2 – Automata, Turing machines

Luc Lapointe
luc.lapointe@ens-paris-saclay.fr
home.lmf.cnrs.fr/LucLapointe/

Abstract

One aim of computer science is to study what can computer achieve, and how efficiently. Abstract machine models allow to fulfill this aim independently of the hardware fast-growing efficiency. *Automata* are abstract memoryless machines. They can efficiently recognize patterns described by regular expressions. *Turing machines* are automata augmented with a memory tape. They are one of the formal models describing the full power of computers.

Introduction activity: The armless and blind bartender

An armless and blind bartender likes to play a game with their clients. The rules are the following:

- Start the game with four glasses put in square on the bartender's tray. Each of them can either be right side up or upside down, at the will of the client.
- The bartender can never know which glasses are upside down, and which are not.
- One turn happens as follows:
  - The bartender choses one to four directions between up, down, left and right.
  - The client can rotate the trail however they want.
  - Then, the client must reverse the glasses in the direction(s) the bartender gave.
- The bartender wins if, at any time of the game, either the whole four glasses are right side up, or the whole four glasses are upside down. When it happens, the client must say it to the bartender, and the game stops.
- The aim of the client is to prevent the bartender from winning.

It is said that the bartender always win in a few turns. Do you think it is possible?

- If no, try your best to beat the bartender.
- If yes, find how!

1. Automata

1.1. Description of the machine

Automata is an abstract machine model designed to recognize patterns. This means that it takes as an input a string, reads it, and then answers wether the string conforms to the patterns the automaton must recognize or not.

It is the neurophysiologist Warren McCulloch and the logician studying neuroscience Walter Pitts who first published a paper describing a model similar to the automata described here, in 1943 [1].

What does an automaton look like?
An automaton can be depicted by a set of labelled *states*, usually circles, linked with labelled arrows. Those arrows are called *transitions*. There is also a state at which an arrow with no start is pointing, the *initial state*, and one or more states from which an arrow starts and points at nothing, the *final states*.

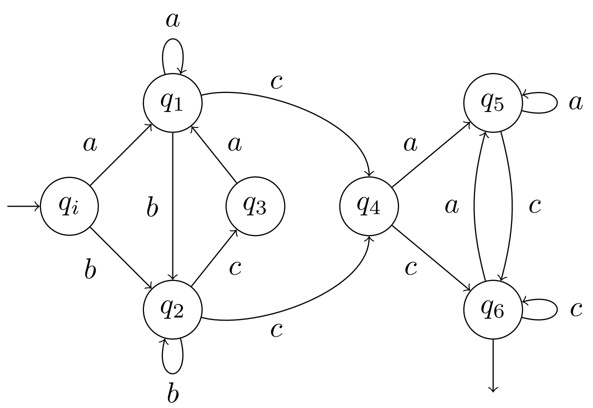


Figure 1: An automaton $\mathcal{A}_0$.

How to use an automaton? Automata are designed to read words, in order to either accept or reject them. When reading a word w, after each letter it reads, the automaton goes from its current state to another through a transition labelled with this letter. If there are none, it rejects the word it is reading. If after reading the whole word, the current state is a final state, the word is accepted. If not, it is rejected.

Example Consider the automaton $\mathcal{A}_0$ depicted in Figure 1.

- The word *bcabbcc* goes through the following states:

$$q_i \xrightarrow{b} q_2 \xrightarrow{c} q_3 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_2 \xrightarrow{c} q_4 \xrightarrow{c} q_6$$

  As q_6 is a final state, *bcabbcc* is *accepted* by $\mathcal{A}_0$.
- The word *bca* ends in q_1, which is not a final state. *bca* is thus *rejected* by $\mathcal{A}_0$.
- The word *bcbb* goes through the following states:

$$q_i \xrightarrow{b} q_2 \xrightarrow{c} q_3 \xrightarrow{b} \text{blocked}$$

  As it is blocked at some point when read by $\mathcal{A}_0$, it is *rejected* by $\mathcal{A}_0$.

