

# Theory of computation: initial remarks

For many purposes, computation is elegantly modeled with simple mathematical objects:

Turing machines, finite automata, pushdown automata, and such.

Turing machines (TMs) make precise the otherwise vague notion of an “algorithm”: no more powerful precise account of algorithms has been found. (Church-Turing Thesis)

An easy consequence of the definition of TMs: most functions (say, from  $\mathcal{N}$  to  $\mathcal{N}$ ) are not computable (by TM-equivalent machines).

Why? There are more functions than there are TMs to compute them.

Remarkably, Turing machines also serve as a common platform for characterizing computational complexity, since the time cost, and space cost, of working with a TM is within a polynomial of the performance of conventional computers.

So, for instance, the most famous open question in computer science theory —

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

— while formulated in terms of TMs, is important for practical computing.

Undergraduate study of the theory of computation typically starts with finite automata. They are simpler than TMs, and it is easier to prove main results about them. They are also intuitively appealing. And automata theory has wide practical importance, in compiler theory and implementation, for instance. (lex – lexical analysis)

The typical next step is to study grammars, which are also nicely intuitive, and already somewhat familiar to students. The theory of context-free grammars is central to compiler theory and implementation. (yacc – compiler generation)

Turing machines are only a little more complicated, but the theory of computation based on them is quite sophisticated and extensive.

One thing all these models have in common is that they formulate computation in terms of questions about *languages*.

In fact, many of the central questions are formulated in terms of **language recognition**: deciding whether a given string belongs to a given language.

# Regular languages, regular expressions and finite automata

“Regular” languages are relatively simple languages.

We'll study means for “generating” regular languages and also for “recognizing” them.

All finite languages are regular.

Some infinite languages are regular.

Each regular language can be characterized by a (finite!) regular expression: which says how to generate the strings in the language.

It can also be characterized by a (finite!) finite state automaton: which provides a mechanism for recognizing the strings in the language.

# Regular languages

A language over an alphabet  $\Sigma$  is *regular* if it can be constructed from the empty language, the language  $\{\Lambda\}$  and the singleton languages  $\{a\}$  ( $a \in \Sigma$ ) by a finite number of applications of union, language product and Kleene star.

Sounds like an inductively-defined set. . .

The set of *regular languages* over an alphabet  $\Sigma$  is the least set of languages over  $\Sigma$  s.t.

1.  $\emptyset$  is a regular language,
2.  $\{\Lambda\}$  is a regular language,
3. For all  $a \in \Sigma$ ,  $\{a\}$  is a regular language,
4. If  $A$  is a regular language, so is  $A^*$ ,
5. If  $A$  and  $B$  are regular languages, so are  $A \cup B$  and  $AB$ .

Here, conditions 1–3 are the *basis* part of the inductive definition, and conditions 4 & 5 are the *induction* part.

The set of *regular languages* over an alphabet  $\Sigma$  is the least set of languages over  $\Sigma$  s.t.

1.  $\emptyset$  is a regular language,
  2.  $\{\Lambda\}$  is a regular language,
  3. For all  $a \in \Sigma$ ,  $\{a\}$  is a regular language,
  4. If  $A$  is a regular language, so is  $A^*$ ,
  5. If  $A$  and  $B$  are regular languages, so are  $A \cup B$  and  $AB$ .
- 

**Example**  $\{00, 01, 10, 11\}^*$  is a regular language.

What is a “nice” description of this language?

Let's check that it is regular:

$L_1 = \{0\}$  and  $L_2 = \{1\}$  are regular.

Hence,  $L_1L_1 = \{00\}$ ,  $L_1L_2 = \{01\}$ ,  $L_2L_1 = \{10\}$  and  $L_2L_2 = \{11\}$  are regular.

It follows that  $\{00\} \cup \{01\} = \{00, 01\}$  is regular,  
as are  $\{00, 01\} \cup \{10\} = \{00, 01, 10\}$  and  
 $\{00, 01, 10\} \cup \{11\} = \{00, 01, 10, 11\}$ .

So  $\{00, 01, 10, 11\}^*$  is regular.

## Regular expressions

**Definition** The set of *regular expressions* over an alphabet  $\Sigma$  is the least set of strings satisfying the following 5 conditions.

