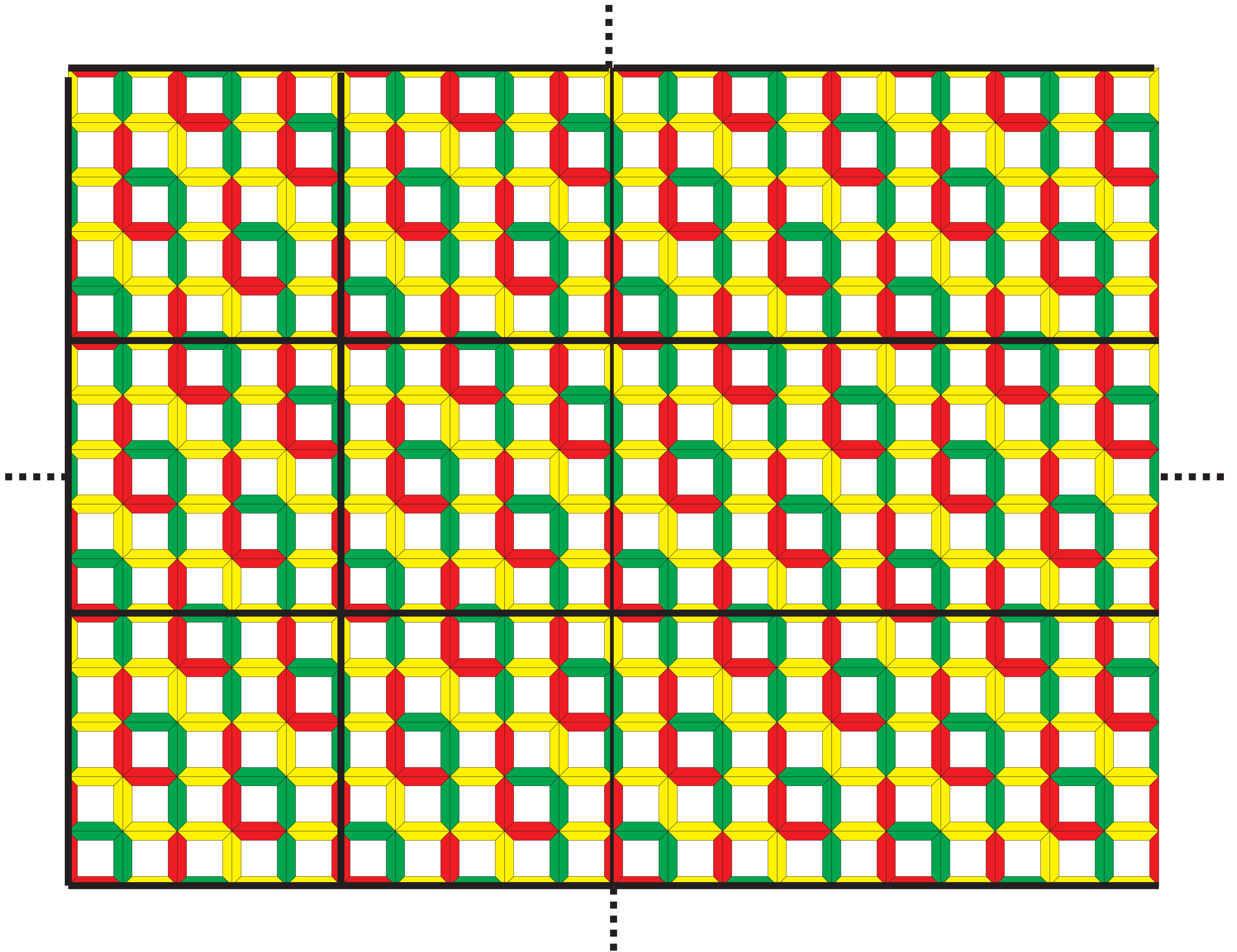
Example. With



we can tile a 5×5 square...



... and since the colors on the borders match this square can be repeated to form a valid periodic tiling of the whole plane.



The **tiling problem** (also known as the **domino problem**) asks whether a given finite set of tiles admits at least one valid tiling. This question turns out to be undecidable.

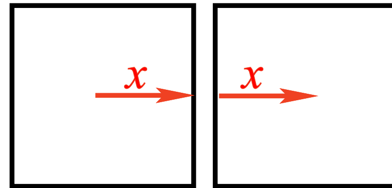
The **tiling problem** (also known as the **domino problem**) asks whether a given finite set of tiles admits at least one valid tiling. This question turns out to be undecidable.

Here we, however, only prove the undecidability of the following variant, called the **tiling problem with a seed tile**:

Given a finite set T of Wang tiles and a seed tile $s \in T$, does there exist a valid tiling of the plane that contains at least one copy of the seed s ?

To prove undecidability we associate to any given Turing machine M a set T_M of Wang tiles such that valid tilings "draw" computations according to M. Horizontal rows represent consecutive ID's, time increasing upwards.

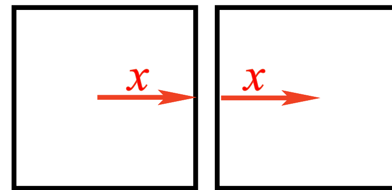
Colors will be represented as **labeled arrows**. The matching rule is that each arrow head must meet an arrow tail with the same label in the neighboring tile.



(Such presentation can easily be converted into colors by identifying each arrow direction/label -pair by a unique color.)

To prove undecidability we associate to any given Turing machine M a set T_M of Wang tiles such that valid tilings "draw" computations according to M. Horizontal rows represent consecutive ID's, time increasing upwards.

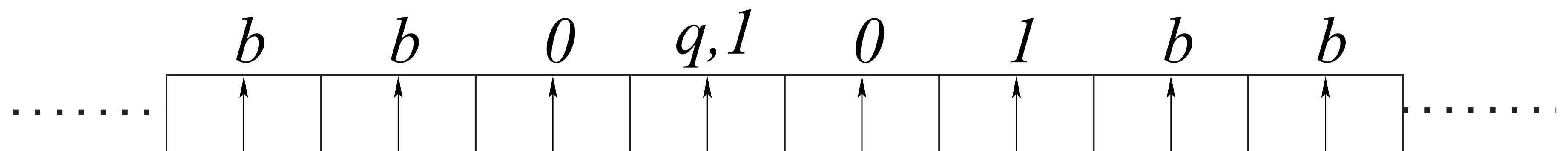
Colors will be represented as **labeled arrows**. The matching rule is that each arrow head must meet an arrow tail with the same label in the neighboring tile.



(Such presentation can easily be converted into colors by identifying each arrow direction/label -pair by a unique color.)

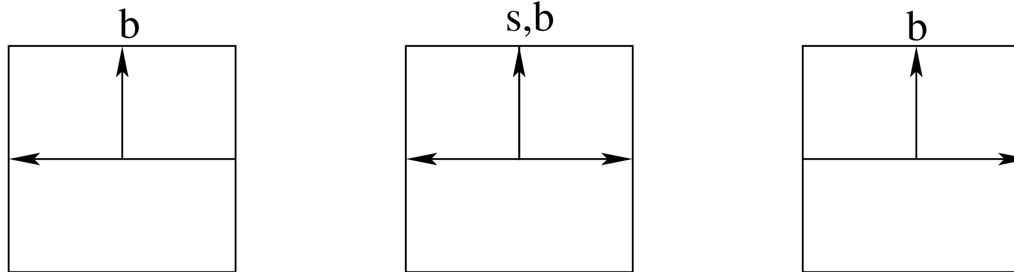
The labels are

- tape symbols (representing a tape cell containing that symbol), or
- state/tape symbol pairs (representing a tape cell containing the control unit at the given state).



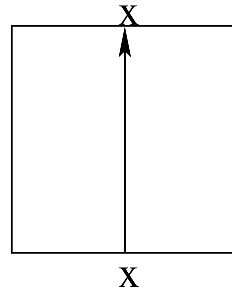
Let $M = (Q, \Gamma, \Sigma, \delta, s, b, f)$. In T_M we have the following tiles:

(i) three starting tiles to represent the blank tape



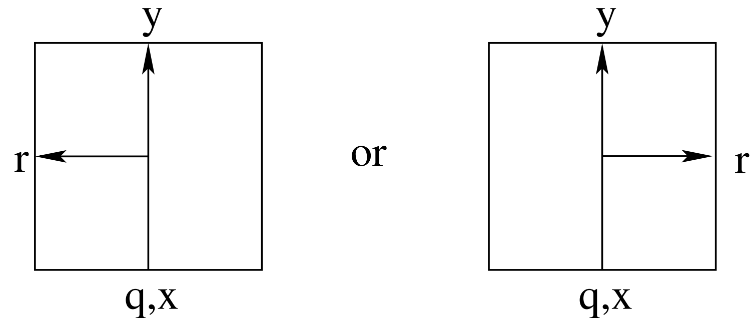
Let $M = (Q, \Gamma, \Sigma, \delta, s, b, f)$. In T_M we have the following tiles:

(ii) for every tape letter $x \in \Gamma$ an alphabet tile



Let $M = (Q, \Gamma, \Sigma, \delta, s, b, f)$. In T_M we have the following tiles:

(iii) for every state $q \in Q \setminus \{f\}$ and and tape symbol $x \in \Gamma$ such that $\delta(q, x)$ is defined one action tile

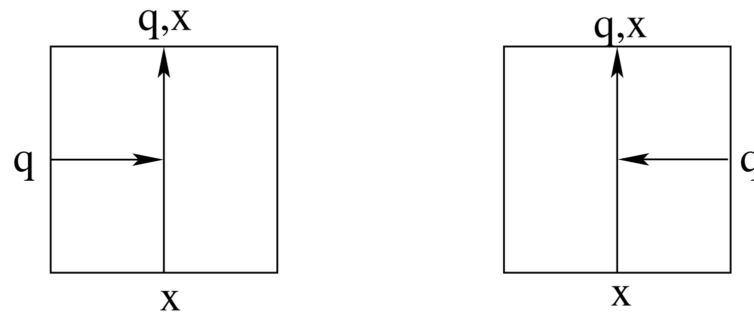


where

- the left tile is used if $\delta(q, x) = (r, y, L)$, and
- the right tile iff $\delta(q, x) = (r, y, R)$.