Definition: Deterministic Finite Automaton

A **deterministic finite automaton** (DFA) is a tuple $\mathcal{A} = (Q, \Sigma, q_i, F, \delta)$ where:
- Q is finite set, whose elements are called **states** of $\mathcal{A}$.
- Σ is an alphabet.
- q_i is an element of Q, called the **initial state** of $\mathcal{A}$.
- F is a subset of Q, whose elements are called **final states** of $\mathcal{A}$.
- $\delta : Q \times \Sigma \to Q$ is a partial function, called the **transition function** of $\mathcal{A}$.

When δ is defined on each possible input, $\mathcal{A}$ is said to be **complete**.

The main interesting property to know when looking at an automaton is what words it accepts, and what words it does not. To mathematically define it, we first define a transition that describes the state a word can reach when read by an automaton.

Definition: Extended transition function

Let $\mathcal{A} = (Q, \Sigma, q_i, F, \delta)$ a deterministic finite automaton. The **extended transition function** of $\mathcal{A}$ is the function δ^* defined as follow:

$$\forall q \in Q, \quad \delta^*(q, \varepsilon) \overset{\text{def}}{=} q$$
$$\forall q \in Q, \forall l \in \Sigma, \forall w \in \Sigma^*, \quad \delta^*(q, lw) \overset{\text{def}}{=} \delta^*(\delta(q, l), w)$$

Then we can define the set of words an automaton accepts. Each word not in this set is rejected by the automaton.

Definition: Language of a deterministic finite automaton

Let $\mathcal{A} = (Q, \Sigma, q_i, F, \delta)$ a deterministic finite automaton. The **language of $\mathcal{A}$**, noted $\mathcal{L}(\mathcal{A})$, is defined as follow:

$$\mathcal{L}(\mathcal{A}) \overset{\text{def}}{=} \{w \in \Sigma^* \mid \delta^*(q_i, w) \in F\}$$

The language of $\mathcal{A}$ can also be called **accepted** or **recognized** by $\mathcal{A}$.

A word is **accepted** by $\mathcal{A}$ when it is in $\mathcal{L}(\mathcal{A})$, and **rejected** by $\mathcal{A}$ if not.

Definition: Recognizable language

Let Σ an alphabet. A language L on Σ is **recognizable** if there exist an automaton $\mathcal{A}$ on Σ that recognizes L. The set of all recognizable languages on Σ is denoted $\mathrm{Rec}(\Sigma)$.

Graphical representation Accepting states are sometimes depicted as a doubly circled state, rather than with a transition pointing at no state.

Example Consider the automaton $\mathcal{A}_1$ depicted in Figure 2. Its accepted language is

$$a(ba)^*a(aa(ba)^*a)^*$$

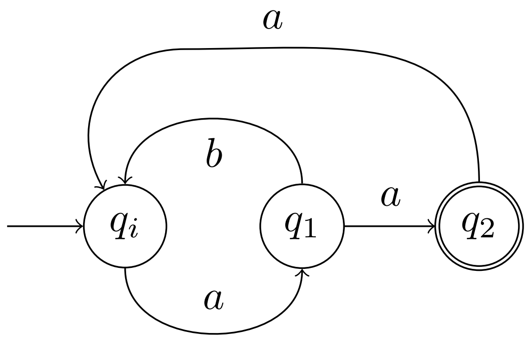


Figure 2: An automaton $\mathcal{A}_1$.

Why choosing alphabets with very few letters?

- Theorems and properties about automata are proven with any alphabet, so they can be used with more relevant alphabets when dealing with more concrete situations.
- Automata solving concrete situation are often huge and barely readable by humans. Example automata in an automaton course do not have to include thousands of states and hundreds of letters.

Exercise: Dessine-moi un ~~mouton~~ automate[1]

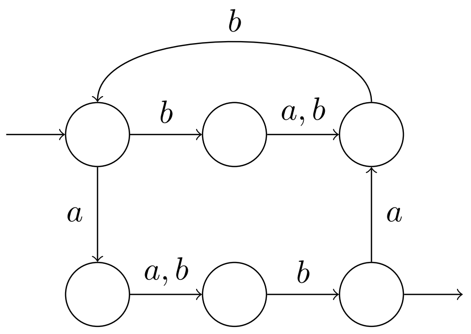
Draw an automaton whose language is:

1. the words on $\Sigma = \{a, b\}$ with exactly one a.
2. the words on $\Sigma = \{a, b\}$ where no b is surrounded by two a.
3. $((ab)^*(b \mid a)^*)^*$
4. $ab(bb \mid a)^*b(\varepsilon \mid ab^*)$

Be sure that your automaton does not accept words not in the language!