1.  $\emptyset$  is a regular expression, and stands for the language  $\emptyset$ .
2.  $\Lambda$  is a regular expression, and stands for the language  $\{\Lambda\}$ .
3. For all  $a \in \Sigma$ ,  $a$  is a regular expression, and stands for the language  $\{a\}$ .
4. If  $r$  is a regular expression that stands for the language  $A$ , then  $r^*$  is a regular expression standing for the language  $A^*$ .
5. If  $r_1$  and  $r_2$  are regular expressions that stand for the languages  $A$  and  $B$  respectively, then  $(r_1 + r_2)$  and  $(r_1 r_2)$  are regular expressions standing for the languages  $A \cup B$  and  $AB$ , respectively.

For example

$$((((00) + (01)) + (10)) + (11))^*$$

is a regular expression that stands for the language

$$\{00, 01, 10, 11\}^* .$$

## We can omit many of the parentheses

We can omit many of the parentheses in regular expressions.

For instance, outermost parentheses can be safely dropped. So

$$((a + b) + c) = (a + b) + c .$$

*Notice that when we write “=” here, we are saying that the expressions stand for the same language (not that the two expressions are identical viewed as strings).*

Product and union are associative, and accordingly

$$(a + b) + c = a + (b + c) = a + b + c$$

and

$$(ab)c = a(bc) = abc .$$

## Precedence conventions help

Under the convention that the Kleene closure operator has the highest precedence, + the lowest, with language product in between, we also have

$$\begin{aligned}ab^* + c^*d &= (ab^*) + (c^*d) \\ab^* + c^*d &\neq (ab)^* + c^*d \\ab^* + c^*d &\neq a(b^* + c^*)d\end{aligned}$$

So the regular expression

$$(((00) + (01)) + (10)) + (11))^*$$

can be written

$$(00 + 01 + 10 + 11)^* .$$

Another regular expression for this language is

$$((0 + 1)(0 + 1))^* .$$

## Exponent notation is also convenient

We can also use exponent notation, much as before. Hence,

$$(a + b)(a + b) = (a + b)^2$$

and so forth. Of course, for every regular expression  $r$

$$r^0 = \Lambda.$$

(Recall that the regular expression  $\Lambda$  stands for the language  $\{\Lambda\}$ !)

As with the Kleene star, the precedence of exponentiation is higher than that of product and union, so

$$ab^2 = abb$$

and

$$a + b^2 = a + bb.$$

**Example** Find a binary string of minimal length among those *not* in the language characterized by the regular expression:

$$(0^* + 1^*)^2$$

$$1^*(01)^*0^*$$

## Recognizing a regular language

For regular languages, we address the language recognition problem roughly as follows. To decide whether a given string is in the language of interest. . .

- ▶ Allow a single pass over the string, left to right.
- ▶ Rather than waiting until the end of the string to make a decision, we make a tentative decision at each step. That is, for each prefix we decide whether it is in the language.

Question: How much do we need to remember at each step about what we have seen previously?

Everything? (If so we're in trouble — our memory is finite, and strings can be arbitrarily long.)

Nothing? (Fine for  $\emptyset$  and  $\Sigma^*$ .)

In general, we can expect there to be strings  $x, y$  s.t.  $x \in L$  and  $y \notin L$ . As we read these strings from left to right, we will need to remember enough to allow us to distinguish  $x$  from  $y$ .

We'll eventually give a “perfect” account of what must be remembered at each step — in terms of so-called “distinguishability”. First, let's define finite automata and get a feeling for them. . .

# Deterministic finite automata

**Definition** A *deterministic finite automaton* is a five-tuple

$$M = (S, \Sigma, T, s_0, F)$$

where

- ▶  $S$  is a finite set of “states”,
- ▶  $\Sigma$  is an alphabet — the “input alphabet”,
- ▶  $T : S \times \Sigma \rightarrow S$  is the “transition function”,
- ▶  $s_0 \in S$  is the “initial state”,
- ▶  $F \subset S$  is the set of “final” or “accepting” states.

