

## Week1 wednesday

Our motivation in studying sets of strings is that they encode problems.

We need to describe the collection of all strings that match the pattern or property of a problem.

Let's start by thinking about how we can describe a language (a set of strings from a given alphabet).

**Definition 1.52:** A **regular expression** over alphabet  $\Sigma$  is a syntactic expression that can describe a language over  $\Sigma$ . The collection of all regular expressions is defined recursively:

*Basis steps of recursive definition*

$a$  is a regular expression, for  $a \in \Sigma$

$\varepsilon$  is a regular expression

$\emptyset$  is a regular expression

*Recursive steps of recursive definition*

$(R_1 \cup R_2)$  is a regular expression when  $R_1, R_2$  are regular expressions

$(R_1 \circ R_2)$  is a regular expression when  $R_1, R_2$  are regular expressions

$(R_1^*)$  is a regular expression when  $R_1$  is a regular expression

The *semantics* (or meaning) of the syntactic regular expression is the **language described by the regular expression**. The function that assigns a language to a regular expression over  $\Sigma$  is defined recursively, using familiar set operations:

*Basis steps of recursive definition*

The language described by  $a$ , for  $a \in \Sigma$ , is  $\{a\}$  and we write  $L(a) = \{a\}$

The language described by  $\varepsilon$  is  $\{\varepsilon\}$  and we write  $L(\varepsilon) = \{\varepsilon\}$

The language described by  $\emptyset$  is  $\{\}$  and we write  $L(\emptyset) = \emptyset$ .

*Recursive steps of recursive definition*

When  $R_1, R_2$  are regular expressions, the language described by the regular expression  $(R_1 \cup R_2)$  is the union of the languages described by  $R_1$  and  $R_2$ , and we write

$$L( (R_1 \cup R_2) ) = L(R_1) \cup L(R_2) = \{w \mid w \in L(R_1) \vee w \in L(R_2)\}$$

When  $R_1, R_2$  are regular expressions, the language described by the regular expression  $(R_1 \circ R_2)$  is the concatenation of the languages described by  $R_1$  and  $R_2$ , and we write

$$L( (R_1 \circ R_2) ) = L(R_1) \circ L(R_2) = \{uv \mid u \in L(R_1) \wedge v \in L(R_2)\}$$

When  $R_1$  is a regular expression, the language described by the regular expression  $(R_1^*)$  is the **Kleene star** of the language described by  $R_1$  and we write

$$L( (R_1^*) ) = ( L(R_1) )^* = \{w_1 \cdots w_k \mid k \geq 0 \text{ and each } w_i \in L(R_1)\}$$

For the following examples assume the alphabet is  $\Sigma_1 = \{0, 1\}$ :

The language described by the regular expression 0 is  $L(0) = \{0\}$

The language described by the regular expression 1 is  $L(1) = \{1\}$

The language described by the regular expression  $\varepsilon$  is  $L(\varepsilon) = \{\varepsilon\}$

The language described by the regular expression  $\emptyset$  is  $L(\emptyset) = \emptyset$

The language described by the regular expression  $((0 \cup 1) \cup 1)$  is  $L( ((0 \cup 1) \cup 1) ) =$

The language described by the regular expression  $1^+$  is  $L( (1)^+ ) =$

The language described by the regular expression  $\Sigma_1^*1$  is  $L( \Sigma_1^*1 ) =$

The language described by the regular expression  $(\Sigma_1 \Sigma_1 \Sigma_1 \Sigma_1 \Sigma_1)^*$  is  $L( (\Sigma_1 \Sigma_1 \Sigma_1 \Sigma_1 \Sigma_1)^* ) =$

A regular expression that describes the language  $\{00, 01, 10, 11\}$  is

A regular expression that describes the language  $\{0^n 1 \mid n \text{ is even}\}$  is

*Shorthand and conventions*

Assuming $\Sigma$ is the alphabet, we use the following conventions	
$\Sigma$	regular expression describing language consisting of all strings of length 1 over $\Sigma$
$*$ then $\circ$ then $\cup$	precedence order, unless parentheses are used to change it
$R_1 R_2$	shorthand for $R_1 \circ R_2$ (concatenation symbol is implicit)
$R^+$	shorthand for $R^* \circ R$
$R^k$	shorthand for $R$ concatenated with itself $k$ times, where $k$ is a natural number
Pages 63 - 65	

**Caution:** many programming languages that support regular expressions build in functionality that is more powerful than the “pure” definition of regular expressions given here.

