Chapter 9

Automata and Regular Languages

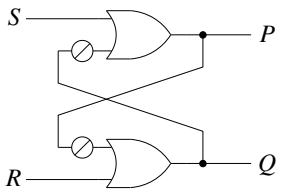
9.1 Introduction

This chapter looks at mathematical models of computation and languages that describe them. The model-language relationship has multiple levels. We shall explore the simplest level, and briefly describe the others at the end.

First, we take a brief look at the foundation in *sequential* digital circuits. A new element is added that can "hold" bits. This capability to capture and remember values greatly increases what digital systems are capable of doing.

9.1.1 Finite State Machines*

The digital circuit shown to the left below is called a *flip-flop*. The $\oslash$ symbol stands for *not*. The circuit is described by the *simultaneous* system of boolean equations in the center, whose truth table is on the right.



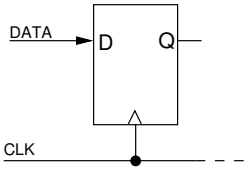
$$\begin{aligned} P &= S + \overline{Q} \\ Q &= R + \overline{P} \end{aligned}$$

S	R	P	Q
0	0	1	0
		0	1
0	1	1	0
1	0	0	1
1	1	1	1

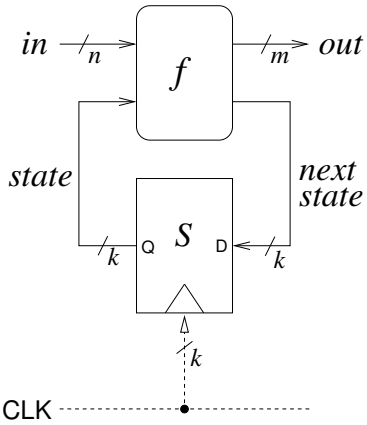
In the case that $R = S = 0$ the system reduces to $\begin{aligned} P &= \overline{Q} \\ Q &= \overline{P} \end{aligned}$, which has two solutions as the truth table shows. A physical circuit must somehow select one of these solutions, and it does so by selecting the solution that was present in the previous instant. In other words, the flip-flop will hold P's and Q's values

indefinitely, until it is forced to change by a pulse on S, making $P = 1$ and $Q = 0$, or a pulse on R, making $P = 0$ and $Q = 1$.

Simultaneous pulses on P and Q can force the flip-flop into a *metastable* solution, $P \approx Q$, but eventually, it will "snap" to a valid state. Additional circuitry is added to minimize this vulnerability. A *clocked flip-flop* has one input presenting the bit to be held. The CLK input is used to control *when* the next bit is captured.



We can think of any digitial system as an instance of a *finite-state machine*

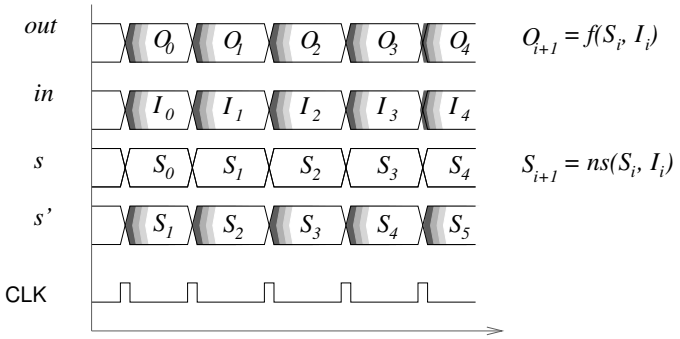


In this architecture,

- Block f represents a *purely combinational* boolean system of $n + k$ inputs and $m + k$ outputs. "Combinational" that each output of f is just a boolean combination of some or all of the $n + k$ inputs.