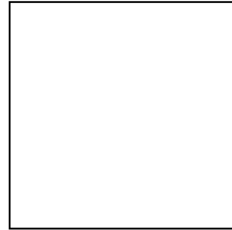
Let $M = (Q, \Gamma, \Sigma, \delta, s, b, f)$. In T_M we have the following tiles:

(iv) for every non-final state $q \in Q \setminus \{f\}$ and tape symbol $x \in \Gamma$ the two merging tiles



Let $M = (Q, \Gamma, \Sigma, \delta, s, b, f)$. In T_M we have the following tiles:

(v) the blank tile

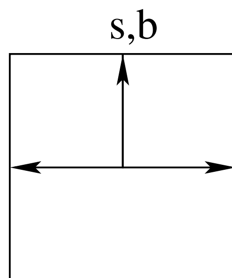


Theorem. The tiling problem with a seed tile is undecidable.

Proof.

Theorem. The tiling problem with a seed tile is undecidable.

Proof. We reduce the undecidable problem of deciding if a given TM halts from the blank initial tape. For any given $M = (Q, \Gamma, \Sigma, \delta, s, b, f)$ we effectively construct the tile set T_M and choose the initialization tile

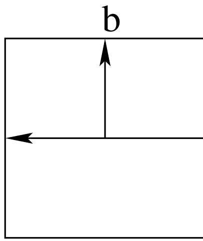


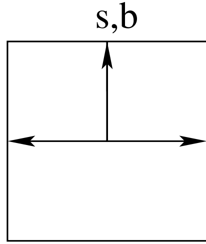
as the seed tile.

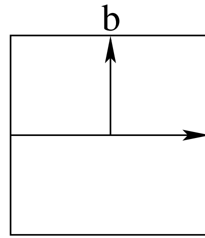
It is now enough to prove that

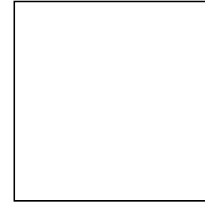
T_M admits a tiling containing the seed $\iff$ M does not halt from the blank tape

$$M = (Q, \Gamma, \Sigma, \delta, s, b, f)$$

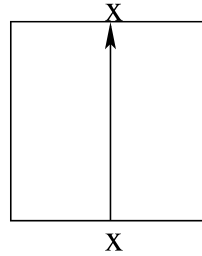




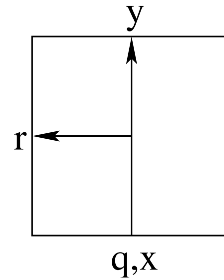




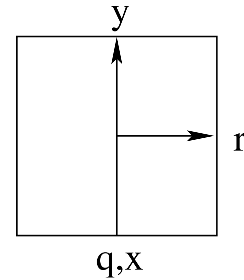
$\forall x \in \Gamma$:



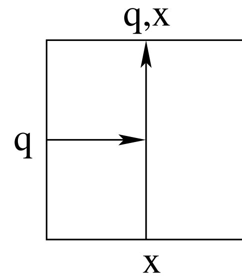
For all q, x, if $\delta(q, x)$ is defined:

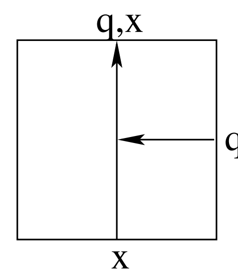


or



$\forall x \in \Gamma, \forall q \in Q \setminus \{f\}$:





Remark. Proving the undecidability without the seed tile constraint is significantly harder. It requires an **aperiodic** Wang tile set: A tile set that admits a tiling but does not admit any periodic tiling.

- Admitting a periodic tiling is semi-decidable: Guess positive integers n and m and a tiling of an $n \times m$ rectangle such that opposite sides of the rectangle contain identical sequences of colors.
- Not admitting a tiling is semi-decidable: Guess number n and verify that the tiles cannot tile the $n \times n$ square without a tiling error. (A compactness argument shows that a tile set that tiles all arbitrarily large squares also admits a tiling of the infinite plane.)

So if there did not exist any aperiodic tile set then both negative and positive instances of the tiling problem would be semi-decidable $\Longrightarrow$ decidability

Undecidability and incompleteness in arithmetics

Next we prove that it is undecidable whether a given first-order sentence concerning natural numbers is true.

Let us briefly recall what first-order sentences are in this setting (no formal definition is given since we all are familiar with the concepts).

Undecidability and incompleteness in arithmetics

Next we prove that it is undecidable whether a given first-order sentence concerning natural numbers is true.

Let us briefly recall what first-order sentences are in this setting (no formal definition is given since we all are familiar with the concepts).

The symbols we use in the sentences are

- **constants** $0, 1, 2, \ldots$
- **variables** that get values in the domain $\mathbb{N} = \{0, 1, 2, \ldots\}$,
- **functions** $+$ (addition) and $\cdot$ (multiplication), both binary and in the infix notation,
- **predicates** $<$, $\leq$, $>$, $\geq$, $=$, $\neq$, all binary and in the infix notation, all with the usual meaning,
- **connectives** $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalence),
- **quantifiers** $\forall x$ (for all $x \in \mathbb{N}$) and $\exists x$ (for some $x \in \mathbb{N}$).

Also **parentheses** are used to indicate order of constructions.

A **term** is build from constant and variables using functions $+$ and $\cdot$ any number of times. A term itself is a function that obtains a value in $\mathbb{N}$ when its variables are assigned values in $\mathbb{N}$.

When evaluating a term we use the usual precedence of multiplication before addition. Also the multiplication sign does not need to be written explicitly.

Example. The term

$$2x + 11$$

has one variable x. It obtains the value 15 when x is given the value 2. The term

$$(x + y)(x + 2) + y$$

has two variables x, y. When given values $x = 2$ and $y = 3$ the term gets the value 23.

An **atomic formula** is a predicate applied to two terms. It obtains a **truth value** when the variables are assigned values in $\mathbb{N}$.

Example. The atomic formula

$$(x+y)(x+2)+y \quad \geq \quad 2x+11$$

is true when $x=2$ and $y=3$, but it is false when $x=2$ and $y=1$.

A **formula** is build from atomic formulas using any number of connectives and quantifiers. Occurrences of variables that are not quantified are **free**. Quantified occurrences are **bound**. A formula obtains a truth value when the variables with free occurrences are assigned values in $\mathbb{N}$.

We employ the following commonly used precedence rules for the connectives: the evaluation is done in the order

1. $\neg$ 2. $\wedge$ and $\vee$ 3. $\forall$ and $\exists$ 4. $\Rightarrow$ and $\Leftrightarrow$.

A formula without any free variables is a **sentence**. A sentence has a well defined truth value (independent of any variable assignments).

Example. The formula

$$(z > 1) \land \forall x \forall y \, ((x > 1 \land y > 1) \Rightarrow xy \neq z)$$

has one free variable z. The formula is true when $z = 3$ but false when $z = 4$. In fact, the formula is true if and only if z is a prime number.

Example. The formula

$$(z > 1) \wedge \forall x \forall y \; ((x > 1 \wedge y > 1) \Rightarrow xy \neq z)$$

has one free variable z. The formula is true when $z = 3$ but false when $z = 4$. In fact, the formula is true if and only if z is a prime number.

Adding the $\forall m \exists z \; (z > m) \wedge \ldots$ fragment in front of the formula above we obtain the formula

$$\forall m \exists z \; ((z > m) \wedge \forall x \forall y \; ((x > 1 \wedge y > 1) \Rightarrow xy \neq z))$$

This formula has no free variables so it is a sentence. The sentence is true: it states the fact that there are arbitrarily large prime numbers, *i.e.*, infinitely many primes.

As a shorthand notation we may introduce names to formulas, together with a parameter list of free variables.

Example. Let us name the formula of primality of z as $\mathrm{Prime}(z)$:

$$\mathrm{Prime}(z) \quad := \quad (z > 1) \wedge \forall x \forall y \; ((x > 1 \wedge y > 1) \Rightarrow xy \neq z)$$

Then $\mathrm{Prime}(2z+1)$ denotes the formula obtained by replacing each free occurrence of z by $2z+1$. So $\mathrm{Prime}(2z+1)$ is true iff $2z+1$ is a prime number.

Now the formula

$$\forall m \; \exists z \; ((z > m) \wedge \mathrm{Prime}(z) \wedge \mathrm{Prime}(z+2))$$

is a sentence stating the famous twin prime conjecture. It has a well defined truth value true or false, but it is not known which one.

Remark. Our sentences are in the **first-order** logic, meaning that quantifications are over variables that represent individual numbers. In the second-order logic variables would be allowed to represent predicates (in particular, sets of numbers).

Hilbert's dream: It would great if there were an algorithm that would determine for any given first-order sentence whether the sentence is true or false.

Unfortunately the problem is undecidable:

Theorem. It is undecidable whether a given first-order sentence over $\mathbb{N}$ is true.

Proof.

Hilbert's dream: It would great if there were an algorithm that would determine for any given first-order sentence whether the sentence is true or false.

Unfortunately the problem is undecidable:

Theorem. It is undecidable whether a given first-order sentence over $\mathbb{N}$ is true.

Proof. We reduce the Post correspondence problem: Given any PCP instance

$$\begin{array}{lllll} L_1 & : w_1, & w_2, & \ldots & w_k \\ L_2 & : x_1, & x_2, & \ldots & x_k \end{array}$$

over the binary alphabet $\Sigma = \{2, 3\}$, we effectively construct a first-order sentence that is true if and only if the PCP instance has a solution.

We use the same notations and ideas as in the matrix mortality section. In particular, recall that $\sigma(w)$ is the number that $w \in \Sigma^*$ represents in base 4.

Let us denote, for all $i = 1, \ldots, k$,

$$\begin{aligned} A_i &= \sigma(w_i) \\ B_i &= 4^{|w_i|} \\ C_i &= \sigma(x_i) \\ D_i &= 4^{|x_i|} \end{aligned}$$

These are natural number constants that can be effectively computed from the given PCP instance. Notice that these are elements in the 3×3 matrices

$$M_i = M_{w_i,x_i} = \begin{pmatrix} B_i & 0 & 0 \\ 0 & D_i & 0 \\ A_i & C_i & 1 \end{pmatrix}$$

that we constructed for the matrix mortality problem.