Regular expressions are everywhere (once you start looking for them).

Software tools and languages often have built-in support for regular expressions to describe **patterns** that we want to match (e.g. Excel/ Sheets, grep, Perl, python, Java, Ruby).

Under the hood, the first phase of **compilers** is to transform the strings we write in code to tokens (keywords, operators, identifiers, literals). Compilers use regular expressions to describe the sets of strings that can be used for each token type.

Next time: we’ll start to see how to build machines that decide whether strings match the pattern described by a regular expression.

*Extra examples for practice:*

Which regular expression(s) below describe a language that includes the string  $a$  as an element?

$a^*b^*$

$a(ba)^*b$

$a^* \cup b^*$

$(aaa)^*$

$(\varepsilon \cup a)b$

# Week1 friday

**Review:** Determine whether each statement below about regular expressions over the alphabet  $\{a, b, c\}$  is true or false:

True or False:  $a \in L( (a \cup b) \cup c )$

True or False:  $ab \in L( (a \cup b)^* )$

True or False:  $ba \in L( a^*b^* )$

True or False:  $\varepsilon \in L(a \cup b \cup c)$

True or False:  $\varepsilon \in L( (a \cup b)^* )$

True or False:  $\varepsilon \in L( a^*b^* )$

**From the pre-class reading, pages 34-36:** A deterministic finite automaton (DFA) is specified by  $M = (Q, \Sigma, \delta, q_0, F)$ . This 5-tuple is called the **formal definition** of the DFA. The DFA can also be represented by its state diagram: with nodes for the state, labelled edges specifying the transition function, and decorations on nodes denoting the start and accept states.

Finite set of states  $Q$  can be labelled by any collection of distinct names. Often we use default state labels  $q_0, q_1, \dots$

The alphabet  $\Sigma$  determines the possible inputs to the automaton. Each input to the automaton is a string over  $\Sigma$ , and the automaton “processes” the input one symbol (or character) at a time.

The transition function  $\delta$  gives the next state of the DFA based on the current state of the machine and on the next input symbol.

The start state  $q_0$  is an element of  $Q$ . Each computation of the machine starts at the start state.

The accept (final) states  $F$  form a subset of the states of the DFA,  $F \subseteq Q$ . These states are used to flag if the machine accepts or rejects an input string.

The computation of a machine on an input string is a sequence of states in the machine, starting with the start state, determined by transitions of the machine as it reads successive input symbols.

The DFA  $M$  accepts the given input string exactly when the computation of  $M$  on the input string ends in an accept state.  $M$  rejects the given input string exactly when the computation of  $M$  on the input string ends in a nonaccept state, that is, a state that is not in  $F$ .

The language of  $M$ ,  $L(M)$ , is defined as the set of all strings that are each accepted by the machine  $M$ . Each string that is rejected by  $M$  is not in  $L(M)$ . The language of  $M$  is also called the language recognized by  $M$ .

What is **finite** about all deterministic finite automata? (Select all that apply)

- ☐ The size of the machine (number of states, number of arrows)
- ☐ The number of strings that are accepted by the machine
- ☐ The length of each computation of the machine



The formal definition of this DFA is

Classify each string  $a, aa, ab, ba, bb, \varepsilon$  as accepted by the DFA or rejected by the DFA.

*Why are these the only two options?*

The language recognized by this DFA is



The language recognized by this DFA is



The language recognized by this DFA is