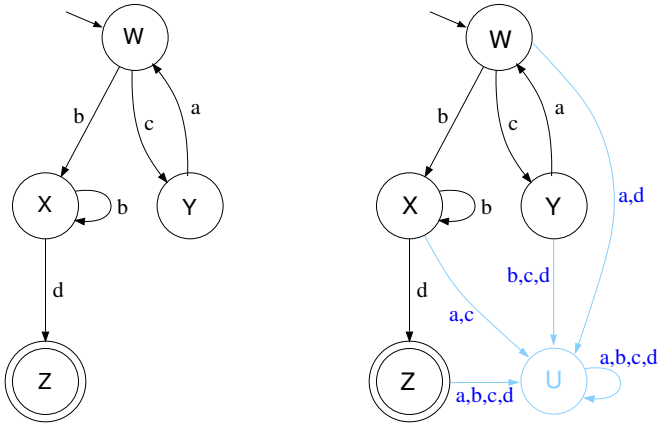
- Block S represents an array of simple *storage elements*, each capable of storing one bit, 0 or 1, indefinitely. Think of the signal CLK as a kind of electronic metronome, going *tick-tock-tick-tock-*$\cdots$. On *tick*, the k one-bit storage elements simultaneously capture the value presented on its input and holds it until the next *tick*.

- The period between CLK ticks is long enough to allow the circuitry to electronically stabilize, so that there are no "race" conditions where an input is changing when the tick occurs.



9.2 Automata

A *finite-state automoton* (FA) models the actions of an idealized computing machine. Before formalizing this concept, let us look at an example depicted by the diagram



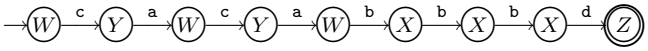
The nodes of the FA form a graph whose edges are labelled by letters of an alphabet. The nodes $S = \{W, X, Y, Z\}$ are called the *states* of the FA and the alphabet is $A = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}$.

- The FA starts in state $s = W$, as indicated by the arrow. It is presented with an input word, $w \in A^*$ which it consumes one letter at a time.
- If there is an edge (s, s') labelled with the next input letter, the FA makes a *transition* into state s' consuming the letter.
- The process of making a transition and consuming a letter repeats until
  (a) the input word is empty, or the FA reaches a state with no transition for the next input letter.

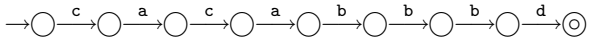
- When input word is empty and the FA is in an *accepting state*, indicated by a double-circle, we say that the FA *accepts* w.

REMARK. Termination condition (b) is not strictly necessary. As depicted to the right above, one can always add edges—an transition labelled with more than one letter represents more than one transitions—to the transistion relation for missing letters, taking the automaton to a "dead" state that consumes all the remaining input letters, and from which there is no path to an accepting state. The result is a finite-automaton that acceptes exactly the same words. □

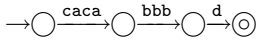
Example 9.1 For example, suppose this FA is presented with the word `cacabbbd` the path it follows is



We call such a path a *trace* of the FA. It is customary to omit the state names, as in



to assert that a trace exists, in this case an *accepting trace*, for `cacabbbd`. We may at times abbreviate this further to something like



□

9.2.1 Formalization

Let us begin to formalize the idea of a finite-state automaton.

Definition 9.1 *Let A be an alphabet. A* finite-state automaton*, or* FA over alphabet A *is a structure*

$$\langle S, s_0, F, T\rangle$$

consisting of

- *a finite set S is of* states*,*
- *a single* intital *or* starting *state $s_0 \in S$*
- *a non-empty subset $F \subseteq S$ of* final *or* accepting *states, and*
- *A* transition relation $T \subseteq (S \times A) \times S$*.*

Next we define how a FA "executes."

Definition 9.2 *Given a finite-state automoton, $FA = \langle S, s_0, F, T\rangle$ over alphabet A and a word $w = a_0\, a_1\, \ldots\, a_{n-1} \in A^*$, we say FA* accepts *$w$ if*

(a) *$w = \varepsilon$ and $s_0 \in F$, or*

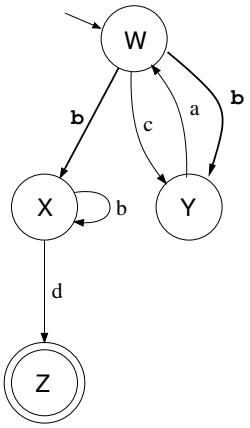
(b) *there is a path,* $P = \langle s_0, s_1, \ldots, s_n \rangle$ *in* T *such that* s_0 *is the starting state,* $s_n \in F$ *is an accepting state, and for each successive pair* (s_i, s_{i+1}), $0 \leq i < n$, *there is a pair* $((s_i, a_i), s_{i+1}) \in T$.