Let $i_1 i_1 \dots i_m$ be a sequence of indices, $i_j \in \{1, 2, \dots, k\}$. For every $j = 0, 1, \dots, m$ we define the natural numbers

$$\begin{aligned} a_j &= \sigma(w_{i_1} \dots w_{i_j}) \\ b_j &= 4^{|w_{i_1} \dots w_{i_j}|} \\ c_j &= \sigma(x_{i_1} \dots x_{i_j}) \\ d_j &= 4^{|x_{i_1} \dots x_{i_j}|} \end{aligned}$$

In the matrix notation these are elements of the product matrices

$$M_{i_1} M_{i_1} \dots M_{i_j} = \begin{pmatrix} b_j & 0 & 0 \\ 0 & d_j & 0 \\ a_j & c_j & 1 \end{pmatrix}.$$

Because

$$\begin{pmatrix} b_j & 0 & 0 \\ 0 & d_j & 0 \\ a_j & c_j & 1 \end{pmatrix} = \begin{pmatrix} b_{j-1} & 0 & 0 \\ 0 & d_{j-1} & 0 \\ a_{j-1} & c_{j-1} & 1 \end{pmatrix} \begin{pmatrix} B_{i_j} & 0 & 0 \\ 0 & D_{i_j} & 0 \\ A_{i_j} & C_{i_j} & 1 \end{pmatrix}$$

$$= \begin{pmatrix} B_{i_j} b_{j-1} & 0 & 0 \\ 0 & D_{i_j} d_{j-1} & 0 \\ B_{i_j} a_{j-1} + A_{i_j} & D_{i_j} c_{j-1} + C_{i_j} & 1 \end{pmatrix}$$

the numbers a_j, b_j, c_j, d_j are determined by the equations

$$a_0 = c_0 = 0 \text{ and } b_0 = d_0 = 1, \tag{Start}$$

$$\forall j = 1, \ldots m \tag{Follow}$$

$$\begin{aligned} a_j &= B_{i_j} a_{j-1} + A_{i_j}, \\ b_j &= B_{i_j} b_{j-1}, \\ c_j &= D_{i_j} c_{j-1} + C_{i_j}, \\ d_j &= D_{i_j} d_{j-1}. \end{aligned}$$

Because σ is injective, the sequence $i_1 i_2 \ldots i_m$ is a solution to the PCP if and only if

$$a_m = c_m \tag{End}$$

$$a_0 = c_0 = 0 \text{ and } b_0 = d_0 = 1, \tag{Start}$$

$$\forall j = 1, \ldots m \tag{Follow}$$

$$\begin{aligned} a_j &= B_{i_j} a_{j-1} + A_{i_j}, \\ b_j &= B_{i_j} b_{j-1}, \\ c_j &= D_{i_j} c_{j-1} + C_{i_j}, \\ d_j &= D_{i_j} d_{j-1}. \end{aligned}$$

$$a_m = c_m \tag{End}$$

We see that the PCP instance has a solution if and only if numbers a_j, b_j, c_j and d_j satisfying the conditions (First), (Follow) and (End) exist for some choice of $m \geq 1$ and $i_1 i_2 \ldots i_m$.

Writing this as a single sentence: the PCP instance has a solution if and only if the "sentence"

$$\begin{aligned}&\exists m\ (m \geq 1) \wedge \exists a_0, \ldots, \exists a_m \exists b_0, \ldots, \exists b_m \exists c_0, \ldots, \exists c_m \exists d_0, \ldots, \exists d_m\\ &\quad \text{Start}(a_0, b_0, c_0, d_0) \wedge\\ &\qquad \forall j\ \ [(j < m) \Rightarrow \text{Follow}(a_j, b_j, c_j, d_j, a_{j+1}, b_{j+1}, c_{j+1}, d_{j+1})] \wedge\\ &\qquad\quad \text{End}(a_m, b_m, c_m, d_m)\end{aligned}$$

is true, where we use the shorthand notations

- $\text{Start}(a,b,c,d)$ for $(a = 0) \wedge (b = 1) \wedge (c = 0) \wedge (d = 1)$
- $\text{Follow}(a,b,c,d,a',b',c',d')$ for

$$\begin{aligned}&[(a' = B_1 a + A_1) \wedge (b' = B_1 b) \wedge (c' = D_1 c + C_1) \wedge (d' = D_1 d)] \quad \vee\\ &\quad [(a' = B_2 a + A_2) \wedge (b' = B_2 b) \wedge (c' = D_2 c + C_2) \wedge (d' = D_2 d)] \quad \vee\\ &\qquad \cdots\\ &\qquad\quad [(a' = B_k a + A_k) \wedge (b' = B_k b) \wedge (c' = D_k c + C_k) \wedge (d' = D_k d)]\end{aligned}$$

- $\text{End}(a,b,c,d)$ for $a = c$

Writing this as a single sentence: the PCP instance has a solution if and only if the "sentence"

$$\begin{aligned}
&\exists m\ (m \geq 1) \wedge \exists a_0, \ldots, \exists a_m \exists b_0, \ldots, \exists b_m \exists c_0, \ldots, \exists c_m \exists d_0, \ldots, \exists d_m \\
&\quad \text{Start}(a_0, b_0, c_0, d_0) \wedge \\
&\quad\quad \forall j\ \ [(j < m) \Rightarrow \text{Follow}(a_j, b_j, c_j, d_j, a_{j+1}, b_{j+1}, c_{j+1}, d_{j+1})] \wedge \\
&\quad\quad\quad \text{End}(a_m, b_m, c_m, d_m)
\end{aligned}$$

is true, where we use the shorthand notations

- Start(a,b,c,d) for $(a = 0) \wedge (b = 1) \wedge (c = 0) \wedge (d = 1)$
- Follow(a,b,c,d,a',b',c',d') for

$$\begin{aligned}
&[(a' = B_1 a + A_1) \wedge (b' = B_1 b) \wedge (c' = D_1 c + C_1) \wedge (d' = D_1 d)] \quad \vee \\
&\quad [(a' = B_2 a + A_2) \wedge (b' = B_2 b) \wedge (c' = D_2 c + C_2) \wedge (d' = D_2 d)] \quad \vee \\
&\quad\quad \ldots \\
&\quad\quad\quad [(a' = B_k a + A_k) \wedge (b' = B_k b) \wedge (c' = D_k c + C_k) \wedge (d' = D_k d)]
\end{aligned}$$

- End(a,b,c,d) for $a = c$

Problem: this is not a correctly formed first-order formula because the number of quantified variables a_i, b_i, c_i and d_i is not constant but depends on the value of variable m (representing the length of the PCP solution).

Solution: an arbitrary finite sequence $n_0, n_1, \ldots, n_m$ of natural numbers can be encoded as a single number.

Actually, we get a simpler formula if we encode each finite sequence as a pair of two numbers. The encoding is given by the **Gödel's β-function**: For any $a, b, i \in \mathbb{N}$ we define

$$\beta(a, b, i) = a \bmod [1 + (i + 1)b]$$

where $x \bmod y$ denotes the smallest natural number that is congruent to x modulo y.

Interpretation: The value $\beta(a, b, i)$ is the i'th element of the sequence encoded by a and b.

The β-function is implemented by a simple first-order formula: $n = \beta(a, b, i)$ if and only if

$$\mathrm{Beta}(a, b, i, n) \quad := \quad (n < 1 + (i + 1)b) \wedge [\exists x \; a = n + x(1 + (i + 1)b)]$$

is true.

$$\beta(a, b, i) = a \bmod [1 + (i+1)b]$$

The β-function can reproduce sequences of numbers from fixed a and b by varying i. Any desired sequence can be obtained:

Lemma. Let $m \geq 1$, and let $n_0, n_1, \ldots n_m$ be arbitrary natural numbers. Then there exist $a, b \in \mathbb{N}$ such that $n_i = \beta(a, b, i)$ for all $i = 0, 1, \ldots, m$.

Proof of the lemma.

$$\beta(a, b, i) = a \bmod [1 + (i + 1)b]$$

The β-function can reproduce sequences of numbers from fixed a and b by varying i. Any desired sequence can be obtained:

Lemma. Let $m \geq 1$, and let $n_0, n_1, \ldots n_m$ be arbitrary natural numbers. Then there exist $a, b \in \mathbb{N}$ such that $n_i = \beta(a, b, i)$ for all $i = 0, 1, \ldots, m$.

Proof of the lemma.

1) Let us choose

$$b = m! \cdot \max\{n_0, n_1, \ldots, n_m\}.$$

Then the numbers $1 + (i + 1)b$ with $0 \leq i \leq m$ are pairwise co-prime: Let p be a prime number that divides both $1 + (i + 1)b$ and $1 + (j + 1)b$ where $0 \leq i < j \leq m$. Then p also divides the difference

$$(1 + (j + 1)b) - (1 + (i + 1)b) = (j - i)b.$$

Because $j - i$ divides b we see that p necessarily divides b. As p also divides $1 + (i + 1)b$, we have that p divides 1, a contradiction.

$$\beta(a, b, i) = a \bmod [1 + (i + 1)b]$$

The β-function can reproduce sequences of numbers from fixed a and b by varying i. Any desired sequence can be obtained:

Lemma. Let $m \geq 1$, and let $n_0, n_1, \ldots n_m$ be arbitrary natural numbers. Then there exist $a, b \in \mathbb{N}$ such that $n_i = \beta(a, b, i)$ for all $i = 0, 1, \ldots, m$.

Proof of the lemma.

2) So we have a number $b = m! \cdot \max\{n_0, n_1, \ldots, n_m\}$ such that $1 + (i + 1)b$ with $0 \leq i \leq m$ are pairwise co-prime.

