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.

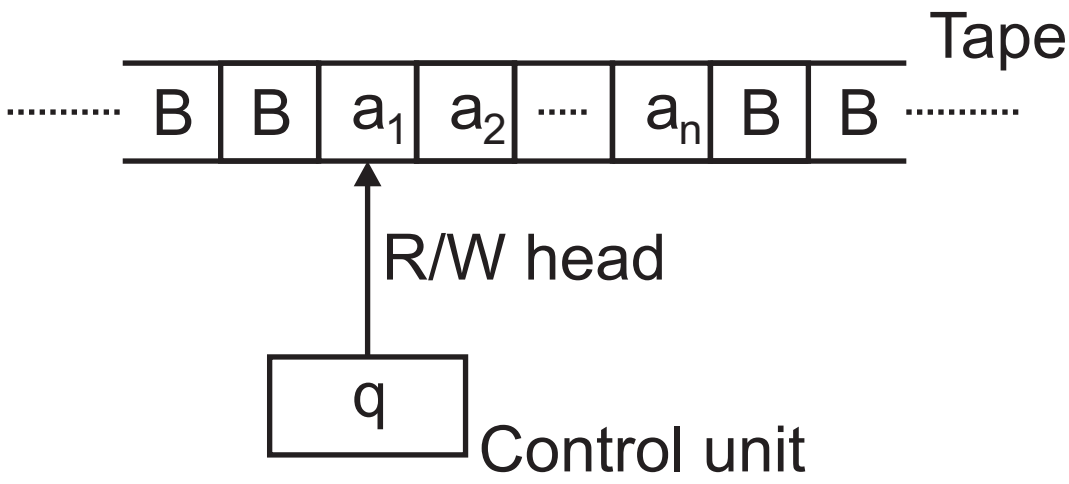
Church's thesis is the claim that Turing machines capture the full power of any effective method of computation. One cannot of course **prove** that nobody ever invents a totally different, more powerful computer architecture. But all evidence indicates that this is not possible, as long as we require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

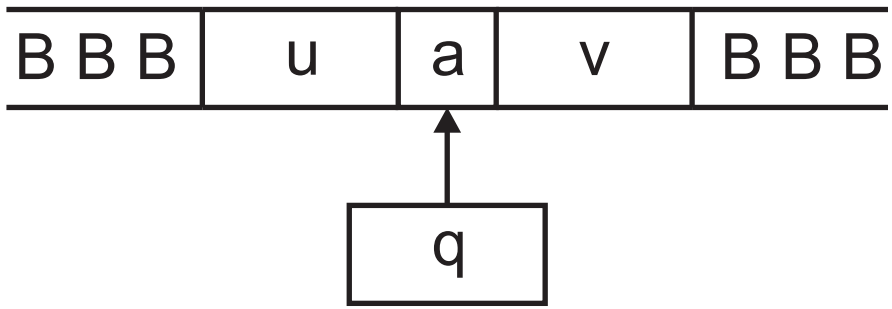
If after some moves the machine enters the final state f the input word is accepted, otherwise it is rejected.

(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

$$u\ q\ a\ v$$

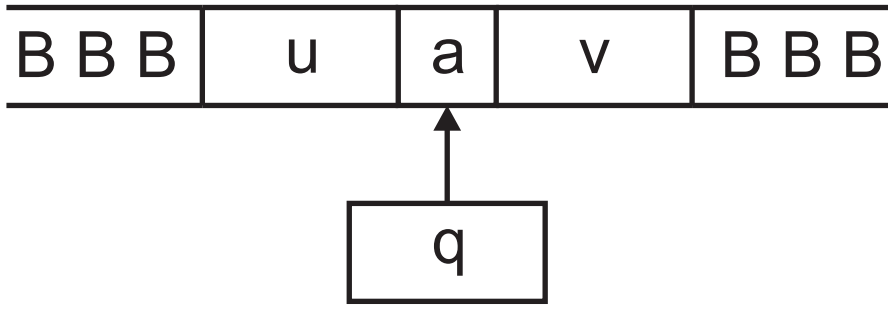
where $q \in Q$ is the state of the control unit, $u, v \in \Gamma^*$ are the content of the tape to the left and to the right of the R/W head, respectively, until the last non-blank symbols, and $a \in \Gamma$ is the symbol currently scanned by the R/W head:



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

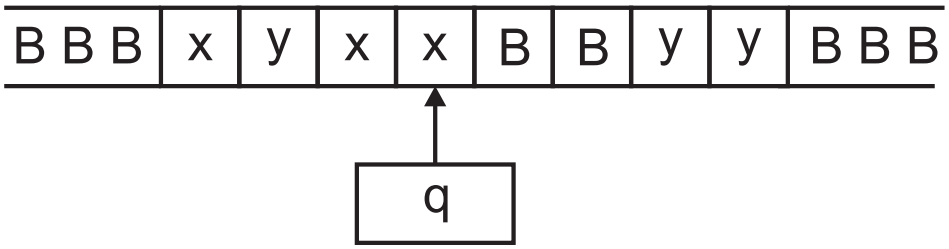
$$u\ q\ a\ v$$

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:

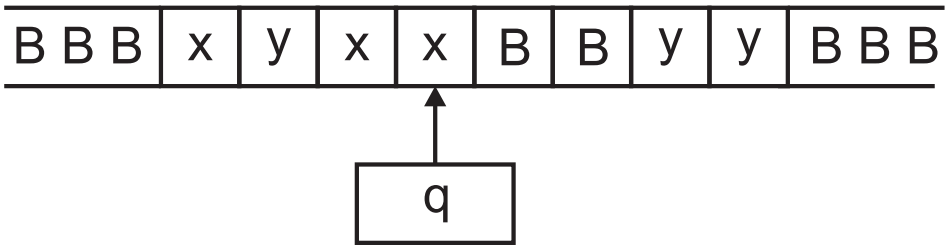


So: The first (last) letter of u (or v, respectively) is not B, because the leading and trailing B's are trimmed away from u and v, respectively. Also that the current state q can be uniquely identified from the word $u\ q\ a\ v$ because we required alphabets Q and Γ to be disjoint.

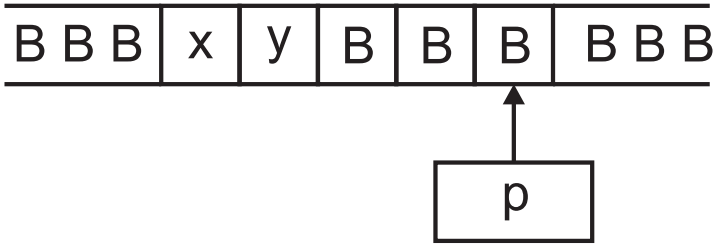
Example. $xyxqxBByy$ (where $q \in Q$) represents the situation



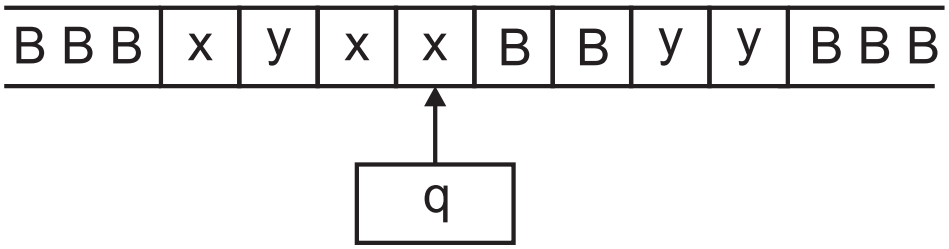
Example. $xyxqxBByy$ (where $q \in Q$) represents the situation



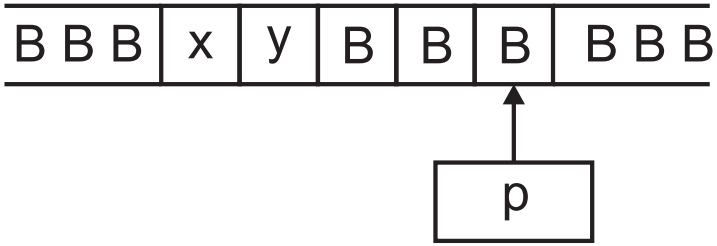
$xyBBpB$ (where $p \in Q$) represents



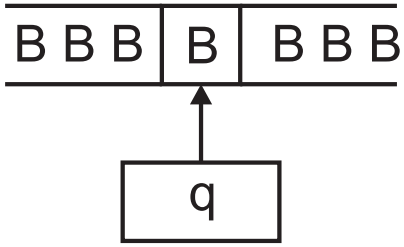
Example. $xyxqxBByy$ (where $q \in Q$) represents the situation



$xyBBpB$ (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

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

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

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

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

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

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

$$
\begin{aligned}
aB\ q\ B &\vdash \\
ab\ q\ B &\vdash
\end{aligned}
$$

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

$$aB\ q\ B \vdash a\ p\ B$$
$$ab\ q\ B \vdash a\ p\ b$$

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

$$aB\ q\ B \vdash$$

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

$$aB\ q\ B \vdash a\ p\ B$$
$$ab\ q\ B \vdash a\ p\ b$$

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:

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

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!)