Definition 9.3 *Given a finite-state automaton, FA over alphabet* A *the language* $\mathcal{L}(FA) \subseteq A^*$ *is the set of all words accepted by FA*

9.2.2 Nondeterminism

The transition relation of Definition 9.1 raises at least one important question: "What does it mean when the transition relation is not a function?"

When T is a function, or a partial function, it is uniqely determined what action is taken on each input letter. Suppose to the contrary that our FA allows two distinct transitions from a state on the same latter. For instance, below we have added a second transition under the letter `b` from state W.

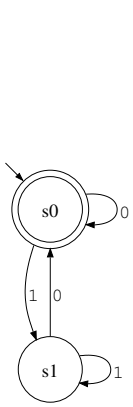


The definition of acceptance still holds; it asks only whether a trace *exists*. However, the question of how that path is selected is less clear. How would a physical mechanism decide which transition to take? Would it have to "predict" what input letters it will see in the future? If so, does this capability make the automata more powerful in the sense that they accepts a broader class of languages?

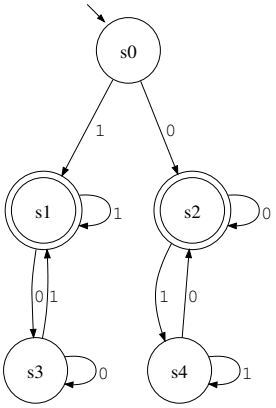
Definition 9.4 *A* deterministic finite-state automaton *(DFA) is one whose transition relation is a (partial) function. A* non-determinsitic finite-state automaton *(NFA) is one whose transition relation is not a function, that is, an FA with multiple distict transitions from some state for the same letter.*

9.3 Composing Automata

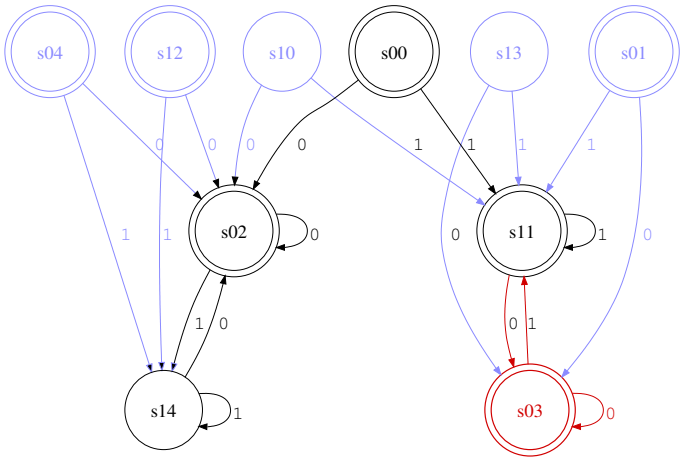
9.3.1 Parallel Composition



$L_1 = \{$*binary words that end with* 0$\}$



$L_2 = \{$*binary words that start and end with the same letter*$\}$



$L_1 \cup L_2$

"product" FA

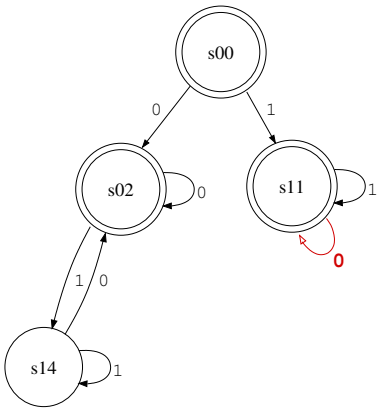
$S = S_1 \cup S_2$

$\mathring{s} = (\mathring{s}_1, \mathring{s}_2)$

$F = (F_1 \times S_2) \cup (F_2 \cup S_1)$ (For $L_1 \cap L_2$, $F = F_1 \times F_2$)

$T = \{((s_1, s_2), (a_1, a_2)), (s_1', s_2')) \mid ((s_1, a_1), s_1') \in T_1$ *and* $((s_2, a_2), s_2') \in T_2\}$

⇓ *Remove unreachable (blue) and redundant (red) states*



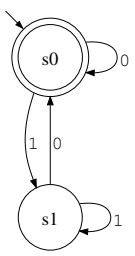
$L_1 \cup L_2$

9.3.2 Empty Transitions

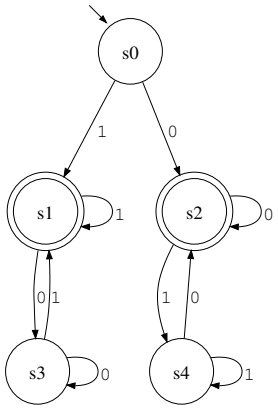
Definition 9.5 *An* empty *transition is a transition labelled by ε. A DFA or NFA with an empty transition may nondeterminstically move to the target state without consuming the next input letter.*

Equivalently, on may think of the input word as having one or more εs between any two letters.

