Cornerstones of Computer Science: *D*eterministic *F*inite *S*tate *A*utomata
Evo—the DFSA Simulator

By Richard Born
Associate Professor Emeritus
Northern Illinois University
rborn2@niu.edu

Introduction

"Deterministic Finite State Automata (DFSA)" sounds like it might be a bit scary and complex at first glance. But it isn't really! Let's start with the word "automata". An automaton (the singular for "automata") is just a fancy name for a machine that acts automatically. A "state" is a particular condition that the machine has at a specific time. "Finite" means that the number of states is limited or measurable. "Deterministic" means that no randomness is involved in deciding future states of machine. For simplicity, let's refer to such a device as a finite state machine (FSM).

Finite state machines pervade our society in numerous forms. They form the basis of microwave oven control, vending machine control, and TV remote controls, just to mention a few. Figure 1 shows three examples of some other finite state machines. The example at the top of Figure 1 is a house window. The window can be in any one of two *states*, either open or closed. The states are shown as complete circles in the *state diagram*. The arrow on the far left simply points to the initial state, which we will take as closed. Two possible *inputs* for the window are to "push down" or "pull up". The inputs are shown as arrowed segments, which represent *transitions* from one state to another. If the window is in the closed state, and the input "pull-up" is applied, the window will transition from the closed to the open state. If the window is in the open state, and the input "pull up" is applied, the window remains in the open state. If the window is in the open state, and the input "push-down" is applied, the window will transition from open to the closed state. Finally, if the window is in the closed state, and the input "push down" is applied, the window remains in the closed state.

The example in the middle of Figure 1 shows the state diagram for an LED flashlight with a single button that the author has. Having just one button, there is only one possible input, and that input is to press the button. As in the previous example, we will assume that the flashlight is initially in the Off_1 state. Press the button, and the state transitions to low beam. Press the button again, and the state transitions to Off_2. Another press and it transitions to high beam. A final press of the button and it transitions back to Off_1.

The example at the bottom of Figure 1 shows the state diagram for a 2011 Chevy Volt car door remote. This remote has two buttons  for controlling the doors. There are three states, with the arrow on the far left pointing to the initial state, in which we assume that all doors are **locked**. Pushing the *lock button* keeps the current state as is. Pushing the *open button* transitions the state from **locked** to **front doors unlocked**. Pushing the *open button* again transitions to a state in which **all doors are unlocked**. From the **front doors**

unlocked or the **all doors unlocked** states, pushing the *lock button* transitions to the locked state. If **all doors are unlocked**, pushing the open button keeps the state as is.