The Chinese remainder theorem provides a natural number a such that

$$a \equiv n_i \pmod{1 + (i + 1)b}$$

for all $0 \leq i \leq m$. This is our choice of a. Because

$$n_i \leq b < 1 + (i + 1)b,$$

we have that $n_i = \beta(a, b, i)$.

$$\exists m \ (m \geq 1) \wedge \exists \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2$$

$$\text{Beta}(\alpha_1, \alpha_2, 0, 0) \wedge \text{Beta}(\beta_1, \beta_2, 0, 1) \wedge \text{Beta}(\gamma_1, \gamma_2, 0, 0) \wedge \text{Beta}(\delta_1, \delta_2, 0, 1) \wedge$$

$$\begin{aligned} \forall j \ [(j < m) \Rightarrow \exists a, b, c, d, a', b', c', d' \ \ & \text{Follow}(a, b, c, d, a', b', c', d') \wedge \\ & \text{Beta}(\alpha_1, \alpha_2, j, a) \wedge \text{Beta}(\alpha_1, \alpha_2, j+1, a') \wedge \\ & \text{Beta}(\beta_1, \beta_2, j, b) \wedge \text{Beta}(\beta_1, \beta_2, j+1, b') \wedge \\ & \text{Beta}(\gamma_1, \gamma_2, j, c) \wedge \text{Beta}(\gamma_1, \gamma_2, j+1, c') \wedge \\ & \text{Beta}(\delta_1, \delta_2, j, d) \wedge \text{Beta}(\delta_1, \delta_2, j+1, d')] \wedge \end{aligned}$$

$$\exists r \ \text{Beta}(\alpha_1, \alpha_2, m, r) \wedge \text{Beta}(\gamma_1, \gamma_2, m, r)$$

The sentence above is true if and only if the PCP instance has a solution.

$$\exists m\ (m \geq 1) \land \exists \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2$$

$$\text{Beta}(\alpha_1, \alpha_2, 0, 0) \land \text{Beta}(\beta_1, \beta_2, 0, 1) \land \text{Beta}(\gamma_1, \gamma_2, 0, 0) \land \text{Beta}(\delta_1, \delta_2, 0, 1) \land$$

$$\begin{aligned}\forall j\ [(j < m) \Rightarrow \exists a, b, c, d, a', b', c', d'\ \ & \text{Follow}(a, b, c, d, a', b', c', d') \land \\ & \text{Beta}(\alpha_1, \alpha_2, j, a) \land \text{Beta}(\alpha_1, \alpha_2, j+1, a') \land \\ & \text{Beta}(\beta_1, \beta_2, j, b) \land \text{Beta}(\beta_1, \beta_2, j+1, b') \land \\ & \text{Beta}(\gamma_1, \gamma_2, j, c) \land \text{Beta}(\gamma_1, \gamma_2, j+1, c') \land \\ & \text{Beta}(\delta_1, \delta_2, j, d) \land \text{Beta}(\delta_1, \delta_2, j+1, d')] \land\end{aligned}$$

$$\exists r\ \text{Beta}(\alpha_1, \alpha_2, m, r) \land \text{Beta}(\gamma_1, \gamma_2, m, r)$$

The sentence uses the shorthand notations Follow and Beta defined before:

- $\text{Follow}(a,b,c,d,a',b',c',d')$ stands for

$$\begin{aligned}&[(a' = B_1 a + A_1) \land (b' = B_1 b) \land (c' = D_1 c + C_1) \land (d' = D_1 d)] \quad \lor \\ &[(a' = B_2 a + A_2) \land (b' = B_2 b) \land (c' = D_2 c + C_2) \land (d' = D_2 d)] \quad \lor \\ &\dots \\ &[(a' = B_k a + A_k) \land (b' = B_k b) \land (c' = D_k c + C_k) \land (d' = D_k d)]\end{aligned}$$

- $\text{Beta}(a, b, i, n)$ stands for

$$(n < 1 + (i+1)b) \land [\exists x\ a = n + x(1 + (i+1)b)]$$

$$\exists m\ (m \geq 1) \wedge \exists \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2$$

$$\text{Beta}(\alpha_1, \alpha_2, 0, 0) \wedge \text{Beta}(\beta_1, \beta_2, 0, 1) \wedge \text{Beta}(\gamma_1, \gamma_2, 0, 0) \wedge \text{Beta}(\delta_1, \delta_2, 0, 1) \wedge$$

$$\begin{aligned} \forall j\ [(j < m) \Rightarrow \exists a, b, c, d, a', b', c', d'\ \ &\text{Follow}(a, b, c, d, a', b', c', d') \wedge \\ &\text{Beta}(\alpha_1, \alpha_2, j, a) \wedge \text{Beta}(\alpha_1, \alpha_2, j+1, a') \wedge \\ &\text{Beta}(\beta_1, \beta_2, j, b) \wedge \text{Beta}(\beta_1, \beta_2, j+1, b') \wedge \\ &\text{Beta}(\gamma_1, \gamma_2, j, c) \wedge \text{Beta}(\gamma_1, \gamma_2, j+1, c') \wedge \\ &\text{Beta}(\delta_1, \delta_2, j, d) \wedge \text{Beta}(\delta_1, \delta_2, j+1, d')] \wedge \end{aligned}$$

$$\exists r\ \text{Beta}(\alpha_1, \alpha_2, m, r) \wedge \text{Beta}(\gamma_1, \gamma_2, m, r)$$

Numbers α_1 and α_2 provide the sequence $a_0, a_1, \ldots, a_m$.

Numbers β_1 and β_2 provide the sequence $b_0, b_1, \ldots, b_m$.

Numbers γ_1 and γ_2 provide the sequence $c_0, c_1, \ldots, c_m$.

Numbers δ_1 and δ_2 provide the sequence $d_0, d_1, \ldots, d_m$.

$$\exists m \ (m \geq 1) \wedge \exists \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2$$

$$\text{Beta}(\alpha_1, \alpha_2, 0, 0) \wedge \text{Beta}(\beta_1, \beta_2, 0, 1) \wedge \text{Beta}(\gamma_1, \gamma_2, 0, 0) \wedge \text{Beta}(\delta_1, \delta_2, 0, 1) \wedge$$

$$\begin{aligned} \forall j \ [(j < m) \Rightarrow \exists a, b, c, d, a', b', c', d' \ \ & \text{Follow}(a, b, c, d, a', b', c', d') \wedge \\ & \text{Beta}(\alpha_1, \alpha_2, j, a) \wedge \text{Beta}(\alpha_1, \alpha_2, j+1, a') \wedge \\ & \text{Beta}(\beta_1, \beta_2, j, b) \wedge \text{Beta}(\beta_1, \beta_2, j+1, b') \wedge \\ & \text{Beta}(\gamma_1, \gamma_2, j, c) \wedge \text{Beta}(\gamma_1, \gamma_2, j+1, c') \wedge \\ & \text{Beta}(\delta_1, \delta_2, j, d) \wedge \text{Beta}(\delta_1, \delta_2, j+1, d')] \wedge \end{aligned}$$

$$\exists r \ \text{Beta}(\alpha_1, \alpha_2, m, r) \wedge \text{Beta}(\gamma_1, \gamma_2, m, r)$$

On the first line, the sentence chooses and commits to

- $m \geq 1$ (=the length of the PCP solution), and to
- the sequences $a_0, \ldots, a_m$ and $b_0, \ldots, b_m$ and $c_0, \ldots, c_m$ and $d_0, \ldots, d_m$, by choosing the corresponding numbers α_i and β_i and γ_i and δ_i.

$$\exists m\ (m \geq 1) \wedge \exists \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2$$

$$\mathrm{Beta}(\alpha_1, \alpha_2, 0, 0) \wedge \mathrm{Beta}(\beta_1, \beta_2, 0, 1) \wedge \mathrm{Beta}(\gamma_1, \gamma_2, 0, 0) \wedge \mathrm{Beta}(\delta_1, \delta_2, 0, 1) \wedge$$

$$\begin{aligned}\forall j\ [(j < m) \Rightarrow \exists a, b, c, d, a', b', c', d'\ \ &\mathrm{Follow}(a, b, c, d, a', b', c', d') \wedge \\ &\mathrm{Beta}(\alpha_1, \alpha_2, j, a) \wedge \mathrm{Beta}(\alpha_1, \alpha_2, j+1, a') \wedge \\ &\mathrm{Beta}(\beta_1, \beta_2, j, b) \wedge \mathrm{Beta}(\beta_1, \beta_2, j+1, b') \wedge \\ &\mathrm{Beta}(\gamma_1, \gamma_2, j, c) \wedge \mathrm{Beta}(\gamma_1, \gamma_2, j+1, c') \wedge \\ &\mathrm{Beta}(\delta_1, \delta_2, j, d) \wedge \mathrm{Beta}(\delta_1, \delta_2, j+1, d')] \wedge\end{aligned}$$

$$\exists r\ \mathrm{Beta}(\alpha_1, \alpha_2, m, r) \wedge \mathrm{Beta}(\gamma_1, \gamma_2, m, r)$$

On the second line we check that $a_0 = 0$ and $b_0 = 1$ and $c_0 = 0$ and $d_0 = 1$.

This is the start condition (=the empty prefix of the index word.)

$$\exists m \ (m \geq 1) \land \exists \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2$$

$$\text{Beta}(\alpha_1, \alpha_2, 0, 0) \land \text{Beta}(\beta_1, \beta_2, 0, 1) \land \text{Beta}(\gamma_1, \gamma_2, 0, 0) \land \text{Beta}(\delta_1, \delta_2, 0, 1) \land$$

$$\begin{aligned}\forall j \ [(j < m) \Rightarrow \exists a, b, c, d, a', b', c', d' \ \ &\text{Follow}(a, b, c, d, a', b', c', d') \land \\ &\text{Beta}(\alpha_1, \alpha_2, j, a) \land \text{Beta}(\alpha_1, \alpha_2, j+1, a') \land \\ &\text{Beta}(\beta_1, \beta_2, j, b) \land \text{Beta}(\beta_1, \beta_2, j+1, b') \land \\ &\text{Beta}(\gamma_1, \gamma_2, j, c) \land \text{Beta}(\gamma_1, \gamma_2, j+1, c') \land \\ &\text{Beta}(\delta_1, \delta_2, j, d) \land \text{Beta}(\delta_1, \delta_2, j+1, d')] \land\end{aligned}$$

$$\exists r \ \text{Beta}(\alpha_1, \alpha_2, m, r) \land \text{Beta}(\gamma_1, \gamma_2, m, r)$$

Next we check that for $j = 0, 1, \ldots, m-1$ the condition

$$\text{Follow}(a_j, b_j, c_j, d_j, a_{j+1}, b_{j+1}, c_{j+1}, d_{j+1})$$

holds. This condition means that the $(j+1)$'st numbers in the sequences are correctly calculated from the j'th numbers, corresponding to some next index choice in the PCP solution sequence.