Exercise: From automata to languages

Describe the language recognized by the following automaton.



1.2. Automata and regular expressions

After designing an abstract computation model, one interesting question to ask is: what can this model achieve? How complex can its computation be? Can my problem be solved by this problem?

It is the american mathematician Stephen Cole Kleene who solved this problem for automata, in a paper published in 1956 [3]. One of his motivation was to know how deeply the model described in [1] can describe nervous activity.

It is him who invented the Kleene star, and described regular expressions, designed as a syntactic description of languages an automaton can recognize.

Theorem: Kleene's theorem

Let Σ and alphabet. $\mathrm{Rec}(\Sigma) = \mathrm{Reg}(\Sigma)$.

Kleene's theorem in natural language The equality in Kleene's theorem is a set equality, i.e. a double inclusion: $\mathrm{Rec}(\Sigma) \subseteq \mathrm{Reg}(\Sigma)$ and $\mathrm{Rec}(\Sigma) \supseteq \mathrm{Reg}(\Sigma)$. Explained with words rather than symbols:

1. Any language of an automaton can be described by a regular expression;
2. **And** any language of a regular expression can be recognized by an automaton.

It is important to clearly understand that those two properties are very different.

[1]In English: "Draw me a ~~sheep~~ automaton". French original quote from [2].

Kleene's theorem is a *huge* theorem, in that it elegantly describes the syntactic description of recognizable languages, and builds a two-way bridge between automata and regular expressions. One consequence of this bridge is that theorems holding for one side are instantly holding for the other side.

Exercise: Regular languages closures[2]

1. Let $\mathcal{A}$ an automaton. Show that there exist an automaton $\mathcal{A}'$ accepting the same language, and such that no word can be blocked when read by $\mathcal{A}'$. In other mathematical words: the transition function of $\mathcal{A}'$ is complete.
2. Let L a *recognizable* language on Σ. Show that the complement of L, the language $\Sigma^* \setminus L$, is also a recognizable language.
3. Let L a *regular* language. Show that the complement of L also is.
4. Let L_1 and L_2 two *regular* languages. Show that $L_1 \cap L_2$ also is.
   Hint: use closure under union and complement.

Regular expressions is an interesting set of languages, but not quite enough to model more complex languages. In particular, they can not recognize the set of balanced strings of brackets, and thus are not relevant for describing a programming language.

1.3. Nondeterministic finite automata

Automata are said *deterministic* when they only have one way to read each word. On some automata, two transitions going from the same state might have the same label, as in Figure 4. They are said *non deterministic*, because some words have more than one possible computation.

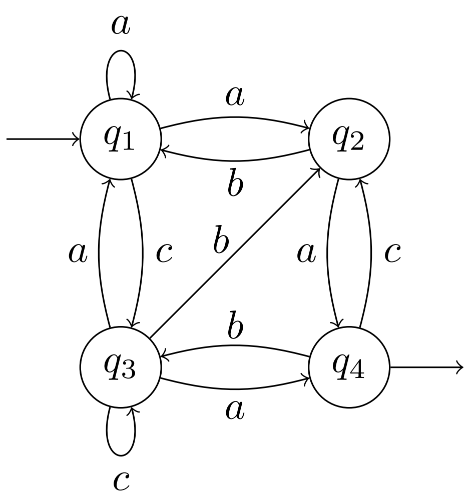


Figure 4: Automaton $\mathcal{A}_4$.

How to chose? In a deterministic automaton, how to chose between two different possible runs? e.g. in Figure 4, should aca reach q_4 and be accepted? Or should it be blocked in q_2?

We chose everything The standard model of nondeterministic automata sort of "choses everything". It accepts a word if and only if *there exist a run* accepting it. It rejects a word if and only if *all runs* are rejecting it.