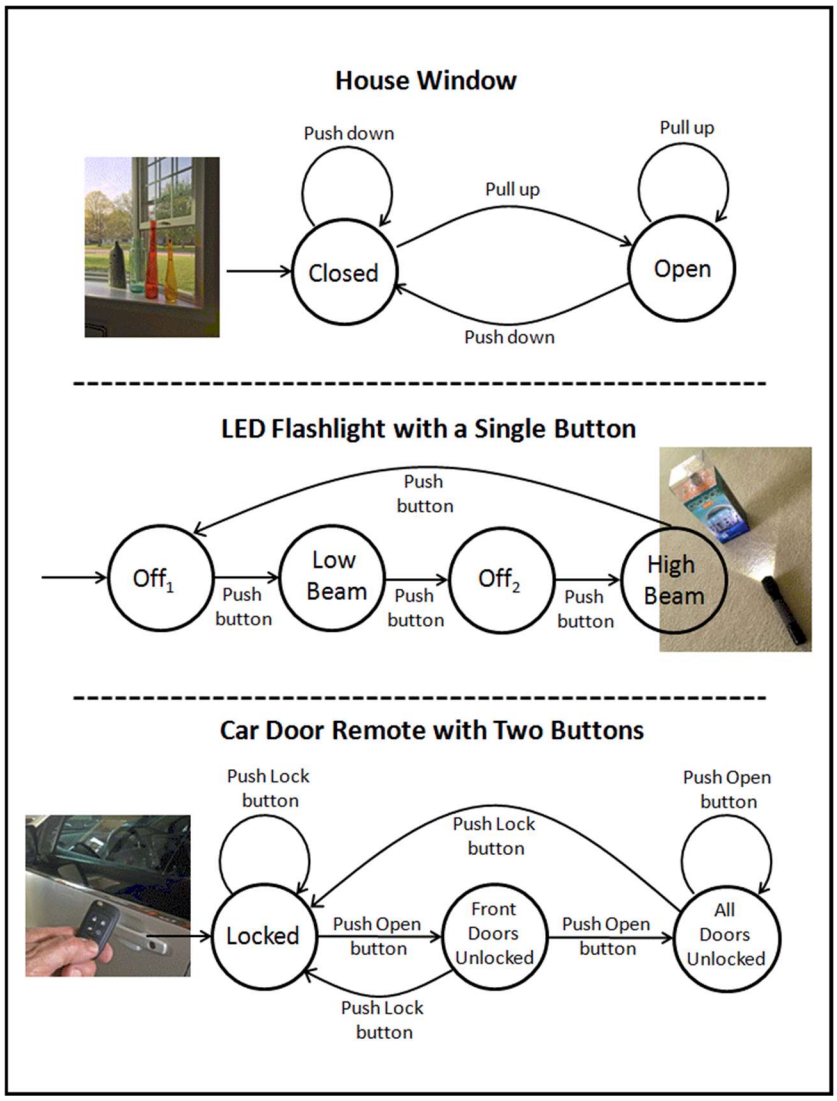

Figure 1

We'll return later to see how the finite state machines of Figure 1 can be simulated by the Evo DFSA simulator. But first we will take a look at DFSA's from the point-of-view of computer science.

DFSA's in Computer Science

A finite state automaton is a language recognition device consisting of a finite control, a read head, and an input tape, as shown in Figure 2. It does not have an auxiliary memory and must therefore rely upon remembering in its finite set of states. The input is usually received as a string of characters on an input tape. The finite control is capable of sensing the character written on any position of the tape by use of the read head. Initially, the read head is scanning the left-most character on the input string, and the finite control is in what is often called the ***start state***. At regular time intervals, the automaton reads one character from the input string, and then it enters a new state that is a function of only two things—the current state and the

character being read. Each time a character is read, the read head advances one character to the right on the input string. This process continues until the read head advances past the right-most character on the input string, in which case the string is said to be ***consumed***. At this point the automaton either accepts or rejects the input string. The string is said to be ***accepted*** if the automaton winds up in a state that is a member of a set of ***final states***; otherwise the string is ***rejected***. Other than acceptance or rejection, the automaton produces no output. It is a language recognition device, and the language recognized by the finite state automaton is simply the set of string it accepts.

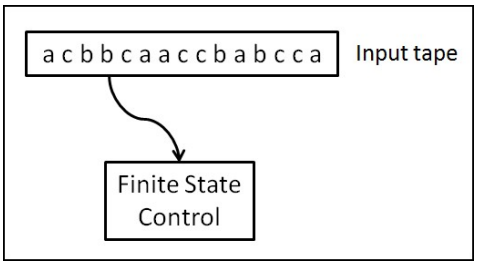

Figure 2

The finite state automaton described above is deterministic because there is exactly one possible next state for the automaton, given its current state and character being read. This deterministic device forms the basic model for the Evo DFSA simulator.

A deterministic finite state automaton can be loosely defined as a quintuple **(Q, I, DELTA, q_0, F)** where:

- **Q** is a finite set of states,
- **I** is an input alphabet,
- $\mathbf{Q_0}$ is the start state (and a member of **Q**),
- **F** is a set of final states (and a subset of **Q**), and
- **DELTA** (which we abbreviate as **D**) is the transition function defining the moves.

The transition function, DELTA, contains the rules that allow the DFSA to determine its next state. For example, if the DFSA is in state **q** and the character being read on the input string is **a**, then **DELTA**(**q**, a) is unique state **r** to which the DFSA goes. See Figure 3, which shows how this transition would be depicted in a state diagram.