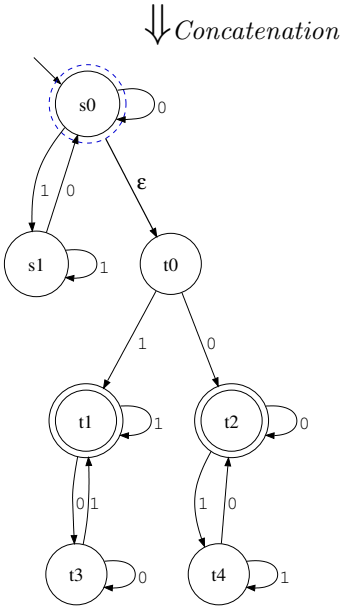
Example 9.2



$L_1 = \{$*words that end with* $\mathtt{0}\}$



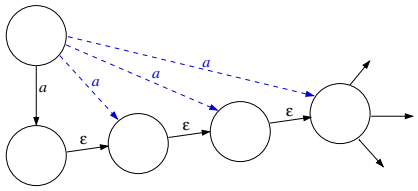
$L_2 = \{$*words that start and end with the same letter*$\}$



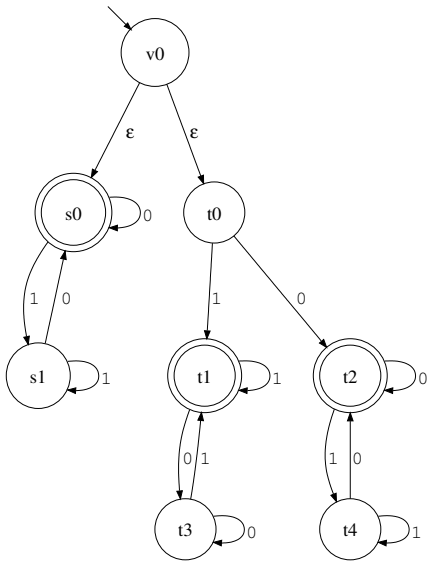
$L_1 \hat{} L_2 = \{u \hat{} v \mid u \in L_1 \text{ and } v \in L_2\}$

□

Proposition 9.1 *Given any FA with empty transitions, A_ε there is an automaton A that accepts the same language.*

