State-transition diagrams

This material is from Chapter 8 in the textbook.

Example. An identifier can be defined as a string of letters and digits that begins with a letter.

- Token **letter** stands for any of the symbols a, . . . , z, A, . . . , Z.
- Token **digit** stands for 0, 1, . . . , 9.

The set of identifiers is specified by the state-transition diagram in Figure 1.

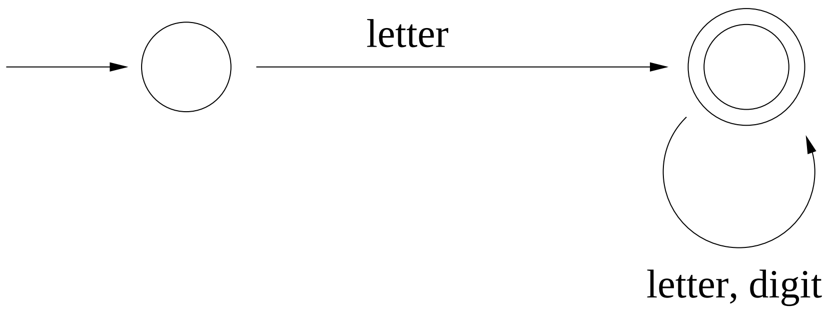

Figure 1: A state-diagram for specifying identifiers.

Example. We design a "sequential lock" as described below. The lock has 1-bit sequential input. Initially, the lock is closed. If the lock is closed it will open when the last three input bits are "1", "0", "1", and then remains open.

In other words, the state-transition diagram should accept exactly all strings that contain substring 101. The state-transition diagram will be constructed in class.

What is a regular expression that denotes the same language?

A state diagram describes a **deterministic finite automaton** (DFA), a machine that at any given time is in one of finitely many states, and whose state changes according to a predetermined way in response to a sequence of input symbols.

A formal definition is given below.

Definition. A DFA is defined as a tuple

$$M = (Q, \Sigma, \delta, s, F)$$

where the components are as follows:

- Q is the finite nonempty set of states
- Σ is the input alphabet
- $\delta : Q \times \Sigma \longrightarrow Q$ is the transition function
- $s \in Q$ is the starting state
- $F \subseteq Q$ is the set of accepting states

A finite state automaton (DFA) is conveniently specified using a state diagram, especially when the set of states is small.

Example. Consider the state diagram in Figure 2.

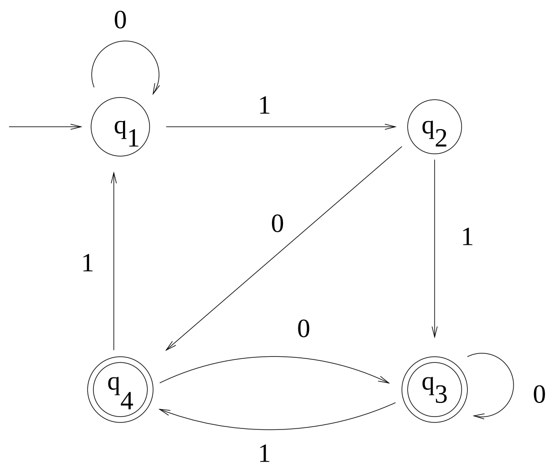

Figure 2: A state-transition diagram.

In a formal notation this automaton can be specified as

$$M = (Q, \Sigma, \delta, s, F)$$

where

- the set of states is $Q = \{q_1, q_2, q_3, q_4\}$,
- the input alphabet is $\Sigma = \{0, 1\}$,
- the starting state is q_0,
- the set of accepting states is $\{q_3, q_4\}$, and
- the transition function $\delta : Q \times \Sigma \longrightarrow Q$ is given by the below transition table:

Current state/input	0	1
q_1	q_1	q_2
q_2	q_4	q_3
q_3	q_3	q_4
q_4	q_3	q_1

In a state diagram the starting state is denoted by a circle with an "incoming arrow" and an accepting state is denoted by a double circle.
Note: A state diagram has only one starting state. There can be more than one accepting states (or no accepting states).