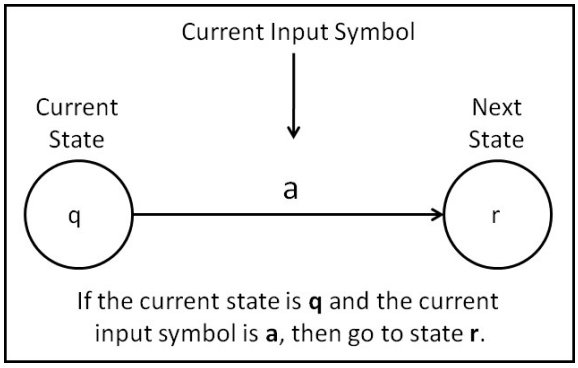

Figure 3

A computation on a DFSA is often expressed by a sequence of what are known as configurations. A configuration takes into account the current state, read head, and the input string. Since the read head of a DFSA can only move to the right, the portion of the input string to the left of the read head cannot have any effect on the future of the DFSA. Therefore, the configuration of a DFSA is completely determined by the current state and the unread portion of the input string. A first configuration **yields** a second configuration **in one step** if there is a transition that can achieve this. **Yields in one step** is denoted by the symbol → in *Cornerstones of Computer Science*.

Evo's Deterministic Finite State Automaton

With all of the above discussion describing the theoretical model of a DFSA complete, we are now ready to look at the automaton that Evo simulates. Evo's quintuple **(Q, I, D, q_0, F)** is as follows:

- **Q** **States:** {0, 1, 2, 3, 4, 5}
- **I** **Input alphabet:** {a, b, c}
- **q_0** **Start state:** 0
- **F** **Final States:** User definable
- **D** **Transition function:** User definable

We see that the Evo DFSA allows up to six states numbered 0 through 5, and that the start state is always state 0. The input alphabet consists of the three characters a, b, and c. The set of final states can be any subset of the set of states Q.

An Example DFSA

Let's see if we can come up with a DFSA recognizing the set of all strings of a's and b's that contain three consecutive a's (and possibly no b's). When designing DFSA's it is often easier to begin with what is known as a state diagram, and from the state diagram determine a more formal definition. A state diagram and list of transitions for our example DFSA is shown in Figure 4.

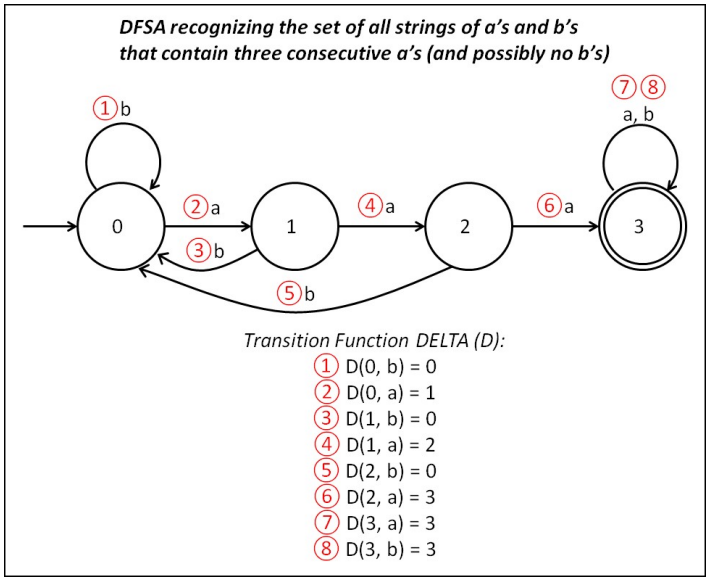

Figure 4

The blank arrow on the far left simply points to the start state, which is always state 0 for the Evo DFSA simulation. All the remaining arrows show transitions. Final states are commonly represented in state diagrams by double circles. In our DFSA, you can see that state 3 is a final accepting state. The red circled numbers in the state diagram and list of transitions are included to make it easier to discuss the purpose of each transition.