**Example** Before we work out all the details of this definition, let's look at a diagram of a DFA that recognizes the language

$$\{x \in \{0, 1\}^* \mid x \text{ has an even number of 1's}\}.$$

A DFA  $M = (S, \Sigma, T, s_0, F)$ , where

- ▶  $S$  is a finite set of “states”,
  - ▶  $\Sigma$  is an alphabet — the “input alphabet”,
  - ▶  $T : S \times \Sigma \rightarrow S$  is the “transition function”,
  - ▶  $s_0 \in S$  is the “initial state”,
  - ▶  $F \subset S$  is the set of “final” or “accepting” states.
- 

Intuitively, a DFA “remembers” (and “decides” about) the prefix it has read so far by changing state as it reads the input string.

Initially, (i.e. after having read the empty prefix of the input string), the DFA is in state  $s_0$ .

The DFA  $M$  “accepts” the empty string iff

$$s_0 \in F.$$

If  $a \in \Sigma$  is the first character in the input string, then the state of  $M$  after reading that first character is given by  $T(s_0, a)$ .

$M$  “accepts” the string  $a$  iff

$$T(s_0, a) \in F.$$

A DFA  $M = (S, \Sigma, T, s_0, F)$ , where

- ▶  $S$  is a finite set of “states”,
  - ▶  $\Sigma$  is an alphabet — the “input alphabet”,
  - ▶  $T : S \times \Sigma \rightarrow S$  is the “transition function”,
  - ▶  $s_0 \in S$  is the “initial state”,
  - ▶  $F \subset S$  is the set of “final” or “accepting” states.
- 

Similarly, if  $a \in \Sigma$  is the first character in the input string and  $b \in \Sigma$  is the second, then the state of  $M$  after reading that second character is given by  $T(T(s_0, a), b)$ .

$M$  “accepts” the string  $ab$  iff

$$T(T(s_0, a), b) \in F.$$

This notation grows cumbersome as the input string grows longer.

We’d like a more convenient way to describe the state that DFA  $M$  is in after reading an input string  $x \dots$

Defining  $T^* : S \times \Sigma^* \rightarrow S$  recursively in terms of  $T : S \times \Sigma \rightarrow S$

The transition function  $T$  takes a state  $s$  and an input symbol  $a$  and returns the resulting state

$$T(s, a).$$

Given  $T$ , we recursively define the function  $T^*$  that takes a state  $s$  and an input string  $x$  and returns the resulting state

$$T^*(s, x).$$

**Definition** Given a DFA  $M = (S, \Sigma, T, s_0, F)$ , we define the multi-step transition function

$$T^* : S \times \Sigma^* \rightarrow S$$

as follows:

1. For any  $s \in S$ ,  $T^*(s, \Lambda) = s$ ,
2. For any  $s \in S$ ,  $x \in \Sigma^*$  and  $a \in \Sigma$ ,

$$T^*(s, xa) = T(T^*(s, x), a).$$

You may notice that, implicitly, we seem to have in mind a different inductive definition of  $\Sigma^*$  than we used previously...

## “Backwards” inductive definition of $\Sigma^*$

It turns out to be more convenient for our purposes to use the following alternative inductive definition of  $\Sigma^*$ .

- ▶ *Basis:*  $\Lambda \in \Sigma^*$ .
- ▶ *Induction:* If  $x \in \Sigma^*$  and  $a \in \Sigma$ , then  $xa \in \Sigma^*$ .

Previously we used an inductive definition of  $\Sigma^*$  in which  $x$  and  $a$  were reversed (in the induction part).

That was in keeping with what is most convenient when working with lists. (Intuitively speaking, you build a list by adding elements to the front of the list.)

From now on, we will use this new “backwards” definition.

It seems reasonably intuitive for strings. (You write a string from left to right, right?)

It is clearly equivalent to the other inductive definition.

It is much more convenient for proving things about finite automata!

## First sanity check for recursive defn of $T^*$

Recall: For DFA  $M = (S, \Sigma, T, s_0, F)$ , we define

$$T^* : S \times \Sigma^* \rightarrow S$$

as follows:

1. For any  $s \in S$ ,  $T^*(s, \Lambda) = s$ ,
2. For any  $s \in S$ ,  $x \in \Sigma^*$  and  $a \in \Sigma$ ,

$$T^*(s, xa) = T(T^*(s, x), a).$$

Here's a property  $T^*$  should have...