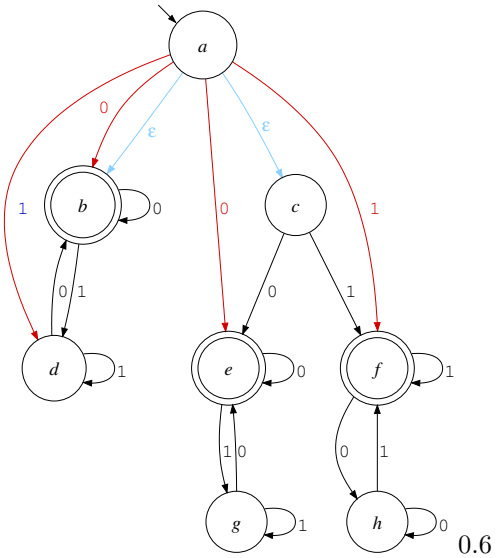


9.3.3 Removing Nondeterminism (!?)

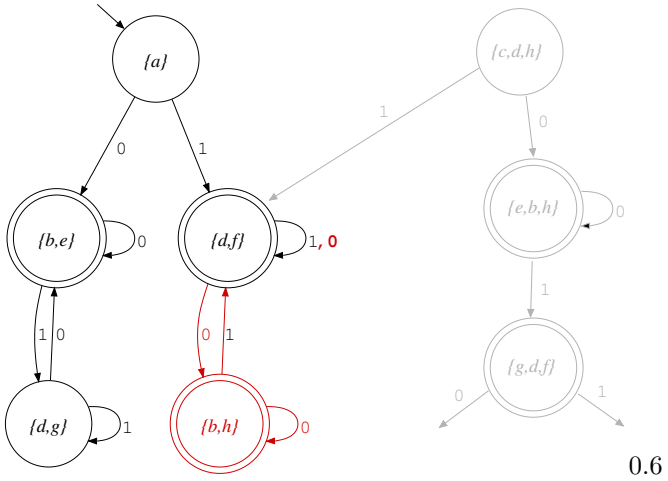


NFA for $L_1 \cup L_2$

⇓ *Remove ε-transitions"*



⇓ *"Power" automaton*



9.4 Regular Expressions

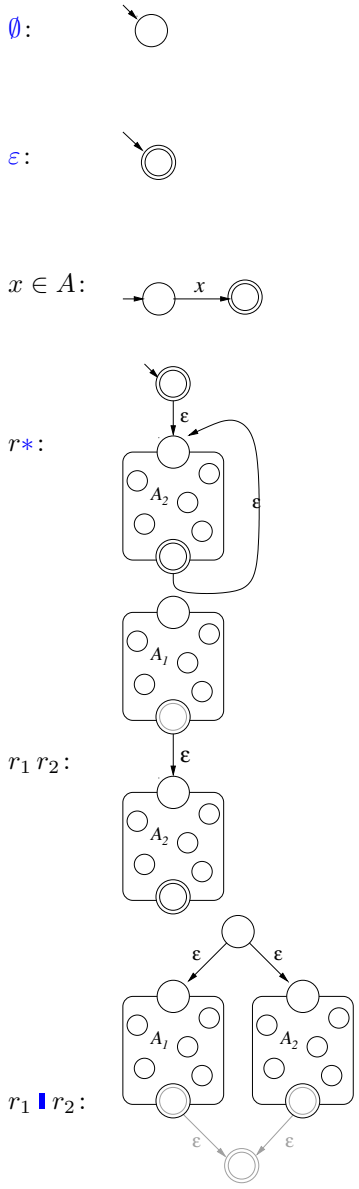
Regular expressions are a language for describing regular languages.

Definition 9.6 *For a given alphabet A, the language* REXP *of* regular expressions *and their interpretation as a languages over A are defined:*

$\text{REXP} \subseteq \big(A \cup \{\, \mid,\ *,\ \varepsilon,\ \emptyset,\ (,\ ) \,\}\big)^*$			$\mathcal{R}\colon \text{REXP} \to \mathcal{P}(A^*)$
$\langle \text{REXP} \rangle$	::=	$\emptyset$	$\mathcal{L}[\,\emptyset\,] = \emptyset$
	$\mid$	ε	$\mathcal{L}[\,\varepsilon\,] = \{\varepsilon\}$
	$\mid$	A	$\mathcal{L}[\,x\,] = \{x\}$ *for* $x \in A$
	$\mid$	$\langle \text{REXP} \rangle *$	$\mathcal{L}[\,r*\,] = \{u^n \mid u \in \mathcal{L}[\,r, n \in \mathbb{N}\,]$
	$\mid$	$\langle \text{REXP} \rangle\, \langle \text{REXP} \rangle$	$\mathcal{L}[\,r_1\, r_2\,] = \{u \hat{} v \mid u \in \mathcal{L}[\,r_1, v \in \mathcal{L}[\,r_2\,]$
	$\mid$	$\langle \text{REXP} \rangle \mid \langle \text{REXP} \rangle$	$\mathcal{L}[\,r_1 \mid r_2\,] = \mathcal{L}[\,r_1\,] \cup \mathcal{L}[\,r_2\,]$
	$\mid$	$(\ \langle \text{REXP} \rangle\ )$	$\mathcal{L}[\,(\ r\ )\,] = \mathcal{L}[\,r\,]$

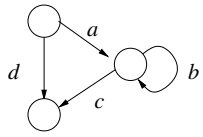
Theorem 9.2 *There is a finite-state automaton that accepts the language specified by any regular expression.*

PROOF. The proof is by induction on REXP. In each case an FA is recursively constructed. The constructions are shown informally below.