$$\exists m\ (m \geq 1) \wedge \exists \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2$$

$$\text{Beta}(\alpha_1, \alpha_2, 0, 0) \wedge \text{Beta}(\beta_1, \beta_2, 0, 1) \wedge \text{Beta}(\gamma_1, \gamma_2, 0, 0) \wedge \text{Beta}(\delta_1, \delta_2, 0, 1) \wedge$$

$$\begin{aligned}
\forall j\ [(j < m) \Rightarrow \exists a, b, c, d, a', b', c', d'\ \ & \text{Follow}(a, b, c, d, a', b', c', d') \wedge \\
& \text{Beta}(\alpha_1, \alpha_2, j, a) \wedge \text{Beta}(\alpha_1, \alpha_2, j+1, a') \wedge \\
& \text{Beta}(\beta_1, \beta_2, j, b) \wedge \text{Beta}(\beta_1, \beta_2, j+1, b') \wedge \\
& \text{Beta}(\gamma_1, \gamma_2, j, c) \wedge \text{Beta}(\gamma_1, \gamma_2, j+1, c') \wedge \\
& \text{Beta}(\delta_1, \delta_2, j, d) \wedge \text{Beta}(\delta_1, \delta_2, j+1, d')] \wedge
\end{aligned}$$

$$\exists r\ \text{Beta}(\alpha_1, \alpha_2, m, r) \wedge \text{Beta}(\gamma_1, \gamma_2, m, r)$$

Finally, we check that $a_m = c_m$. This is the condition to verify that the index sequence we used is indeed a solution to the PCP instance.

$$\exists m \ (m \geq 1) \wedge \exists \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2, \delta_1, \delta_2$$

$$\text{Beta}(\alpha_1, \alpha_2, 0, 0) \wedge \text{Beta}(\beta_1, \beta_2, 0, 1) \wedge \text{Beta}(\gamma_1, \gamma_2, 0, 0) \wedge \text{Beta}(\delta_1, \delta_2, 0, 1) \wedge$$

$$\begin{aligned} \forall j \ [(j < m) \Rightarrow \exists a, b, c, d, a', b', c', d' \ \ & \text{Follow}(a, b, c, d, a', b', c', d') \wedge \\ & \text{Beta}(\alpha_1, \alpha_2, j, a) \wedge \text{Beta}(\alpha_1, \alpha_2, j+1, a') \wedge \\ & \text{Beta}(\beta_1, \beta_2, j, b) \wedge \text{Beta}(\beta_1, \beta_2, j+1, b') \wedge \\ & \text{Beta}(\gamma_1, \gamma_2, j, c) \wedge \text{Beta}(\gamma_1, \gamma_2, j+1, c') \wedge \\ & \text{Beta}(\delta_1, \delta_2, j, d) \wedge \text{Beta}(\delta_1, \delta_2, j+1, d')] \wedge \end{aligned}$$

$$\exists r \ \text{Beta}(\alpha_1, \alpha_2, m, r) \wedge \text{Beta}(\gamma_1, \gamma_2, m, r)$$

Conclusion: The sentence is true if and only if the PCP instance has a solution.

It follows that there does not exist an algorithm that can determine if a given sentence is true. Indeed, using such an algorithm we could decide whether any given PCP instance has a solution.

As an immediate corollary we see that the same problem is not even semi-decidable:

Corollary There is no semi-algorithm to determine if a given first-order arithmetic sentence is true.

Proof.

As an immediate corollary we see that the same problem is not even semi-decidable:

Corollary There is no semi-algorithm to determine if a given first-order arithmetic sentence is true.

Proof.

For any sentence φ either φ is true or $\neg\varphi$ is true. If there is a semi-algorithm to detect true sentences then we can execute this semi-algorithm on inputs φ and $\neg\varphi$ in parallel until we receive an answer. This provides an algorithm to determine whether φ is true, contradicting the theorem we just proved.

Remark: Consider any sound axiomatization of true first order sentences on natural numbers. This means a finite collection of (true) **axioms** and effective **inference rules** that can be used to deduce new true sentences from other known true sentences. Let us call a sentence **provable** in this axiomatization if it can be deduced from the axioms. Every provable sentence is true (soundness).

The set of provable sentences is **semi-decidable**: a semi-algorithm to check the provability of a given sentence φ guesses and verifies a deduction of φ from the axioms.

Because the set of true sentences is not semi-decidable we see that the sets of provable and true sentences cannot be the same set. There exists a true sentence that is not provable in the axiomatization. The axiomatization is **not complete**.

Remark: Consider any sound axiomatization of true first order sentences on natural numbers. This means a finite collection of (true) **axioms** and effective **inference rules** that can be used to deduce new true sentences from other known true sentences. Let us call a sentence **provable** in this axiomatization if it can be deduced from the axioms. Every provable sentence is true (soundness).

The set of provable sentences is **semi-decidable**: a semi-algorithm to check the provability of a given sentence φ guesses and verifies a deduction of φ from the axioms.

Because the set of true sentences is not semi-decidable we see that the sets of provable and true sentences cannot be the same set. There exists a true sentence that is not provable in the axiomatization. The axiomatization is **not complete**.

This is **Gödel's first incompleteness theorem**.

The previous remark can be generalized to any effective axiomatization. (For example, the set of axioms does no need to be finite – it is ok to have a recursively enumerable set of axioms.) In the full generality, an **effective axiomatization** is a semi-algorithm to recognize some subset of true sentences.

Because the set of true sentences is not semi-decidable, no effective axiomatization can capture all true sentences.

The previous remark can be generalized to any effective axiomatization. (For example, the set of axioms does no need to be finite – it is ok to have a recursively enumerable set of axioms.) In the full generality, an **effective axiomatization** is a semi-algorithm to recognize some subset of true sentences.

Because the set of true sentences is not semi-decidable, no effective axiomatization can capture all true sentences.

There exist some true sentences that we can never know for sure to be true!

(This gets a bit philosophical, but I claim here that the set of sentences that we can know to be true for sure must be a semi-decidable set: we must be able to provide a convincing and mechanically step-by-step verifiable argument for the fact that the sentence is true.)

A similar argument can be made for **any non-semi-decidable problem**: they must have positive instances that we cannot prove to be positive instance.

So there exists

- a Turing machine that does not halt from the blank tape but we'll never know this non-haltingness for sure,
- a context-free grammar that is unambiguous but we'll never know its unambiguity for sure,
- a PCP instance without a solution but we can never be sure that it does not have a solution,
- a Wang tile set that tiles the plane but we never know for sure it tiles the plane.
- ...

Remark: An important subfamily of first-order arithmetic formulas are **diophantine equations**. These are polynomial equations with integer coefficients, and one is interested in finding integer solutions.

For example,

$$x^3 + y^3 = z^3$$

is a diophantine equation with three variables. It has solutions in $\mathbb{N}$ (for example $x = y = z = 0$) so the first-order sentence

$$\exists x\ \exists y\ \exists z\ x^3 + y^3 = z^3$$

is true.

On the other hand, the sentence

$$\exists x\ x^2 + 1 = 0$$

is clearly false in $\mathbb{N}$.

In year 1900, **D. Hilbert** proposed a list of 23 open mathematical problems, that greatly influenced the mathematical research in the 20'th century. The 10'th problem in the list asked to "device a process" (=algorithm, in modern terms) to determine if a given diophantine equation has an integer solution.

(The problem is algorithmically equivalent if natural number solutions are required rather than integer solutions.)

In year 1900, **D. Hilbert** proposed a list of 23 open mathematical problems, that greatly influenced the mathematical research in the 20'th century. The 10'th problem in the list asked to "device a process" (=algorithm, in modern terms) to determine if a given diophantine equation has an integer solution.

(The problem is algorithmically equivalent if natural number solutions are required rather than integer solutions.)

In 1970, the problem was proved undecidable by **Y. Matiyasevich**. This means that there is no algorithm to determine if a given first-order sentence

$$\exists x \exists y \ldots \; P(x, y, \ldots) = Q(x, y, \ldots)$$

over $\mathbb{N}$ is true. Here P and Q are given polynomials with natural number coefficients (*i.e.*, P and Q are terms formed using $+$, $\cdot$, variables and constants.)

This result is much stronger than the undecidability result we have proved here, because the types of considered sentences are of very restricted form.

In year 1900, **D. Hilbert** proposed a list of 23 open mathematical problems, that greatly influenced the mathematical research in the 20'th century. The 10'th problem in the list asked to "device a process" (=algorithm, in modern terms) to determine if a given diophantine equation has an integer solution.

(The problem is algorithmically equivalent if natural number solutions are required rather than integer solutions.)

In 1970, the problem was proved undecidable by **Y. Matiyasevich**. This means that there is no algorithm to determine if a given first-order sentence

$$\exists x \exists y \ldots \; P(x, y, \ldots) = Q(x, y, \ldots)$$

over $\mathbb{N}$ is true. Here P and Q are given polynomials with natural number coefficients (*i.e.*, P and Q are terms formed using $+$, $\cdot$, variables and constants.)

This result is much stronger than the undecidability result we have proved here, because the types of considered sentences are of very restricted form.

Clearly having solutions is semi-decidable, so not having them is not semi-decidable. As discussed above, there hence exists an equation that does not have a solution in $\mathbb{N}$ but it is unprovable that it does not have a solution.

Computable functions and reducibility

Informally, we say that a (total) function

$$f : \Sigma^* \longrightarrow \Delta^*$$

is **total computable** (or **total recursive**) if there is an algorithm that produces from any input $w \in \Sigma^*$ the output $f(w) \in \Delta^*$.

Computable functions and reducibility

Informally, we say that a (total) function

$$f : \Sigma^* \longrightarrow \Delta^*$$

is **total computable** (or **total recursive**) if there is an algorithm that produces from any input $w \in \Sigma^*$ the output $f(w) \in \Delta^*$.

A partial function $f : \Sigma^* \longrightarrow \Delta^*$ is **partial computable** (or **partial recursive**) if there is a semi-algorithm-like process that produces from input $w \in \Sigma^*$ the output $f(w) \in \Delta^*$ if $f(w)$ is defined, and returns no answer if $f(w)$ is not defined.

(We define these concepts later more rigorously using Turing machines.)

Typically algorithms are applied on objects that are not words. But the objects can be **encoded** as words. Let us denote by $\langle x \rangle$ the encoding of an object x. So $\langle x \rangle$ is a word over some alphabet Σ.

If X is the set of object we are encoding, it is reasonable to require that

- the encoding function is **one-to-one** on X so that different objects $x_1 \neq x_2$ have different encodings $\langle x_1 \rangle \neq \langle x_2 \rangle$, and
- the set $\{\langle x \rangle \mid x \in X\}$ of valid encodings is a **recursive language**. So there is an algorithm to check if a given word is a valid encoding.