While in state 0, transition ① indicates that any non-negative number of **b**'s can be accepted. Transition ② takes us to state 1, to recognize the first **a** in a possible sequence of three **a**'s. If we read a **b** while in state 1, we go back to state 0 by transition ③, as our possible sequence of three **a**'s has been destroyed by reading the **b**. However, if we are in state 1 and read an **a**, by transition ④, we go to state 2 to recognize the second **a** in a possible sequence of three **a**'s. Transition ⑤ takes us back to state 0 if we read a **b**, as once again our possible sequence of three **a**'s has been destroyed. Transition ⑥ takes us to state 3 if we input our third **a** in sequence. Once is state 3, transitions ⑦ and ⑧ allow reading any sequence of **a**'s and **b**'s. Since state 3 is a final state, the input string is accepted after being consumed.

Now, as an example, let's look at the sequence of configurations that leads to the acceptance of the input string babaabaaaba:

(0, babaabaaaba) → (0, abaabaaaba)	by Transition ①
→ (1, baabaaaba)	by Transition ②
→ (0, aabaaaba)	by Transition ③
→ (1, abaaaba)	by Transition ②
→ (1, baaaba)	by Transition ④
→ (0, aaaba)	by Transition ⑤
→ (1, aaba)	by Transition ②
→ (2, aba)	by Transition ④
→ (3, ba)	by Transition ⑥
→ (3, a)	by Transition ⑧
→ (3, ^)	by Transition ⑦

Since (0, babaabaaaba) "eventually" yields the configuration (3, ^) and 3 is a final state, then this DFSA accepts the string babaabaaaba.

Setting up the Example DFSA Transitions in OzoBlockly

A program entitled *DFSA Template.ozocode* is an OzoBlockly file that accompanies this document. This program contains the transitions for our example DFSA program. It is also intended for inputting transitions for the automata of the exercises or automata of your own design. Finally, this program is used to test strings and debug your automaton logic, if necessary. You don't have to worry about any code other than that for your automaton's transitions and final states. In this section, you will learn how to set up the transitions and final states for the example DFSA of the previous section of this document.

If you open *DFSA Template.ozocode* from within the OzoBlockly web site, you will find the transitions and final states for the example DFSA are on the far left of the OzoBlockly workspace. Figure 5 shows the transitions along with their respective circled transition numbers discussed in the previous section

of this document. Embedded in the *Define Transitions* subroutine are eight calls to the subroutine *Transition with*. The three arguments for the *Transition with* subroutine correspond to the three parameters of the DELTA (D) function: current state, input character, and next state.