[2]We say that a set is *closed under a certain operator* when using this operator on elements of this language can not "go outside" of the language. Example: Positive integers are closed under addition, but not subtraction.

You can see this model as, somehow, always choosing the best possible choice when performing a task.

Exercise: Non-deterministic automaton

Find a 5-letter word that is accepted by $\mathcal{A}_4$ in Figure 4. Find one that is rejected.

Definition: Nondeterministic Finite Automaton

A **nondeterministic finite automaton** (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ where:

- Q is finite set, whose elements are called **states** of $\mathcal{A}$.
- Σ is an alphabet.
- I is a subset of Q, whose elements are called **initial states** of $\mathcal{A}$.
- F is a subset of Q, whose elements are called **final states** of $\mathcal{A}$.
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is a partial function, called the **transition function** of $\mathcal{A}$.

Definition: Extended transition function

Let $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ a deterministic finite automaton. The **extended transition function** of $\mathcal{A}$ is the function δ^* defined as follow:

$$\forall S \in \mathcal{P}(Q), \qquad \delta^*(S, \varepsilon) \overset{\text{def}}{=} S$$

$$\forall S \in \mathcal{P}(Q), \forall l \in \Sigma, \forall w \in \Sigma^*, \quad \delta^*(S, lw) \overset{\text{def}}{=} \delta^*\left(\bigcup_{q \in S} \delta(q, l), w\right)$$

Definition: Language of a deterministic finite automaton

Let $\mathcal{A} = (Q, \Sigma, I, F, \delta)$ a nondeterministic finite automaton. The **language of $\mathcal{A}$**, noted $\mathcal{L}(\mathcal{A})$, is defined as follow:

$$\mathcal{L}(\mathcal{A}) \overset{\text{def}}{=} \{w \in \Sigma^* \mid \delta^*(I, w) \cap F \neq \emptyset\}$$

The language of $\mathcal{A}$ can also be called **accepted** or **recognized** by $\mathcal{A}$.

Exercise: Differences

Can you spot the differences between the definition of a DFA and an NFA?

Just as for deterministic automata, knowing what can exactly recognize nondeterministic automata is an interesting question. In particular, is this model powerful enough to recognize programming languages?

Theorem: Nondeterministic Finita Automata are not more powerful

Let L a language recognized by a nondeterministic finite automaton. Then there exist a deterministic finite automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$.

The automaton $\mathcal{A}_5$ depicted in Figure 5 is *deterministic*. It however has the same language as the automaton $\mathcal{A}_4$ depicted in Figure 4. Remember what it means:

- If a word w is accepted by $\mathcal{A}_4$, then it also is by $\mathcal{A}_5$.
- If a word w is accepted by $\mathcal{A}_5$, then it also is by $\mathcal{A}_4$.

Why using non-determinism? In implementations, non-determinism is always mimicked by determinism. On the fields of ideas however, drawing nondeterministic automata might be way more human-readable and compact than deterministic one. Just compare $\mathcal{A}_4$ and $\mathcal{A}_5$ to convince yourself!

Optional: Can you guess what the state labels of $\mathcal{A}_5$ mean?