Typically algorithms are applied on objects that are not words. But the objects can be **encoded** as words. Let us denote by $\langle x \rangle$ the encoding of an object x. So $\langle x \rangle$ is a word over some alphabet Σ.

If X is the set of object we are encoding, it is reasonable to require that

- the encoding function is **one-to-one** on X so that different objects $x_1 \neq x_2$ have different encodings $\langle x_1 \rangle \neq \langle x_2 \rangle$, and
- the set $\{\langle x \rangle \mid x \in X\}$ of valid encodings is a **recursive language**. So there is an algorithm to check if a given word is a valid encoding.

In algorithms that deal with objects in X one is not usually concerned about the encoding function $\langle \cdot \rangle$, but one expects some "reasonable" encoding. (Unreasonable encoding could, for example, include in the encoding an extra bit that gives the answer to the decision problem we are interested to solve!)

All reasonable encodings can be effectively converted into each other, and then the choice of the encoding is rather arbitrary from the computability point of view. The fact that encodings $\langle \cdot \rangle_1$ and $\langle \cdot \rangle_2$ on the set X can be effectively converted into each other means that there are total computable word functions f and g that map

$$\begin{aligned} f &: \langle x \rangle_1 \mapsto \langle x \rangle_2 \\ g &: \langle x \rangle_2 \mapsto \langle x \rangle_1 \end{aligned}$$

for all $x \in X$.

We now say that a function $f : D \longrightarrow R$ between sets D and R of objects is **total computable** (with respect to some encoding schemes for D and R) if the corresponding word function between encodings of objects is total computable. More precisely, the function that maps

$$\langle d \rangle \mapsto \langle f(d) \rangle$$

for all $d \in D$ should be total computable.

We now say that a function $f : D \longrightarrow R$ between sets D and R of objects is **total computable** (with respect to some encoding schemes for D and R) if the corresponding word function between encodings of objects is total computable. More precisely, the function that maps

$$\langle d \rangle \mapsto \langle f(d) \rangle$$

for all $d \in D$ should be total computable.

Similarly, we say that a partial function $f : D \longrightarrow R$ is **partial computable** if there is a partial computable word function that

- maps $\langle d \rangle \mapsto \langle f(d) \rangle$ for all $d \in D$ such that $f(d)$ is defined,
- is undefined on $\langle d \rangle$ for all $d \in D$ such that $f(d)$ is undefined.

Example. For $D = R = \mathbb{N}$ we may choose to use the unary encoding

$$\langle n \rangle = a^n$$

in the single letter alphabet $\{a\}$.

With this encoding, a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ is total computable iff the function

$$a^n \mapsto a^{f(n)}$$

is a total computable word function $a^* \longrightarrow a^*$.

Example. For $D = R = \mathbb{N}$ we may choose to use the unary encoding

$$\langle n \rangle = a^n$$

in the single letter alphabet $\{a\}$.

With this encoding, a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ is total computable iff the function

$$a^n \mapsto a^{f(n)}$$

is a total computable word function $a^* \longrightarrow a^*$.

If we decided to use instead the usual binary encoding $\text{bin}(n)$ of numbers in the binary alphabet $\{0, 1\}$ then exactly the same functions $f : \mathbb{N} \longrightarrow \mathbb{N}$ would be computable because there are total computable conversions

$$\text{bin}(n) \longleftrightarrow a^n$$

between the unary and binary encodings.

Example. The Busy beaver function

$$BB : \mathbb{N} \longrightarrow \mathbb{N}$$

we have seen in the home exercises is not total computable (using any reasonable encoding of numbers).

In fact, for any total computable function $f : \mathbb{N} \longrightarrow \mathbb{N}$ there exists $n \in \mathbb{N}$ such that $BB(n) > f(n)$.

Let us now define formally the concepts of total and partial computable (word) functions.

Let $M = (Q, \Sigma \cup \Delta, \Gamma, \delta, q_0, B, q_F)$ be a Turing machine, with the final state q_F. We define the halting ID ζ_w corresponding to word $w \in \Delta^*$ analogously to the initial ID ι_w:

$$\zeta_w = \begin{cases} q_F w, & \text{if } w \neq \varepsilon, \\ q_F B, & \text{if } w = \varepsilon. \end{cases}$$

The partial function $f_M : \Sigma^* \longrightarrow \Delta^*$ computed by M is

$$f_M(w) = \begin{cases} u, & \text{if } \iota_w \vdash^* \zeta_u \text{ and } u \in \Delta^* , \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

In other words, we require the Turing machine to halt in the state q_F, reading the first letter of the output word $f_M(w)$. In all other cases (if M does not halt or halts in a different ID) the value $f_M(w)$ is undefined.

A function f is a **partial computable function** if there exists a Turing machine M such that $f = f_M$. If, moreover, f is defined for all $w \in \Sigma^*$ then f is a **total computable function**.

Normally we use informal descriptions of computable functions, but let us do one example in detail using a Turing machine.

Example. Let us prove that the function $f : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ that maps

$$(n, m) \mapsto n + m$$

is total computable when we use the encodings $\langle n, m \rangle = a^n \# a^m$ and $\langle k \rangle = a^k$ for all $n, m, k \in \mathbb{N}$.

Normally we use informal descriptions of computable functions, but let us do one example in detail using a Turing machine.

Example. Let us prove that the function $f : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ that maps

$$(n, m) \mapsto n + m$$

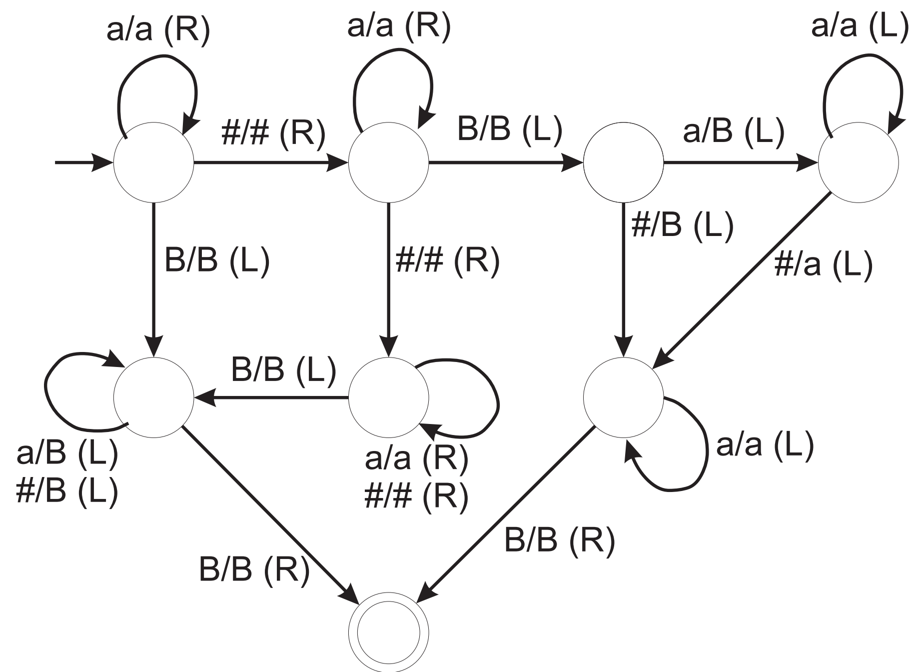
is total computable when we use the encodings $\langle n, m \rangle = a^n \# a^m$ and $\langle k \rangle = a^k$ for all $n, m, k \in \mathbb{N}$.

It is enough to construct a Turing machine that maps

$$\begin{array}{rcll} a^n \# a^m & \mapsto & a^{n+m} & \quad \text{for all } n, m \geq 0 \\ w & \mapsto & \varepsilon & \quad \text{for all } w \notin a^* \# a^* \end{array}$$

$$\begin{aligned} a^n\#a^m &\mapsto a^{n+m} \qquad \text{for all } n,m \geq 0 \\ w &\mapsto \varepsilon \qquad \text{for all } w \notin a^*\#a^* \end{aligned}$$

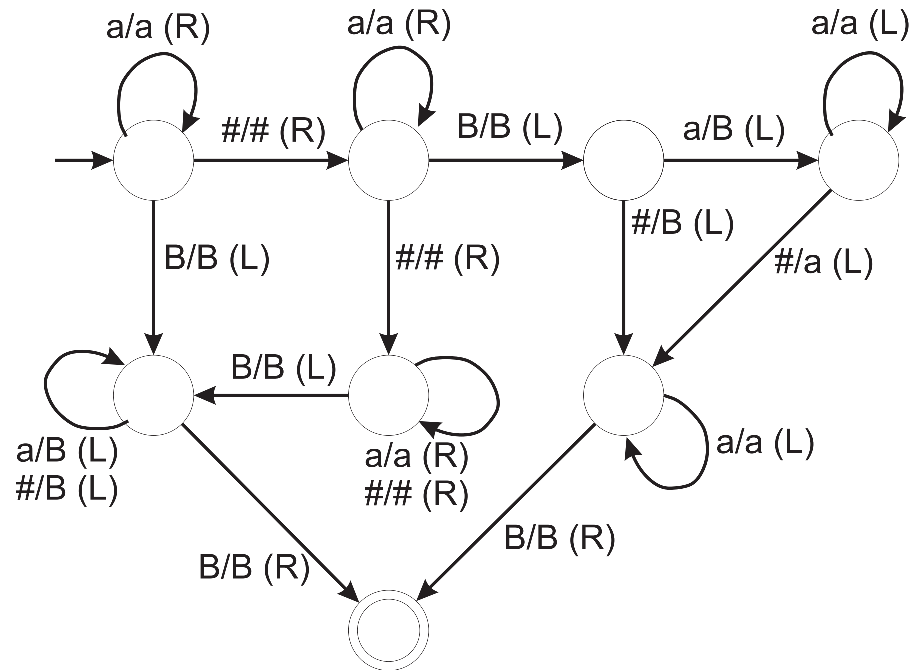
A possible TM is described by the diagram



whose vertices are the states, and for each transition $\delta(q,a) = (p,b,D)$ there is an edge in the diagram from vertex q into vertex p labeled by $a/b\ (D)$. The initial state q_0 is indicated by a single incoming arrow and the final state q_F is the one with a double circle.

$$\begin{aligned} a^n\#a^m &\mapsto a^{n+m} && \text{for all } n,m \geq 0 \\ w &\mapsto \varepsilon && \text{for all } w \notin a^*\#a^* \end{aligned}$$

A possible TM is described by the diagram



The machine first scans the input from left-to-right to check that the input belongs to $a^*\#a^*$. (If not, the machine erases the input and halts.) It then returns to the beginning, replacing the symbol # by the last a of the input.

$$\begin{aligned} a^n\#a^m &\mapsto a^{n+m} && \text{for all } n, m \geq 0 \\ w &\mapsto \varepsilon && \text{for all } w \notin a^*\#a^* \end{aligned}$$

We can simplify the construction by ignoring what happens on words that are no valid encodings.