**Claim** For any DFA  $M = (S, \Sigma, T, s_0, F)$ , and any  $s \in S$  and  $a \in \Sigma$ ,

$$T^*(s, a) = T(s, a).$$

This is easy to check...

$$\begin{aligned} T^*(s, a) &= T^*(s, \Lambda a) \\ &= T(T^*(s, \Lambda), a) && \text{(defn } T^*) \\ &= T(s, a) && \text{(defn } T^*) \end{aligned}$$

$$T^* : S \times \Sigma^* \rightarrow S$$

$$T^*(s, \Lambda) = s, \text{ for all } s \in S$$

$$T^*(s, xa) = T(T^*(s, x), a), \text{ for all } s \in S, x \in \Sigma^*, \text{ and } a \in \Sigma$$

---

Here's another sanity check on the definition of  $T^*$ : Roughly, does “reading  $xy$ ” take you to the same state as “reading  $x$  and then reading  $y$ ”?

**Claim** For any DFA  $M = (S, \Sigma, T, s_0, F)$ , any  $s \in S$ , and  $x, y \in \Sigma^*$ ,

$$T^*(s, xy) = T^*(T^*(s, x), y).$$

*Proof.*

By structural induction on  $y$  (using the “backwards” inductive defn of  $\Sigma^*$ ).

*Basis:*

$$\begin{aligned} T^*(s, x\Lambda) &= T^*(s, x) \\ &= T^*(T^*(s, x), \Lambda) \quad (\text{defn } T^*, T^*(s, x) \in S) \end{aligned}$$

*Induction:*  $y \in \Sigma^*$ ,  $a \in \Sigma$ .

IH:  $T^*(s, xy) = T^*(T^*(s, x), y)$ .

NTS:  $T^*(s, xya) = T^*(T^*(s, x), ya)$ .

$$\begin{aligned} T^*(s, xya) &= T(T^*(s, xy), a) && (\text{defn } T^*) \\ &= T(T^*(T^*(s, x), y), a) && (\text{IH}) \\ &= T^*(T^*(s, x), ya) && (\text{defn } T^*) \end{aligned}$$

## The language $L(M)$ recognized by an DFA $M$

Now that we have adequate notation for the state of a DFA after it reads an input string, we can define when a DFA “accepts” a string. . .

**Definition** For any DFA  $M = (S, \Sigma, T, s_0, F)$ , a string  $x \in \Sigma^*$  is *accepted* by  $M$  if

$$T^*(s_0, x) \in F.$$

The language *recognized* (or *accepted*) by  $M$ , denoted  $L(M)$ , is the set of strings accepted by  $M$ .

That is,

$$L(M) = \{ x \in \Sigma^* \mid T^*(s_0, x) \in F \}.$$

Notice that even if  $L \subset L(M)$  we still do not say that  $L$  is recognized by  $M$  unless  $L = L(M)$ .

(That is, if DFA  $M$  accepts language  $L$ , then not only does  $M$  accept *all* strings from  $L$  — it also accepts *only* those strings!)

# Kleene's Theorem

Recall that the regular languages over alphabet  $\Sigma$  are exactly those that can be constructed (in a finite number of steps) using (binary) union, language product and Kleene closure, starting from the languages  $\emptyset$ ,  $\{\Lambda\}$ , and  $\{a\}$  for each  $a \in \Sigma$ .

Here is the remarkable theorem characterizing regular languages in terms of DFA's:

**Kleene's Theorem** A language is regular iff some DFA recognizes it.

## DFA diagrams

It is convenient to represent a DFA as a diagram, as we have seen. . .

Recall: A DFA is a 5-tuple  $(S, \Sigma, T, s_0, F)$ .

For every state  $s \in S$  there is a corresponding node, represented by the symbol  $s$  with a circle around it:

We indicate the initial state by an incoming arrow (with no “source”):

We indicate that  $s \in F$  by adding a concentric circle:

For any  $q, r \in S$  and  $a \in \Sigma$ ,  
there is a directed edge labeled  $a$  from  $q$  to  $r$  if  $T(q, a) = r$ :

Let's draw a few DFA's. . .

## A DFA for $01^+$

We can write  $r^+$  (where  $r$  is a regular expression) to stand for  $rr^*$ .

(Recall: If  $L$  is a language, we write  $L^+$  to stand for  $LL^*$ .)

A DFA for  $((0 + 1)(0 + 1))^*$

A DFA for  $(0 + 1)^*10$

A DFA for  $0^*10^*(10^*10^*)^*$