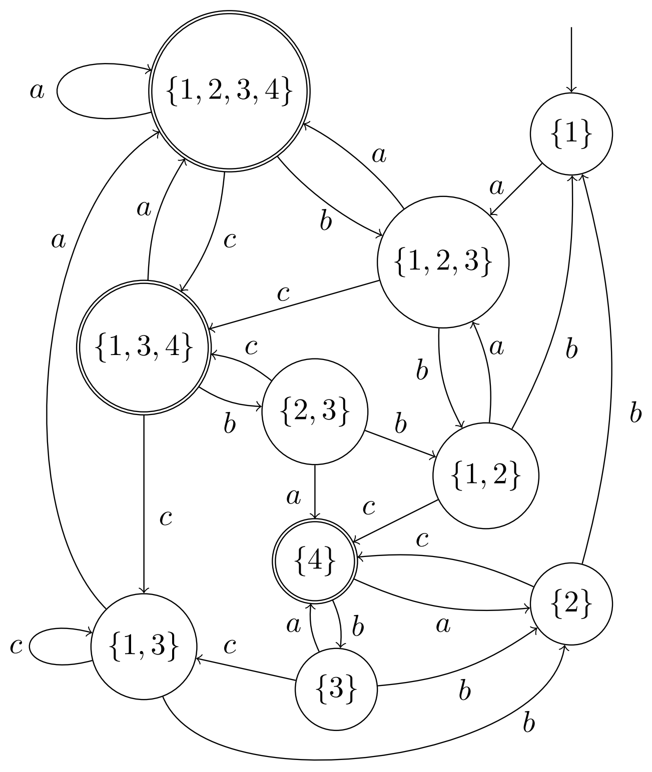


Figure 5: Deterministic automaton $\mathcal{A}_5$.

We will see in lecture 4 that non-determinism is also a way to classify problems.

2. Turing machines

Note: There are many variations on the Turing machine model. If you have already heard about "Turing machines" and what you know does not match what is written here, do not panic. The reason of the difference will be explained next course.

Why are automata not enough? The only memory automata have is implemented within their states. As they have a finite amount of states, we say that automata have a *bounded memory*. This is for example the intuitive reason why they can not recognize the language of balanced strings of brackets: this would need to *count* the amount of open brackets, which is not bounded.

Different kinds of memories Modern computers have a disk memory that is way larger than their RAM. The theoretical side of the RAM is the memory implemented within automata states. The theoretical model that implements the disk memory is called *Turing machines*. They have been invented by British mathematician Alan Turing in 1936 [4].

What are Turing machines? Informally, Turing machines are automata equipped with an infinite tape, which is used as a memory tape. The automata is moving on the tape, it can read it, and it can also write on it.

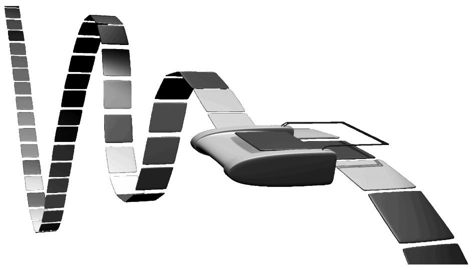


Figure 6: An artistic view of a Turing machine.

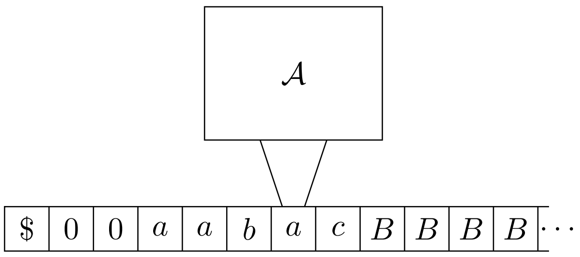


Figure 7: An illustration of a Turing machine closer to the formal definition.

How are automata augmented? Turing machine are automata that are very similar to deterministic finite automata, but with a few upgrades:

- They are *always* pointing at *exactly one* letter on the tape.
- In a DFA drawing, each transition is only labelled with one letter. In a Turing machine drawing, each transition is labelled with three symbols:
  - One first letter, whose use is the same as in DFAs: it is the letter that is read.
  - One second letter: it is the letter that is written on the current position.
  - One arrow: it describes in which direction the automaton should move on the tape *after* writing the letter on its current position.
- Among the states of the machine, there is *always* one `accept` and one `reject` state. When either is met, the computation stops.

The three allowed directions are $\rightarrow$, $\leftarrow$ and $\downarrow$. “Not writing” is possible by writing the letter that has just been read.

What does the tape look like? The memory tape is an infinite tape with one end. Its first character is a special \$ character: it can never be removed, and when the machine reads it, it **must** move to the right. After the \$ character, there are some letters of a *working alphabet* Σ, and then infinitely many *blank characters* B.