On the other hand, when a DFA is implemented a graphic state diagram notation is not sufficient. One needs to, basically, give names for the states and encode the transition table in the code, see the examples in section 8.2 of the textbook.

Definition. A state diagram (or DFA) *accepts* a string

$$c_1 c_2 \cdot \ldots \cdot c_n$$

if there is a path from the starting state to an accepting state that is labeled by symbols c_1, …, c_n. The language *recognized* by the state diagram (or DFA) consists of all strings accepted by it.

Example. Construct a state diagram for recognizing comments that may go over several lines:

`/* ......*/`

Nondeterminism

- The state diagrams that we have considered up to now were *deterministic:* for any state and input symbol pair there can be at most one outgoing transition.
- A *nondeterministic* state diagram allows the following type of situations:

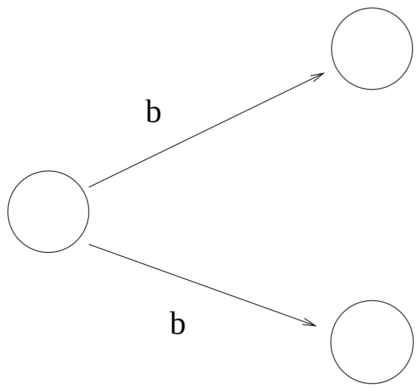

Figure 3: Nondeterministic transitions.

When dealing with nondeterministic state diagrams it is important to remember that now a given string may label more than one path beginning from the starting state and we have to be careful how the acceptance of a string is defined:

Nondeterministic acceptance: a string is accepted if it appears on **any** path from the starting state to an accepting state.

Why nondeterminism?

- Often it is much easier to construct a nondeterministic state diagram than a deterministic one.
- A nondeterministic state diagram can be *much* smaller than the smallest possible deterministic state diagram that recognizes the same language.
- On the other hand, it is not immediately clear how we can implement nondeterministic state diagrams since program behavior is deterministic (or, at least, it *should* be deterministic). Fortunately, there is a way to convert a nondeterministic state diagram into an equivalent deterministic one, and the conversion can even be automated.

Example. Consider the following nondeterministic state diagram with input alphabet $\{0, 1\}$:

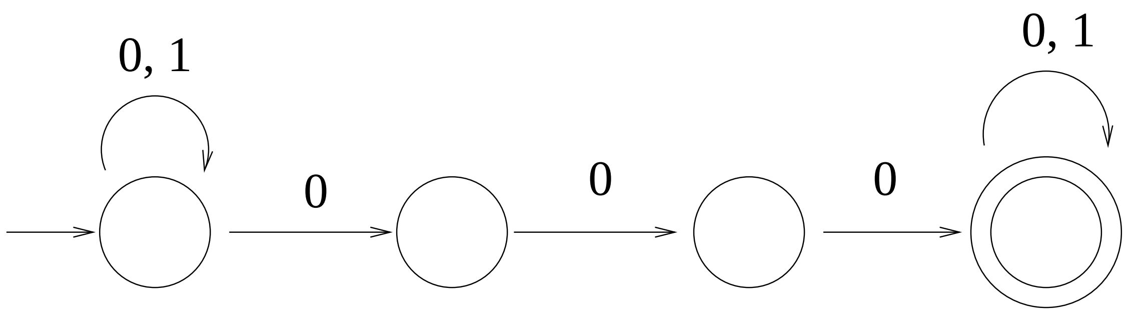

Figure 4: A nondeterministic state diagram.

- Where does the nondeterminism appear in the state diagram of Figure 4?
- What is the language recognized by the state diagram of Figure 4?
- How would you construct an equivalent DFA?

Nondeterministic state diagrams are also called *nondeterministic finite automata,* NFA.

An NFA can be implemented as a DFA using the so called *subset construction.* The subset construction is explained on pages 179–181 in the textbook. We illustrate the subset construction by applying it to the following example (to be done in class).