Since the set of valid encodings of $D = \mathbb{N} \times \mathbb{N}$ is a recursive language $a^*\#a^*$, it is actually enough to build a Turing machine M that correctly operates on words that belong to $a^*\#a^*$. It is irrelevant what happens on inputs that are not valid encodings.

We can namely the build a TM that first check whether the input w is a valid encoding. If it is not, one returns an arbitrary output (say ε). If w a valid encoding, one starts M on w.

Remark. A language $L \subseteq \Sigma^*$ is recursive if and only if its characteristic function $\chi_L \colon \Sigma^* \longrightarrow \{0, 1\}^*$ defined by

$$\chi_L(w) = \begin{cases} 1, & \text{if } w \in L, \\ 0, & \text{if } w \notin L \end{cases}$$

is total computable.

Analogously, a language $L \subseteq \Sigma^*$ is recursively enumerable iff the partial function $\chi'_L \colon \Sigma^* \longrightarrow \{0, 1\}^*$ defined by

$$\chi'_L(w) = \begin{cases} 1, & \text{if } w \in L, \\ \text{undefined}, & \text{if } w \notin L \end{cases}$$

is partial computable.

Turing machines are great, but rather cumbersome to construct (to say the least).

As any informally described algorithm can be implemented as a Turing machine, in the following we describe total and partial computable functions as informal algorithms.

Example. Some time ago we saw an effective construction that builds for any given TM M and input word w a new TM M' with the property that $L(M') = \{a, b\}^*$ if $w \in L(M)$, and $L(M') = \emptyset$ if $w \notin L(M)$.

This means that there is a corresponding total computable word function that maps

$$\langle M \rangle \# w \mapsto \langle M' \rangle$$

We did not explicitly construct a Turing machine that computes this word function, but we know that it exists.

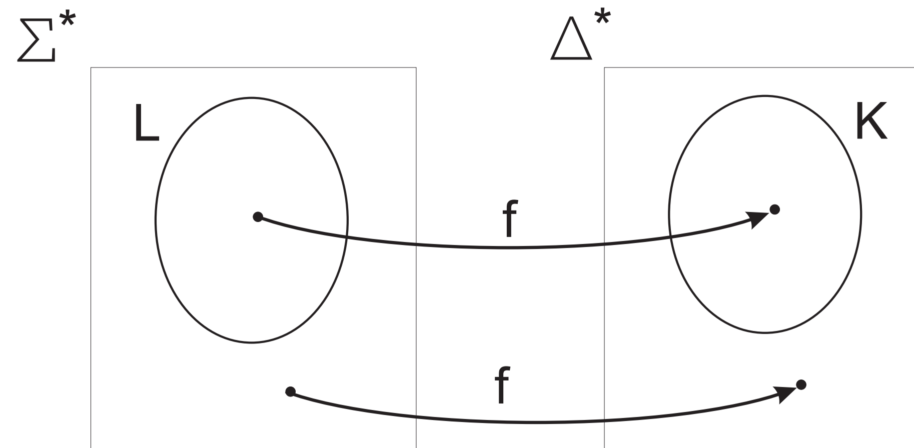
Let $L \subseteq \Sigma^*$ and $K \subseteq \Delta^*$ be two languages. A total computable function

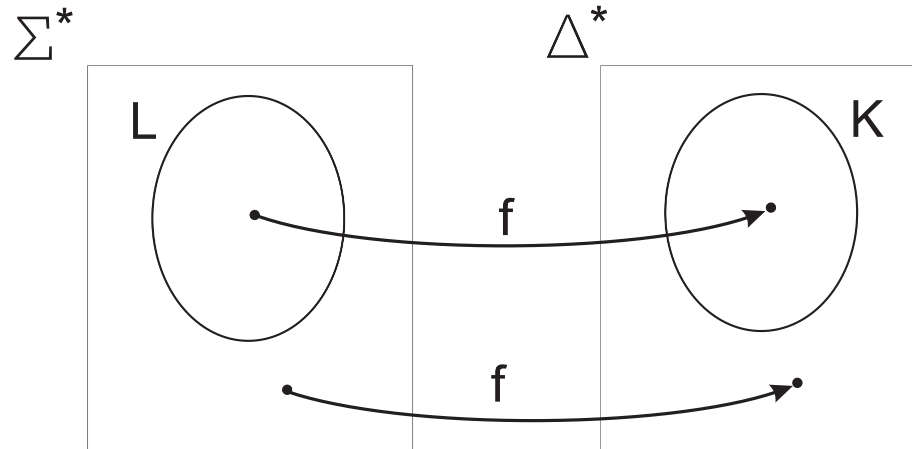
$$f : \Sigma^* \longrightarrow \Delta^*$$

is a **many-one reduction** of L to K if for all $w \in \Sigma^*$

$$w \in L \iff f(w) \in K.$$

If a reduction exists from L to K, we denote $L \leq_m K$ and say that L is **many-one reducible** (or **mapping reducible**) to K.





Theorem. Let $L \subseteq \Sigma^*$ and $K \subseteq \Delta^*$ be languages such that $L \leq_m K$. Then also $\Sigma^* \setminus L \leq_m \Delta^* \setminus K$. Moreover,

- if K is recursive then L is recursive,
- if K is recursively enumerable then L is recursively enumerable,
- if $\Delta^* \setminus K$ is recursively enumerable then $\Sigma^* \setminus L$ is recursively enumerable.

Proof.

More informally, we say that a decision problem A is many-one reducible to a decision problem B if there is an effective conversion that takes instances of A into equivalent instances of B. (Equivalent=both positive or both negative)

We then denote $A \leq_m B$.

It follows that if A is known to be undecidable (or non-semi-decidable) then B is undecidable (or non-semi-decidable) as well.

Our undecidability proofs have been (mostly) of this type.

More informally, we say that a decision problem A is many-one reducible to a decision problem B if there is an effective conversion that takes instances of A into equivalent instances of B. (Equivalent=both positive or both negative)

We then denote $A \leq_m B$.

It follows that if A is known to be undecidable (or non-semi-decidable) then B is undecidable (or non-semi-decidable) as well.

Our undecidability proofs have been (mostly) of this type.

Example. There is a many-one reduction from $L_u = \{\langle M\rangle\#w \mid w \in L(M)\}$ to $K = \{\langle M\rangle \mid L(M) \neq \emptyset\}$. Because the complement of L_u is not r.e., we concluded that the complement of K is not r.e. either.

Transitivity of $\leq_m$ is obvious:

Lemma. If $L_1 \leq_m L_2$ and $L_2 \leq_m L_3$ then $L_1 \leq_m L_3$.

Proof. Let f_1 be a many-one reduction of L_1 to L_2. Let f_2 be a many-one reduction of L_2 to L_3. Then the composition $w \mapsto f_2(f_1(w))$ is a reduction of L_1 to L_3.

An r.e language K is **r.e.-complete** if for every r.e. language L we have $L \leq_m K$. In other words, K is an r.e. language such that all r.e. languages can be many-one reduced to K.

A decision problem is r.e.-complete if the language of (encodings of) its positive instances is r.e.-complete.

Theorem. The language $L_u = \{\langle M \rangle \# w \mid w \in L(M)\}$ is r.e.-complete.

Proof.

Once a language L is known to be r.e.-complete, many-one reductions can be used to find other r.e.-complete languages:

Theorem. Let L be an r.e.-complete language. If K is an r.e. language such that $L \leq_m K$ then also K is r.e.-complete

Proof.

Once a language L is known to be r.e.-complete, many-one reductions can be used to find other r.e.-complete languages:

Theorem. Let L be an r.e.-complete language. If K is an r.e. language such that $L \leq_m K$ then also K is r.e.-complete

Proof. Let M be an arbitrary r.e. language. Then $M \leq_m L$, and so by transitivity of $\leq_m$ we have $M \leq_m K$.

This means that all semi-decidable decision problems that we proved undecidable using a many-one reduction from L_u are r.e.-complete. So, for example, all the following decision problems are r.e-complete:

- "Does a given TM halt when started on the blank initial tape ?"
- "Does a given PCP instance have a solution ?"
- "Is $L_1 \cap L_2 \neq \emptyset$ for given CFLs L_1 and L_2 ?"
- "Is a given CFG G ambiguous ?"
- "Is $L \neq \Sigma^*$ for a given CFL L ?"
- "Does a given Wang tile set not admit a valid tiling that contains a given seed tile ?"

If (not necessarily r.e.) language K has the property that $L \leq_m K$ for all r.e. languages L then we say that K is **r.e.-hard**.

Hence

$$L \text{ is r.e.-complete} \iff L \text{ is r.e.-hard and } L \text{ is r.e.}$$

If (not necessarily r.e.) language K has the property that $L \leq_m K$ for all r.e. languages L then we say that K is **r.e.-hard**.

Hence

$$L \text{ is r.e.-complete} \quad \Longleftrightarrow \quad L \text{ is r.e.-hard and } L \text{ is r.e.}$$

Clearly a language is r.e.-hard if some r.e.-complete language K can be many-one reduced into it. Analogous terminology is used on decision problems.

- The proof of the Rice's theorem shows that all non-trivial questions (or their complements) about r.e. languages are r.e.-hard.
- We have also shown the r.e.-hardness of determining whether a given first-order sentence over $\mathbb{N}$ is true.

Remark. Many-one reducibility is only one type of reducibility that can be used to show that a language is not r.e.

A language L is said to be **Turing reducible** to a language K if there is an algorithm for the membership problem of L that may use as a subroutine a hypothetical algorithm (=oracle) that solves the membership problem of K. The oracle may be invoked several times and the input to the oracle may depend on the answers the oracle has provided on previous queries. We denote this

$$L \leq_T K$$

Remark. Many-one reducibility is only one type of reducibility that can be used to show that a language is not r.e.