Definition: Turing machine

A **Turing machine** is a tuple (Q, q_i, Σ, δ) where:

- Q is a set of **states**. It must contain two special states `accept` and `reject`.
- $q_i \in Q$ is the **initial state**.
- Σ is the **(working) alphabet** of the machine. It must contain two special characters \$ and B.
- $\delta : Q \times \Sigma \to \Sigma \times \{\leftarrow, \rightarrow, \downarrow\} \times Q$ is the **transition function**. It must ensure:

$$\forall q \in Q, \exists q' \in Q, \quad \delta(q, \$) = (\$, \rightarrow, q')$$

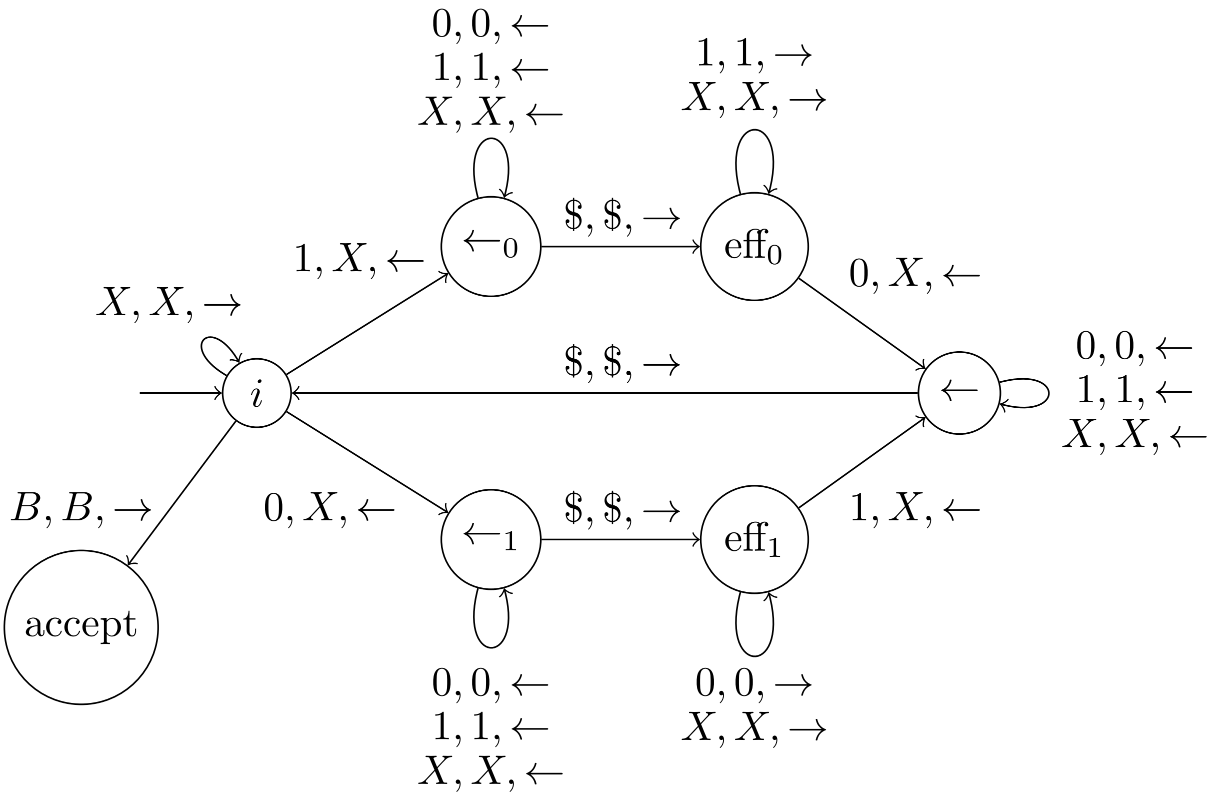


Figure 8: A Turing machine $\mathcal{M}_1$.
All non depicted transitions lead to the `reject` state.

How to read and accept a word? A machine always starts a computation on the character just at the right of \$. Testing wether a machine $\mathcal{M}$ recognizes a word w or not is achieved by starting its computation on a tape where w is written right after \$ on the tape, and there are only blank characters after it. If $\mathcal{M}$ ends its computation on its `accept` state, then w is accepted. If not, it is rejected.

The alphabet inside the machine definition is sometimes called a *working* alphabet if it uses some more characters that never occur in the word it reads.