Example. Consider the following NFA:

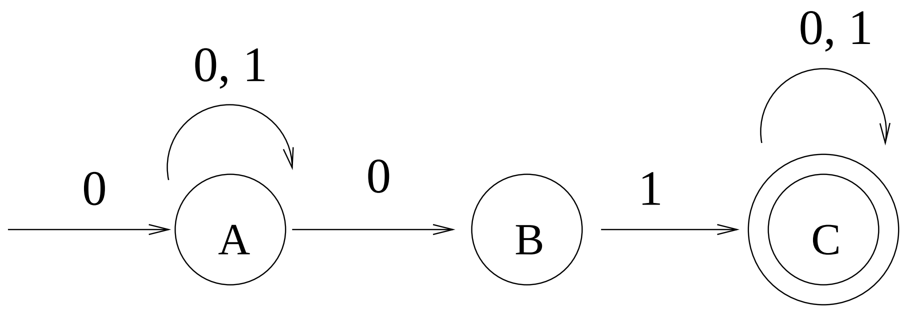

Figure 5: A nondeterministic state diagram.

We construct a DFA that is equivalent to the NFA from Figure 5 using the subset construction. The DFA keeps track of the set of *all* possible states that the NFA may be in after reading the current sequence of input symbols. The starting state is $\{A\}$. A set of states is accepting if it contains at least one accepting state of the original NFA. The transitions of the DFA are defined by following all possible nondeterministic transitions from the current state (as explained on pages 179–181 in the textbook). The transition table and the state diagram of the DFA that is obtained by applying the subset construction to the NFA of Figure 5 will be done in class.
Question: How could you simplify the resulting deterministic state diagram?

ε-transitions

Sometimes it is convenient to allow in nondeterministic state diagrams transitions that do not read an input token ("spontaneous" transitions). These are called ε-transitions.

Note: A deterministic state diagram <u>cannot</u> have ε-transitions.

Example. Let $\Sigma = \{0, 1\}$ and define

$S_1 = \Sigma^* 01 \Sigma^*$

$S_2 = \{w \in \Sigma^* \mid w \text{ has an even number of 1's } \}$

The following state diagram recognizes $S_1 \cup S_2$:

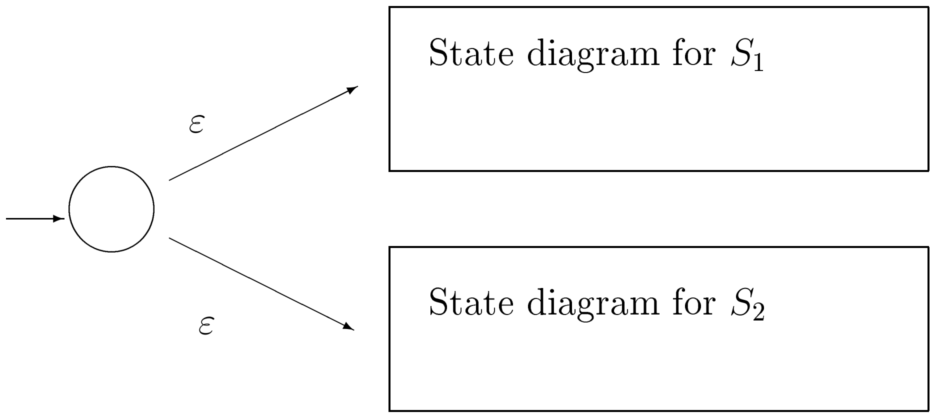

Figure 6: A state diagram for $S_1 \cup S_2$.

A state diagram with ε-transitions is called an ε-NFA.

Given an ε-NFA M we can construct an equivalent NFA without ε-transitions as follows (see page 183 in the textbook):

1. Make a copy M' of M where the ε-transitions have been removed. Remove states that have only ε-transitions coming in, however, the starting state is not removed.