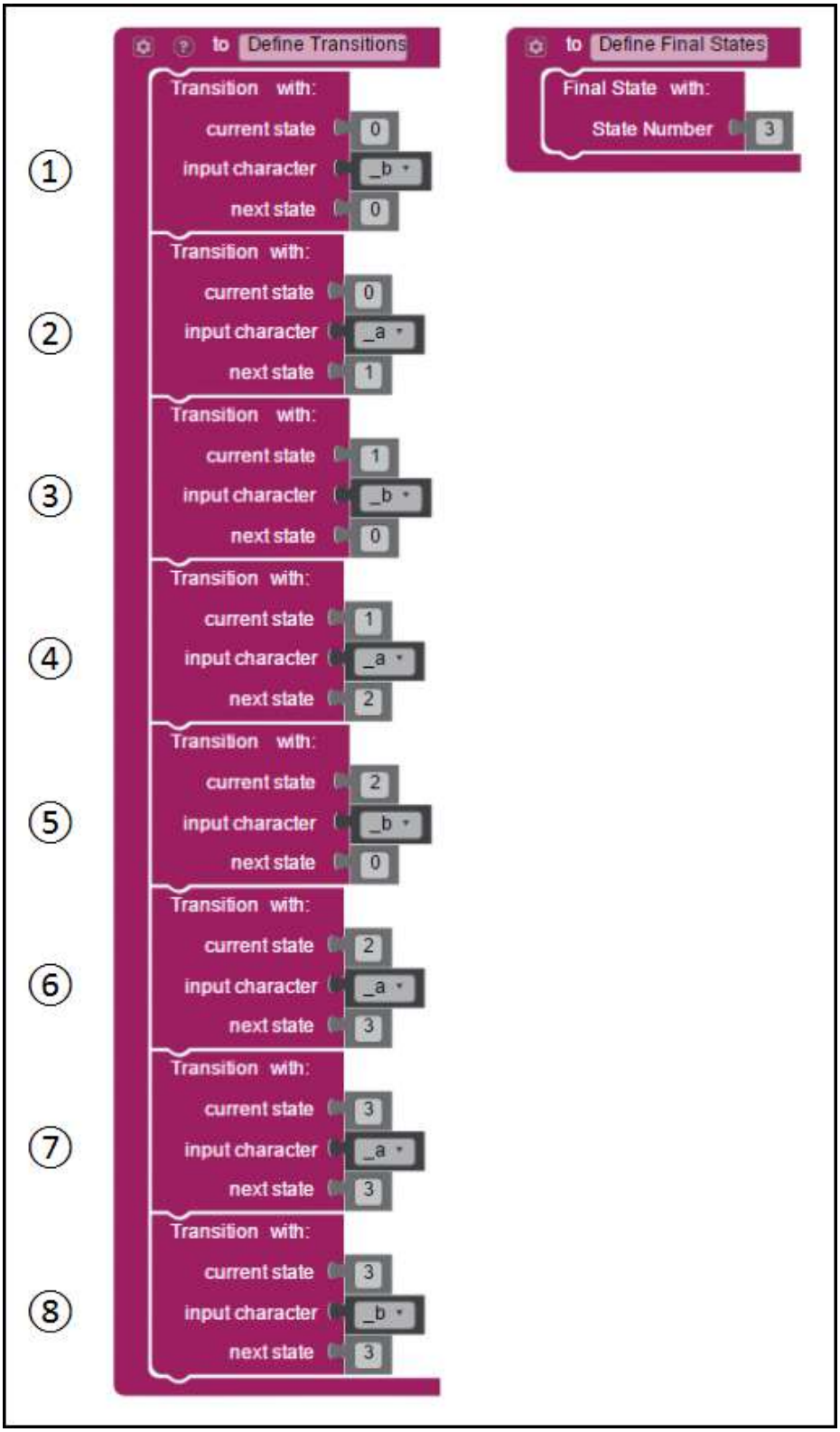

Figure 5

The current state and next state are always integers in the range 0 through 5. These numbers can be adjusted as needed to reflect states in your transitions. The input character is always one of _a, _b, or _c. These can be changed quickly by clicking the down arrow and selecting the desired character from these three. Think of the underscore in the variable name as meaning *character*. For example, _b would mean the input *character* b. The underscore is required.

The *Define Final States* subroutine shows that this DFSA has one final state, *i.e.,* state 3. This state is declared in the subroutine *"Final State with"* that is embedded within the *Define Final States* subroutine. If there is more than one final state for your automaton, you can embed additional copies of *"Final State with"* to define those final states.

The suggested way to ***add new transactions*** is as follows. Right-click on any transition and select *Duplicate* from the pop-up menu. Adjust the three arguments to the desired values. Drag the transition so that it is embedded in the *Define Transitions* subroutine. ***The order of transitions within the Define Transitions subroutine makes no difference.***

To ***delete a transaction***, right-click on the transition to be deleted, and select *Delete 4 Blocks* from the pop-up menu. To immediately undo the delete, click on the *clockwise circle-arrow* icon on the top right of the OzoBlockly screen.

This space is intentionally blank.

Working with the Evo DFSA Simulation Program

Let's assume that you have loaded the program *DFSA Template.ozocode* into Evo. This program will allow Evo to simulate acceptance or rejection of strings for the language of our example—strings with three consecutive **a**'s. For ready reference, a small copy of the Ozomap that is used for any and all DFSA's that you want to simulate using Evo is shown in Figure 6. A full page version, suitable for use with Evo, can be found on the last page of this document.

Evo is always initially placed on the maze centered on the intersection by the start state 0 and must be facing to the right. The information in the center of the circular path of states is a reminder of the definition of the DFSA simulated by Evo. Evo will remain stopped at any state for 6 seconds, giving you time to enter the next character from the input tape. As explained in the bottom left corner of the Ozomap, you would press Evo' button once to input the character **a**, twice to input the character **b**, and three times to input the character **c**. Each time you press Evo's button, Evo will quickly flash white to let you know that it has recognized the button press. Make sure that you see the flash before

pressing the button again. After the 6 seconds have elapsed, Evo will speak the number of the next state, and then move to that state. If you don't press the button at all, this is an indication that all characters on the input tape have been read.