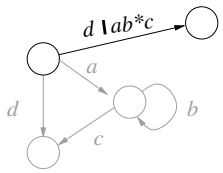


Theorem 9.3 *The language described by any finite-state automaton can be specified by a regular expression.*

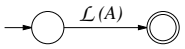
PROOF IDEA. There are a number of ways to reduce a FA to a regular expression (Google "finite-automaton to regular expression"). One way is to perform "state reduction" on *automata over* REXP. For instance, one reduction rule might translate



to



As reduction continues some of the states become unreachable and can be discarded. The automaton ultimately reduces to



and the REXP labelling the single remaining edge specifies the language accepted by the original automaton. □

Like other constructions, such as removing nondeterministic transitions, the reduction of an automaton to a regular expression may involve explosion in the size of the resulting expression.

Example 9.3 Let $A = \{a, b, c, d\}$. Is it possible to use regular expressions to specify a language of words in which no letter occurs twice in succession?

SOLUTION The expression below was contributed by Abhijit Mahabal, who

generated it using the JFLAP visualization tool (`www.jflap.org`).

$$
\begin{array}{l}
\quad \varepsilon \\
| \ a \\
| \ (b|ab)(ab)^*(\varepsilon|a) \\
| \ (c|ac|(b|ab)(ab)^*(c|ac))(ac|(b|ab)(ab)^*(c|ac))^*(\varepsilon|a|(b|ab)(ab)^*(\varepsilon|a)) \\
| \quad (\ d \\
\quad\ \ | \ ad \\
\quad\ \ | \ (b|ab)(ab)^*(d|ad) \\
\quad\ \ | \ (c|ac|(b|ab)(ab)^*(c|ac))(ac|(b|ab)(ab)^*(c|ac))^*(d|ad|(b|ab)(ab)^*(d|ad)) \\
\quad\ )(\ ad \\
\quad\ \ | \ (b|ab)(ab)^*(d|ad) \\
\quad\ \ | \ (ac|(b|ab)(ab)^*(c|ac))(ac|(b|ab)(ab)^*(c|ac))^*(d|ad|(b|ab)(ab)^*(d|ad)) \\
\quad\ )^*(\ \varepsilon \\
\quad\ \ | \ a \\
\quad\ \ | \ (b|ab)(ab)^*(\varepsilon|a) \\
\quad\ \ | \ (ac|(b|ab)(ab)^*(c|ac))(ac|(b|ab)(ab)^*(c|ac))^*(\varepsilon|a|(b|ab)(ab)^*(\varepsilon|a)) \\
\quad\ )
\end{array}
$$

REMARK. Though valid (probably), this is not something a human could write, or even comprehend, without a bit of thought. If you examine the formula, you will begin to see some repeated patterns. We can identify some of these subformulas and define languages to go with them, in order to build a hierarchy.

The subformula $(b|ab)(ab)^*(\varepsilon|a)$ describes a language over the alphabet $\{a, b\}$ in which neither a nor b occurs twice in succession in any word. Name this language L_b

$$L_b = \varepsilon|a|(b|ab)(ab)^*(\varepsilon|a)$$

The sublanguages L_c and L_d add some words that contain cs and some that contain ds.

$$L_c = ac|(b|ab)(ab)^*(c|ac)$$
$$L_d = d|ad|(b|ab)(ab)^*(d|ad)$$

Using these languages, the large formula above reduces to

$$L_b|(c|L_c){L_c}^*L_b|(d|L_d|(c|L_c){L_c}^*(d|L_d))(L_d|L_c{L_c}^*(d|L_d))^*(L_b|L_c{L_c}^*L_b)$$

This formula combines the sublanguages, adding in some more cs and ds where needed. You can check that words in this language satisfy the property of no successive occurances a, b, c or d; but it is much more difficult to determine whether this formula works *for all* such words.

□

9.5 Is That All There Is?

Proposition 9.4 *(Pumping Lemma) Let L be a regular language. Then there is a whole number p that depends only on L such that every word $w \in L$ of length at least p is of the form $w = xyz$ where*

(a) $|y| > 1$

(b) $|xy| \leq p$

(c) *for all $i \in \mathbb{N}$, $xy^i z \in L$.*

Example 9.4 $\{u^n\, v^n | n \in \mathbb{N}\}$ is not regular. □

