C.3 State machines

State machines are sequential. They begin in a starting state, and react to a sequence of inputs by sequentially transitioning between states. Implementation of state machines in software is fairly straightforward. In this lab, we explore doing this systematically, and build up to an implementation that composes two state machines.

C.3.1 Background

Strings in Matlab

State machines operate on sequences of symbols from an alphabet. Sometimes, the alphabet is numeric, but more commonly, it is a set of arbitrary elements with names that suggest their meaning. For example, the input set for the answering machine in figure 3.1 is

$$\mathit{Inputs} = \{\mathit{ring}, \mathit{offhook}, \mathit{end\ greeting}, \mathit{end\ message}, \mathit{absent}\}.$$

Each element of the above set can be represented in Matlab as a string (try `help strings`). Strings are surrounded by single quotes. For example,

```
>> x = 'ring';
```

The string itself is an array of characters, so you can index individual characters, as in

```
>> x(1:3)

ans =

rin
```

You can join strings just as you join ordinary arrays,

```
>> y = 'the';
>> z = 'bell';
>> [x, y, z]

ans =

ringthebell
```

However, this is not necessarily what you want. You may want instead to construct an array of strings, where each element of the array is a string (rather than a character). Such a collection of strings can be represented in Matlab as a **cell array**,

```
>> c = {'ring' 'offhook' 'end greeting' 'end message' 'absent'};
```

Notice the curly braces instead of the usual square braces. A cell array in Matlab is an array where the elements of the array are arbitrary Matlab objects (such as strings and arrays). Cell arrays are indexed like ordinary arrays, so

```
>> c(1)

ans =

    'ring'
```

Often, you wish to test a string to see whether it is equal to some string. You usually cannot compare strings or cells of a cell array using `==`, as illustrated here:

```
>> c = 'ring';
>> if (c == 'offhook') result = 1; end
??? Error using ==> ==
Array dimensions must match for binary array op.

>> c = {'ring' 'offhook' 'end greeting' 'end message' 'absent'};
>> if (c(1) == 'ring') result = 1; end
??? Error using ==> ==
Function '==' not defined for variables of class 'cell'.
```

Strings should instead be compared using `strcmp` or `switch` (see the on-line help for these commands).

M-files

In Matlab, you can save programs in a file and execute them from the command line. The file is called an **m-file**, and has a name of the form *command*.m, where *command* is the name of the command that you enter on the command line to execute the program.

You can use any text editor to create and edit m-files, but the one built into Matlab is probably the most convenient. To invoke it, select "New" and "M-file" under the "File" menu.

To execute your program, Matlab needs to know where to find your file. The simplest way to handle this is to make the current directory in Matlab the same as the directory storing the m-file. For example, if you put your file in the directory

```
    D:\users\eal
```

then the following will make the file visible to Matlab

```
>> cd D:\users\eal
>> pwd

ans =

D:\users\eal
```

The `cd` command instructs Matlab to change the current working directory. The `pwd` command returns the current working directory (probably the mnemonic is *present* working directory).

You can instruct Matlab to search through some sequence of directories for your m-files, so that they do not have to all be in the same directory. See `help path`. For example, instead of changing the current directory, you could type

```
path(path, 'D:\users\eal');
```

This command tells Matlab to search for functions wherever it was searching before (the first argument `path`) and also in the new directory.

Suppose you create a file called `hello.m` containing

```
% HELLO - Say hello.
disp('Hello');
```

The first line is a comment. The `disp` command displays its argument without displaying a variable name. On the command line, you can execute this

```
>> hello
Hello
```

Command names are not case sensitive, so `HELLO` is the same as `Hello` and `hello`. The comment in the file is used by Matlab to provide on-line help. Thus,

```
>> help hello

  HELLO - Say hello.
```

The M-file above is a program, not a function. There is no returned value. To define a function, use the `function` command in your m-file. For example, store the following in in a file `reverse.m`:

```
function result = reverse(argument)
% REVERSE - return the argument array reversed.
result = argument(length(argument):-1:1);
```

Then try:

```
>> reverse('hello')

ans =

olleh
```

The returned value is the string argument reversed.