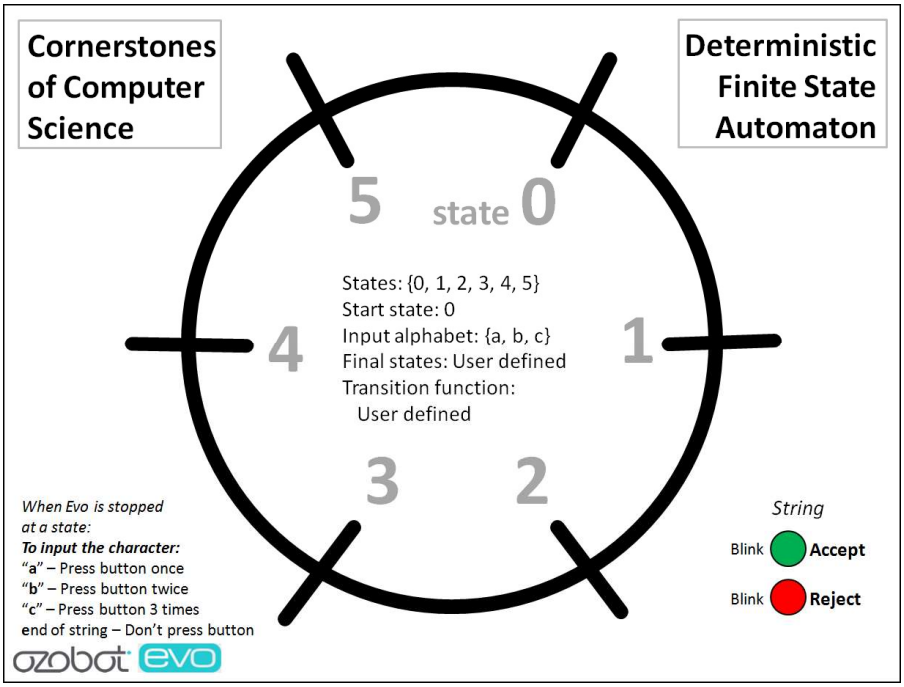

Figure 6

If Evo accepts the string, his top light will blink green for a few seconds and Evo will then turn off. If Evo blinks red for a few seconds and then turns off, this indicates that the input string has been rejected. This information is summarized in the bottom right corner of the Ozomap.

Running the Evo DFSA Simulation Program

1. Open the program *DFSA Template.ozocode* while at the OzoBlockly web site. After scrolling to the far left of the OzoBlockly workspace, you will see the subroutines called *"Define Transitions"* and *"Define Final States"*. We'll assume here these subroutines appear as shown in Figure 5. (These are the default transitions and final states in this file for our example, assuming you have made no changes.) Suppose that you want to test the string babaa, to determine if it is in the language.
2. Now load the program *DFSA Template.ozocode* into Evo, and then place Evo on intersection at state 0 on the Ozomap found on the final page of this document.
3. Assuming that Evo is currently turned off, press his button once to turn him on. The top and front lights will turn light blue. While they are light blue, quickly ***double-press*** Evo's button. This will start running the loaded OzoBlockly program. The top and front light will turn off for three seconds.
4. After the three seconds have elapsed, Evo' top light will turn deep blue. You will have six seconds to enter the first character in your test string. Since the first character is **b**, press Evo's button twice. Evo will respond by quickly flashing his top light white after each press to let you know that he has understood your button press. (Don't press twice too quickly; wait for the white flash acknowledgement after each press of the button.)