2. Add transitions to M' as follows: whenever M has a chain of ε-transitions followed by a "real" transition on $x \in \Sigma$, see Figure 7:

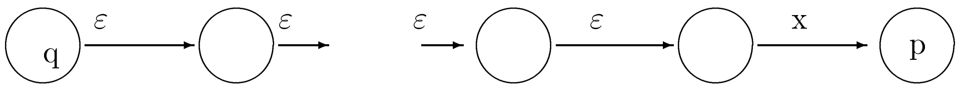

Figure 7: A chain of ε-transitions followed by a "real" transition.

we add to M' a transition from state q to state p that is labeled by x. Note that here q and p may be any states. For example, the above construction step is used also in the case where $q = p$.

3. If M has a chain of ε-transitions from a state r to an accepting state, then r is made to be an accepting state of M'.

Example. How does the construction work with the state diagram:

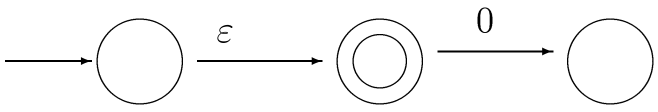

Figure 8: An ε-NFA.

Example. The below state diagram (Figure 9) recognizes unsigned decimal numbers:

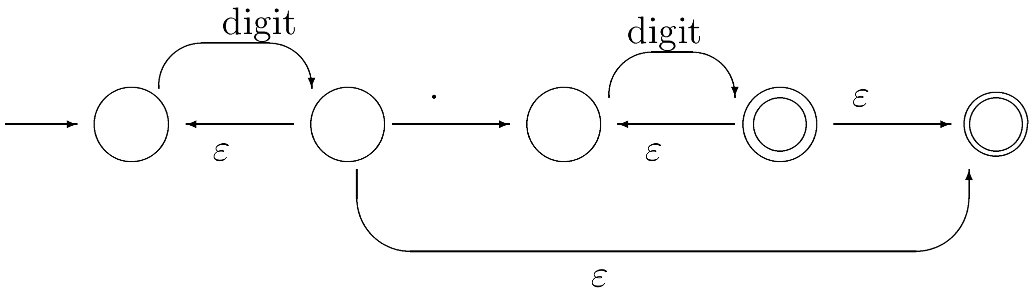

Figure 9: An ε-NFA for unsigned decimal numbers.

We construct an equivalent nondeterministic state diagram without ε-transitions (in class).

Note: By combining the above two transformations, that is, the elimination of ε-transitions and the subset construction, we can always convert a state diagram with ε-transitions to a deterministic state diagram.

Applications: Natural language processing

Noam Chomsky, one of the pioneers of language theory, showed already in the 50's that English (or any natural language) is not a finite-state language. Natural languages contain unlimited nested dependencies, and it is impossible to construct a finite automaton that keeps track of such dependencies. Next week (in section 9.4) we will develop techniques that allow us to formally prove that a given language cannot be recognized by *any* finite automaton.

For many decades, computational linguistics concentrated on more powerful formalisms, namely on extensions of context-free grammars (that we will study in chapters 10 and 11).

However, finite-state machines have made a comeback in modern natural language processing [1]. It turns out that writing large-scale high-level grammars for languages such as English is very hard. Although English as a whole is not a finite-state language, there are subsets of English for which a finite-state description is quite adequate and much easier to construct than an equivalent (phrase-structure) grammar. Also it was discovered that formal descriptions of phonological alterations used by linguists were, in fact, finite-state models.

Researchers have developed special finite-state formalisms that are suited for the description of linguistic phenomena, and compilers that efficiently produce finite automata from such descriptions [1]. The automata in linguistic applications are much too large and complex to be produced by hand. For example, the transition tables of automata used for text–to–speech translation of languages such as English, French or German typically require 25–30 Mbytes.

References

[1] K.R. Beesley and L. Karttunen: *Finite State Morphology,* CSLI Publications, 2003. Web page for the book: `http://www.fsmbook.com`