Exercise: Reading a word with a Turing machine

Consider the Turing machine depicted in Figure 8. Draw the successive tapes and current state of the automata when reading the word

10

then when reading the word

001

Optional: What does this machine recognize?

The aim of the following definitions is to formalize the notion of reading a word with a Turing machine.

Definition: Turing machine configuration

A **configuration** of a Turing machine (Q, q_i, Σ, δ) is a tuple

$$(w, q, w') \in \Sigma^* \times Q \times \Sigma^*$$

where:

- w represents what is written on the tape on the left of the pointed cell.
- q is the current state the machine is in.
- the first letter of w' represents the content of the cell pointed by the machine. The next letters represent the content of the tape between this cell and the start of the infinitely many blank characters.

Example Assuming the current state of the automaton in Figure 7 is q, the configuration depicted is

$$(\$00aab, q, ac).$$

Definition: Turing machine computation

A **computation** of a Turing machine (Q, q_i, Σ, δ) is a sequence of configurations

$$c_0 \to c_1 \to c_2 \to \dots$$

such that each pair of successive configurations (w_n, q, w'_n) and (w_{n+1}, q', w'_{n+1}) have the following properties:

- Write $w'_n = lv'_n$, meaning w'_n has first letter l, and following letters v'_n.
- $\delta(q, l)$ third coordinate must be q'.
- If $\delta(q, l) = (l', \downarrow, q')$, then the following equalities hold:
  - $w_{n+1} = w_n$, and
  - $w'_{n+1} = l'v'_n$.
- If $\delta(q, l) = (l', \leftarrow, q')$, then write $w_n = v_n m$, meaning w_n has last letter m, and first letters v_n. The following equalities hold:
  - $w_{n+1} = v_n$, and
  - $w'_{n+1} = l'w'_n$.
- If $\delta(q, l) = (l', \rightarrow, q')$, then the following equalities hold:
  - $w_{n+1} = w_n l'$, and
  - $w'_{n+1} = v'_n$.
- If c_n has a state $q = \texttt{accept}$ or $q = \texttt{reject}$, then it is the last configuration of the sequence.

When w'_n is empty, consider instead $w'_n = B$.

Take some time to understand the link between those formal definitions and previous explanations in English.

Definition: Language of a Turing machine

The **language of a Turing machine** $\mathcal{M}$ is the set of words w such that a computation of $\mathcal{M}$ on an initial configuration $c_0 = (w, q_i, \varepsilon)$ ends with a configuration whose state is `accept`.

Definition: Computable language

A language L is a **computable language** if there exist a Turing machine $\mathcal{M}$ such that L is the language of $\mathcal{M}$.

What can Turing machines recognize? Turing machines can be considered as usual computers, with an infinite memory tape. They can encode `for` or `while` loops, use conditional branching, or store in variables. As such, they can recognize whatever your favorite programming language can.

Because of this, and because writing a Turing machine is a very tedious task, it is often pseudo-code with basic operations that is written instead of Turing machines.

Example: The set of balanced strings of brackets can be recognized by the pseudo-code opposite. As such, it can be recognized by a Turing machine.

```
is_balanced(w):
    nb_open <- 0
    for i in [1, |w|]:
        if w[i] = "("
            nb_open += 1
        else:
            if nb_open = 0:
                return false
            nb_open -= 1
    return (nb_open = 0)
```

Can Turing machines compute functions, rather than just recognizing languages? Yes they can. And, just as our real computers, they can also have bugs. One simple example is a computation of a Turing machine that does not end, which is the theoretical counterpart of a `while` loop that does not end.

Exercise: Forever looping

Find a Turing machine M that never ends its computation.

Bibliography

[1] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943, doi: 10.1007/BF02478259.

[2] A. de Saint-Exupéry, *Le Petit Prince.* 1943.

[3] S. Kleene, "Representation of Events in Nerve Nets and Finite Automata," *Automata Studies: Annals of Mathematics Studies. Number 34*, no. 34, p. 3–4, 1956, doi: 10.1515/9781400882618-002.

[4] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, no. 1, pp. 230–265, 1937, doi: 10.1112/plms/s2-42.1.230.