5. When the six-second interval has elapsed, Evo will speak the next state “0”, and then loop to that state on the Ozomap. While moving, Evo’s top light will always be light blue. (Transition ① has just completed.)
6. The top light will remain deep blue for six seconds, during which you need to enter the next character in the test string. Since the next character is **a**, press Evo’s button once.
7. When the six-second interval has elapsed, Evo will speak the next state “1”, and then move to that state on the Ozomap. (Transition ② has just completed.)
8. The top light will again be deep blue for six seconds, during which you need to enter the next character in the test string. Since the next character is **b**, press Evo’s button twice.
9. When the six-second interval has elapsed, Evo will speak the next state “0”, and then move to that state on the Ozomap. (Transition ③ has just completed.)
10. The top light will remain deep blue for six seconds, during which you need to enter the next character in the test string. Since the next character is **a**, press Evo’s button once.
11. When the six-second interval has elapsed, Evo will speak the next state “1”, and then move to that state on the Ozomap. (Transition ② has just completed.)
12. The top light will again be deep blue for six seconds, during which you need to enter the next character in the test string. Since the next character is **a**, press Evo’s button once.
13. When the six-second interval has elapsed, Evo will speak the next state “2”, and then move to that state on the Ozomap. (Transition ④ has just completed.)
14. The top light will again be deep blue for six seconds, during which you need to enter the next character in the test string. Since the string has been consumed, do not press the button at all to indicate this fact.
15. When the six-second interval has elapsed, Evo will blink the top light red, indicating that the string is rejected.
16. Test the empty string. It should be rejected.
17. Test the string babaabaaaba. It should be accepted. Evo will blink the top light green to indicate acceptance of the string.

Returning to the “House Window” as a Finite State Machine

Ozobot Evo can be used to simulate many “real-world” finite state machines. As an illustration, let’s return to our three example FSM’s in the introduction. Specifically, let’s take a look at the simplest of these—a house window. Figure 7 shows our “house window” FSM along with the corresponding state diagram converted to Evo’s model of a DFSA.

It is seen that state 0 corresponds to the window *closed* state, and state 1 corresponds to the window *open* state. With 0 being the start state, we have assumed that the window is *initially* closed. Input **a** corresponds to the *push down* input, while input **b** corresponds to the *pull up* input being applied to the window. In the case of this FSM, it doesn’t really matter which states are final and which states are not final. You could make either one of the two states final, both final, or neither final. In our state diagram, shown at the bottom of Figure 7, we have elected to make state 0 final, as can be seen by the double circle there. When we stop supplying inputs, Evo will blink green if the window is closed and red if it is left open. This is probably a good idea, to provide a warning reminder to us in the event of an upcoming thunder storm. We will leave it as an exercise for you to implement this house window FSM with the Evo DFSA simulator. See Exercise 1.

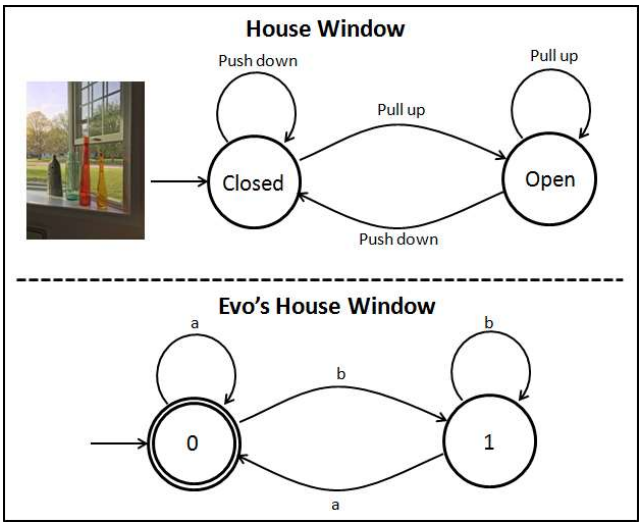

Figure 7

Exercises

1. Construct the *"**Evo's House Window**"* DFSA that was discussed on page 9 and whose state diagram appears in Figure 7. Begin with the *DFSA Template.ozocode* program, adjusting the *Define Transitions* subroutine as needed as well as the *Define Final States* subroutine. Load the program into Evo and then test your program, and debug if necessary, to make sure that it correctly follows your input according to the state diagram.
2. Construct a DFSA for the "***LED Flashlight with a Single Button***" that appears in Figure 1. Begin with the *DFSA Template.ozocode* program, adjusting the *Define Transitions* subroutine as needed as well as the *Define Final States* subroutine. Decide on a reasonable choice for final states—states for which Evo will blink green when input is consumed. Load the program into Evo and then test your program, and debug if necessary, to make sure that it correctly follows your input according to the state diagram.
3. Construct a DFSA for the "***Car Door Remote with Two Buttons***" that appears in Figure 1. Begin with the *DFSA Template.ozocode* program, adjusting the *Define Transitions* subroutine as needed as well as the *Define Final States* subroutine. Decide on a reasonable choice for final states—states for which Evo will blink green when input is consumed. Load the program into Evo and then test your program, and debug if necessary, to make sure that it correctly follows your input according to the state diagram.