A function can have any number of arguments and returned values. To define a function with two arguments, use the syntax

```
function [result1, result2] = myfunction(arg1, arg2)
```

and then assign values to `result1` and `result2` in the body of the file. To use such function, you must assign each of the return values to a variable as follows:

```
>> [r1, r2] = myfunction(a1, a2);
```

The names of the arguments and result variables are arbitrary.

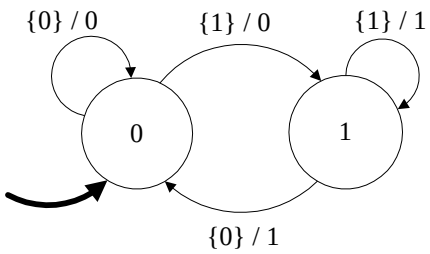
C.3.2 In-lab section

1. Write a for loop that counts the number of occurrences of `'a'` in

   ```
   >> d = {'a' 'b' 'a' 'a' 'b'};
   ```

   Then define a function `count` that counts the number of occurrences of `'a'` in any argument. How many occurrences are there in the following two examples?

   ```
   >> x = ['a', 'b', 'c', 'a', 'aa'];
   >> y = {'a', 'b', 'c', 'a', 'aa'};
   >> count(x)

   ans =

        ??

   >> count(y)

   ans =

        ??
   ```



Inputs = {0, 1, *absent*}
Outputs = {0, 1, *absent*}

Figure C.4: A simple state machine.

Why are they different?

2. The `input` function can be used to interactively query the user for input. Write a program that repeatedly asks the user for a string and then uses your `count` function to report the number of occurrences of `'a'` in the string. Write the program so that if the user enters `quit` or `exit`, the program exits, and otherwise, it asks for another input. **Hint**: Try `help while` and `help break`.

3. Consider the state machine in figure C.4. Construct an m-file containing a definition of its *update* function. Then construct an m-file containing a program that queries the user for an input, then if the input is in the input alphabet of the machine, uses it to react, and then asks the user for another input. If the input is not in the input alphabet, the program should assume the input is *absent* and stutter. Be sure that your update function handles stuttering.

C.3.3 Independent section

1. Design a virtual pet,[1] in this case a cat, by constructing a state machine, writing an *update* function, and writing a program to repeatedly execute the function, as in (3) above. The cat should behave as follows:

   > It starts out *happy*. If you *pet* it, it *purrs*. If you *feed* it, it *throws up*. If *time passes*, it gets *hungry* and *rubs* against your legs. If you feed it when it is hungry, it purrs and gets happy. If you pet it when it is hungry, it *bites* you. If time passes when it is hungry, it *dies*.

   The italicized phrases in this description should be elements in either the state space or the input or output alphabets. Give the input and output alphabets and a state transition diagram. Define the *update* function in Matlab, and write a program to execute the state machine until the user types 'quit' or 'exit.'

[1] This problem is inspired by the Tamagotchi virtual pet made by Bandai in Japan. Tamagotchi which translates as "cute little egg," were extremely popular in the late 1990's, and had behavior considerably more complex than that described in this exercise.

2. Construct a state machine that provides inputs to your virtual cat so that the cat never dies. In particular, your state machine should generate *time passes* and *feed* outputs in such a way that the cat never reaches the *dies* state.

   Note that this state machine does not have particularly meaningful inputs. You can let the input alphabet be

   $$\mathit{Inputs} = \{1, \mathit{absent}\}$$

   where an input of 1 indicates that the machine should output a non-stuttering output, and an input of *absent* means it should output a stuttering output.

   Write a program where your feeder state machine is composed in cascade with your cat state machine, and verify (experimentally) that the cat does not die. Your state machine should allow time to pass (by producing an infinite number of 'time passes' outputs) but should otherwise be as simple as possible.

   Note that a major point of this exercise is to show that systematically constructed state machines can be very easily composed.

   The feeder state machine is called an **open-loop controller** because it controls the pet without observing the output of the pet. For most practical systems, it is not possible to design an open-loop controller. The next lab explores closed-loop controllers.

Instructor Verification Sheet for C.3

Name: _____ Date: _____

1. Count the number of occurrences of `'a'`. Understand the difference between a cell array and an array.

   Instructor verification: _____

2. Write a program with an infinite loop and user input.

   Instructor verification: _____

3. Construct and use *update* function.

   Instructor verification: _____