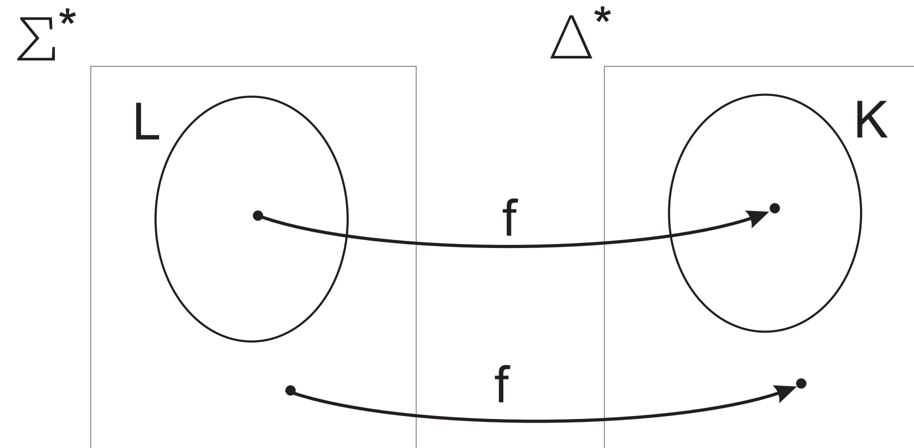
A language L is said to be **Turing reducible** to a language K if there is an algorithm for the membership problem of L that may use as a subroutine a hypothetical algorithm (=oracle) that solves the membership problem of K. The oracle may be invoked several times and the input to the oracle may depend on the answers the oracle has provided on previous queries. We denote this

$$L \leq_T K$$

Many-one reductions are special types of Turing reductions where the oracle is invoked only once, and this happens at the end of the algorithm and the answer from the oracle is directly relayed as the final answer of the algorithm.

Still, if $L \leq_T K$ and K is known to be recursive (or r.e) then L has to be recursive (or r.e., respectively) as well. So the more powerful Turing reductions can also be used to prove undecidability results.

Remark. The name **many-one reduction** comes from the fact that the reduction $f\colon \Sigma^* \longrightarrow \Delta^*$ is not required to be injective.



A weaker type of reductions called **one-one reduction** requires f to be injective.

Tag systems

An example of an alternative computation model to Turing machines.

A **tag system** is a triple $T = (\Sigma, k, g)$ where

- Σ is a finite alphabet,
- $k \geq 1$ is the **deletion number** and
- $g : \Sigma \longrightarrow \Sigma^*$ is a function assigning a word to each letter in Σ.

A tag-system with deletion number k is called a k**-tag system**.

Tag systems

An example of an alternative computation model to Turing machines.

A **tag system** is a triple $T = (\Sigma, k, g)$ where

- Σ is a finite alphabet,
- $k \geq 1$ is the **deletion number** and
- $g : \Sigma \longrightarrow \Sigma^*$ is a function assigning a word to each letter in Σ.

A tag-system with deletion number k is called a **k-tag system**.

A k-tag system is a **deterministic rewrite system.** Any word $u \in \Sigma^*$ whose length is at least k gets rewritten as follows: Write $u = avw$ where $a \in \Sigma$ and $|av| = k$. Then

$$u \Rightarrow wg(a)$$

(In other words, the first k letters of u are erased and the word $g(a)$ is appended to the end where a is the first letter of u.)

The rewriting **terminates** if a word is reached whose length is less than k.

Example. The 3-tag system $(\{0,1\},3,g)$ where $g(0) = 00$ and $g(1) = 1101$ was studied (and found intractable) by E.Post. For example, starting from the word 10010 we obtain the computation

1 0 0 1 0

Example. The 3-tag system $(\{0,1\},3,g)$ where $g(0) = 00$ and $g(1) = 1101$ was studied (and found intractable) by E.Post. For example, starting from the word 10010 we obtain the computation

$$\begin{array}{cccccccccc} 1 & 0 & 0 & 1 & 0 & & & & \\ & & & 1 & 0 & 1 & 1 & 0 & 1 \end{array}$$

Example. The 3-tag system $(\{0,1\},3,g)$ where $g(0) = 00$ and $g(1) = 1101$ was studied (and found intractable) by E.Post. For example, starting from the word 10010 we obtain the computation

$$
\begin{array}{cccccccccccc}
1 & 0 & 0 & 1 & 0 & & & & & & & \\
 & & & 1 & 0 & 1 & 1 & 0 & 1 & & & \\
 & & & & & & 1 & 0 & 1 & 1 & 1 & 0 & 1
\end{array}
$$

Example. The 3-tag system $(\{0,1\},3,g)$ where $g(0) = 00$ and $g(1) = 1101$ was studied (and found intractable) by E.Post. For example, starting from the word 10010 we obtain the computation

$$
\begin{array}{llllllllllllllllll}
1 & 0 & 0 & 1 & 0 & & & & & & & & & & & & & \\
& & & 1 & 0 & 1 & 1 & 0 & 1 & & & & & & & & & \\
& & & & & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & & & & & \\
& & & & & & & & & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 &
\end{array}
$$

Example. The 3-tag system $(\{0,1\},3,g)$ where $g(0) = 00$ and $g(1) = 1101$ was studied (and found intractable) by E.Post. For example, starting from the word 10010 we obtain the computation

```
1 0 0 1 0
      1 0 1 1 0 1
            1 0 1 1 1 0 1
                  1 1 0 1 1 1 0 1
                        1 1 1 0 1 1 1 0 1
```

Example. The 3-tag system $(\{0,1\},3,g)$ where $g(0)=00$ and $g(1)=1101$ was studied (and found intractable) by E.Post. For example, starting from the word 10010 we obtain the computation

```
1 0 0 1 0
      1 0 1 1 0 1
            1 0 1 1 1 0 1
                  1 1 0 1 1 1 0 1
                        1 1 1 0 1 1 1 0 1
                              0 1 1 1 0 1 1 1 0 1
```

Example. The 3-tag system $(\{0,1\},3,g)$ where $g(0)=00$ and $g(1)=1101$ was studied (and found intractable) by E.Post. For example, starting from the word 10010 we obtain the computation

```
1 0 0 1 0
      1 0 1 1 0 1
            1 0 1 1 1 0 1
                  1 1 0 1 1 1 0 1
                        1 1 1 0 1 1 1 0 1
                              0 1 1 1 0 1 1 1 0 1
                                    1 0 1 1 1 0 1 0 0
```

$10010 \Rightarrow$

$101101 \Rightarrow$

$1011101 \Rightarrow$

$11011101 \Rightarrow$

$111011101 \Rightarrow$

$0111011101 \Rightarrow$

$101110100 \Rightarrow$

$1101001101 \Rightarrow$

$10011011101 \Rightarrow$

$110111011101 \Rightarrow$

$1110111011101 \Rightarrow$

$01110111011101 \Rightarrow$

$1011101110100 \Rightarrow$

$11011101001101 \Rightarrow$

$111010011011101 \Rightarrow$

$0100110111011101 \Rightarrow$

$\color{red}{011011101110100} \Rightarrow$

$01110111010000 \Rightarrow$

$1011101000000 \Rightarrow$

$11010000001101 \Rightarrow$

$100000011011101 \Rightarrow$

$0000110111011101 \Rightarrow$

$\color{red}{011011101110100}$

The computation enters a loop, and thus never halts.

$10010 \Rightarrow$
$101101 \Rightarrow$
$1011101 \Rightarrow$
$11011101 \Rightarrow$
$111011101 \Rightarrow$
$0111011101 \Rightarrow$
$101110100 \Rightarrow$
$1101001101 \Rightarrow$
$10011011101 \Rightarrow$
$110111011101 \Rightarrow$
$1110111011101 \Rightarrow$
$01110111011101 \Rightarrow$
$1011101110100 \Rightarrow$
$11011101001101 \Rightarrow$
$111010011011101 \Rightarrow$
$0100110111011101 \Rightarrow$
$0110111011110100 \Rightarrow$
$01110111010000 \Rightarrow$
$1011101000000 \Rightarrow$
$11010000001101 \Rightarrow$
$100000011011101 \Rightarrow$
$0000110111011101 \Rightarrow$
0110111011110100

The computation enters a loop, and thus never halts.

On the other hand, from the initial word 100100100000 one reaches in 419 steps the halting word 00.

Example. Consider the 2-tag system $T = (\{a, b, c\}, 2, g)$ with g that maps

$$\begin{aligned} a &\mapsto bc \\ b &\mapsto a \\ c &\mapsto aaa \end{aligned}$$

If we represent a number n as the word a^n then the system simulates the famous Collatz-function

$$f(n) = \begin{cases} n/2, & \text{if } n \text{ is even,} \\ 3n+1, & \text{if } n \text{ is odd.} \end{cases}$$

It is indeed easy to verify that

$$\begin{array}{llll} a^n & \Rightarrow^{n} & a^{n/2} & \text{if } n > 1 \text{ is even} \\ a^n & \Rightarrow^{n+1} & a^{(3n+1)/2} & \text{if } n > 1 \text{ is odd} \end{array}$$

Hence the word a^n eventually halts if and only if iterating f from n eventually leads to number 1.

Let T be a tag-system. The **language** $L(T)$ defined by T is the set of words that eventually halt (*i.e.*, evolve into a word of length less than k). Clearly $L(T)$ is recursively enumerable. But it can be non-recursive:

Theorem. There exists a 2-tag system T such that the language $L(T)$ is r.e.-complete.

Proof. Skipped

Remark. Another natural decision problem associated to a tag-system is the **word problem**, asking whether a given word x derives another given word y. As the halting problem, also the word problem is **semi-decidable**.

Moreover, the halting problem of any tag-system can be Turing reduced to its word problem: Solving the word problem with all target words y that are shorter than k will tell whether the source word eventually halts. So also the word problem must be undecidable.

Remark. Our sample 3-tag $(\{0,1\}, 3, g)$ with $g(0) = 00$ and $g(1) = 1101$ is still a mystery. It is not known whether its word problem is decidable or not. It is not even known whether there exists any word that neither eventually halts nor enters a cycle.

(If one of these always happens then the word problem and the halting problem are both decidable.)