For the remaining exercises, design state diagrams for the described language, define your transitions and final states in the Evo DFSA simulation program, and then load the program and test the suggested strings using Evo.

4. The language consisting of the set of all strings of a's and b's beginning with aa (and possibly containing no b's).

Test strings:	^	reject (empty string)
	abab	reject
	aabab	accept
	babbba	reject
	aa	accept
	b	reject

5. The language consisting of the set of all string of a's, b's, and c's containing the substring bab.

Test strings:		
	^	reject
	acbabca	accept
	ccbc	reject
	bbab	accept
	cbac	reject
	bab	accept

6. The language consisting of the set of all strings of the form b* or b*aaa*. Note that b* means any non-negative integral number of b's. b*aaa* would be a string of any non-negative number of b's followed by at least two a's. Computer scientists refer to expressions of this type as *regular expressions*.

Test strings:		
	^	accept
	b	accept
	bb	accept
	a	reject
	aa	accept
	baaaa	accept
	aab	reject
	bbaa	accept
	bbbaab	reject
	bbbaaa	accept

7. The language consisting of the set of all strings of a's, b'c, and c's with at least one a, one b, and one c for which the first **a** must precede the first **b**, and the first **b** must precede the first **c**.

Test strings:		
	^	reject
	abaacba	accept
	b	reject
	aaababa	reject
	abc	accept
	caab	reject
	aac	reject
	aababccab	accept

8. The language consisting of the set of all strings of a's, b's, and c's of length two or greater for which the first two characters are different.

Test strings:		
	^	reject
	aabcc	reject
	abc	accept
	bb	reject
	ccaba	reject
	abcab	accept
	baccb	accept
	accb	accept
	ca	accept
	b	reject
	cbabc	accept
	bcabc	accept

9. The language consisting of the set of all strings of a's, b's, and c's of the form $a^{2m+1}b^{2(n+1)}c^{p+1}$, where m, n, p ≥ 0. (Note that **a**$^{\mathbf{x}}$ means **x** a's in sequence, not **a** to the power **x**.)

Test strings:		
	^	reject
	aa	reject
	abb	reject
	abbc	accept
	aab	reject
	aaabbbbbcc	accept

10. The language consisting of the set of all strings of a's, b's, and c's containing *only* any number, including none, of abc triples. This would be represented by $(abc)^n$ as a regular expression, where n ≥ 0.

Test strings:		
	^	accept
	abca	reject
	abcabc	accept
	aabc	reject
	abc	accept
	abcabcabc	accept

11. The language consisting of the set of all strings of a's and b's with the number of a's congruent to 0, mod5, or congruent to 3, mod 5. (In other words, the number of a's can be 0, 5, 10, 15, … or 3, 8, 13, 18, …).

Test strings:		
	^	accept
	bb	accept
	ab	reject
	bbabaa	accept
	aaabaa	accept
	baaabbaaaa	reject

12. The language consisting of the set of all strings of a's and b's that have at least one b and an even number of a's after the final b. (Note that 0 is an even number).

Test strings:		
	^	reject
	a	reject
	ab	accept
	aabb	accept
	abbaa	accept
	abaaaa	accept
	aba	reject
	bbabaa	accept

13. The language consisting of the set of all strings of a's and b's for which the number of a's **and** the number of b's is *odd*. What simple change would you need to make in your state diagram if you wanted the set of all strings of a's and b's for which the number of a's and number of b's is *even*? (Note that 0 is an even number).

Test strings:		
	^	reject
	ab	accept
	aab	reject
	bab	reject
	bbba	accept