# State Machines

There are many ways to implement the state machine concept in code. The essence is that the input is processed one item at a time, in order, with only a fixed amount of information (the "state") preserved from one item to the next. The iteration over the items may be performed by a loop or recursion; the state may be maintained explicitly in variables or hidden in an object, or it may be implicit in the current section of code being executed; the transition from one state to the next may be controlled by a series of conditional statements, a data structure representing the transition graph, or a function which encapsulates the required logic. Here are some examples:

## Strings containing "aeiou"

This is the example from Section 10.2 of the text. A string of lower-case letters will be accepted if it contains the vowels *a*, *e*, *i*, *o*, and *u*, in that order (the vowels may occur in other positions as well, as in "sacrilegious").

### State implicit in conditional statements

Here to start is a Scala version of Figure 10.2 from the text:

Show source

```
> def testWord(word: String): Boolean = {
|   var index = 0
|
|   /**
|    * Advance index through the characters of word until either
|    * c is found (return true; index points at character after c), or
|    * the end of the string is reached (return false).
|    */
|   def findChar(c: Char): Boolean = {
|     while (index < word.length && word(index) != c) {
|       index += 1
|     }
|
|     if (index == word.length) {
|       false
|     } else {
|       index += 1
|       true
|     }
|   }
|
|   // state 0
|   if (findChar('a'))
|     // state 1
|     if (findChar('e'))
|       // state 2
```

```
|       if (findChar('i'))
|         // state 3
|         if (findChar('o'))
|           // state 4
|           if (findChar('u'))
|             // state 5
|             return true
|   // error state
|   false
| }
testWord: (word: String)Boolean

> testWord("abstemious") // should be true
res17: Boolean = true

> testWord("sacrilegious") // should be true
res18: Boolean = true

> testWord("undercoating") // should be false -- not in order
res19: Boolean = false

> testWord("religious") // should be false -- no a
res20: Boolean = false

> testWord("aeiou") // should be true
res21: Boolean = true
```

The input is processed in this example by the *while* loop in the *findChar* function; the variable *index* is used to step through the characters in the word. The current "state" is reflected in how far the execution has progressed through the nested *if* statements in *testWord*; after reading characters for a while in state 0, it transitions to state 1 when the first *a* is seen, then to state 2 upon seeing a following *e*, etc. If any of the calls to *findChar* return false, meaning the end of the string has been reached while looking for one of the vowels, then the machine enters an error state. If all of the calls to *findChar* succeed, then the machine reaches state 5 and immediately returns *true* (without looking at the rest of the string).

## Integer state with transitions in a graph

Here is the same state machine, with the state explicitly represented by an integer in the range 0 to 5. The transitions are stored in an array of maps: trans(i) is the map, for state i, from the current character to the next state. Each of the maps in this case is particularly simple, since at most one edge leads away from each state to another; Scala allows a map to have a "default value", so that any transition not explicitly specified will go to that default state (here, remaining in the same state). See below for other examples using more complicated graphs.

Show source

```
> val trans = Array[Map[Char, Int]](
|   Map('a' -> 1) withDefaultValue 0, // state 0
|   Map('e' -> 2) withDefaultValue 1, // state 1
|   Map('i' -> 3) withDefaultValue 2, // state 2
|   Map('o' -> 4) withDefaultValue 3, // state 3
```

```
|   Map('u' -> 5) withDefaultValue 4, // state 4
|   Map() withDefaultValue 5 // state 5
| )
trans: Array[Map[Char,Int]] = Array(Map((a,1)), Map((e,2)), Map((i,3)),
Map((o,4)), Map((u,5)), Map())

> def testWord2(word: String): Boolean = {
|   var state = 0 // initial state
|   for (c <- word) {
|     state = trans(state)(c)
|   }
|   state == 5 // returns true if accepting state reached
| }
testWord2: (word: String)Boolean

> testWord2("abstemious") // should be true
res22: Boolean = true

> testWord2("sacrilegious") // should be true
res23: Boolean = true

> testWord2("undercoating") // should be false -- not in order
res24: Boolean = false

> testWord2("religious") // should be false -- no a
res25: Boolean = false

> testWord2("aeiou") // should be true
res26: Boolean = true
```

## Object-oriented state and transitions

Instead of assigning arbitrary numbers to the states, and collecting all of the transition information into a global graph data structure, a more object-oriented approach associates an object with each state:

Show source

```
> object TestWord {
|   trait State {
|     def trans(c: Char): State
|     def accept: Boolean = false // override to identify an accepting
state
|   }
|
|   object InitialState extends State {
|     def trans(c: Char) = if (c == 'a') AState else InitialState
|   }
|
|   object AState extends State {
|     def trans(c: Char) = if (c == 'e') AEState else AState
|   }
|
```

```
|   object AEState extends State {
|     def trans(c: Char) = if (c == 'i') AEIState else AEState
|   }
|
|   object AEIState extends State {
|     def trans(c: Char) = if (c == 'o') AEIOState else AEIState
|   }
|
|   object AEIOState extends State {
|     def trans(c: Char) = if (c == 'u') AEIOUState else AEIOState
|   }
|
|   object AEIOUState extends State {
|     def trans(c: Char) = AEIOUState
|     override def accept = true
|   }
|
|   def apply(word: String): Boolean = {
|     var state: State = InitialState
|     for (c <- word) {
|       state = state.trans(c)
|     }
|     state.accept
|   }
| }
defined module TestWord

> TestWord("abstemious") // should be true
res27: Boolean = true

> TestWord("sacrilegious") // should be true
res28: Boolean = true

> TestWord("undercoating") // should be false -- not in order
res29: Boolean = false

> TestWord("religious") // should be false -- no a
res30: Boolean = false

> TestWord("aeiou") // should be true
res31: Boolean = true
```

# Vending Machine

Suppose we want to model a vending machine which accepts nickels, dimes, and quarters, and dispenses a piece of candy when 25 cents has been deposited. If more than 25 cents is put in (for example, three dimes), then after dispensing the candy the remaining amount is applied toward the next transaction (that is, it doesn't give any change).

### Integer state with transitions in a graph

We will adopt a similar solution as for the second version of the vowel problem. This time, the integer state numbers will be more meaningful: they will be 0, 5, 10, 15, and 20, representing the amount of money which has been deposited so far. Since these are not consecutive, the graph will be represented by a map of maps instead of an array of maps. Also, there will be no need for default transitions, since each of the three possible inputs ('N', 'D', or 'Q', for nickel, dime, and quarter) will cause a different change of state. One further difference is that each edge in the graph will give not only a new state but also an indication of whether candy was given out on the transition. Finally, there is no accepting state, since the machine will keep running as long as there is input; in the example below, we will print "Candy!" whenever it produces a piece of candy.

Show source

```
> val vtrans = Map[Int, Map[Char, (Int, Boolean)]](
|    0 -> Map('N' ->  (5, false), 'D' -> (10, false), 'Q' -> (0, true)),
|    5 -> Map('N' -> (10, false), 'D' -> (15, false), 'Q' -> (5, true)),
|   10 -> Map('N' -> (15, false), 'D' -> (20, false), 'Q' -> (10, true)),
|   15 -> Map('N' -> (20, false), 'D' ->  (0, true),  'Q' -> (15, true)),
|   20 -> Map('N' ->  (0, true),  'D' ->  (5, true),  'Q' -> (20, true))
| )
vtrans: scala.collection.immutable.Map[Int,Map[Char,(Int, Boolean)]] =
Map((0,Map(N -> (5,false), D -> (10,false), Q -> (0,true))), (5,Map(N ->
(10,false), D -> (15,false), Q -> (5,true))), (10,Map(N -> (15,false), D
-> (20,false), Q -> (10,true))), (20,Map(N -> (0,true), D -> (5,true), Q
-> (20,true))), (15,Map(N -> (20,false), D -> (0,true), Q -> (15,true))))

> def vend(input: String) {
|   var state = 0 // initial state
|   for (coin <- input) {
|     val (newState, candy) = vtrans(state)(coin)
|     if (candy) println("Candy!")
|     state = newState
|   }
| }
vend: (input: String)Unit

> vend("NNNNQDNND") // Should print Candy! three times
Candy!
Candy!
Candy!
```

## Object-oriented approach with encapsulated state

Instead of exposing an explicit state, we can wrap it up inside an object with (mutable) internal state. Here is another approach to the vending machine, which also replaces the discrete transition graph with a calculated transition function.

Show source

```
> class VendingMachine(price: Int) {
|   private var balance = 0
|
```

```
|   /**
|    * Insert the given amount of money; returns true if an item is
vended.
|    */
|   def deposit(amount: Int): Boolean = {
|     balance += amount
|     if (balance >= price) {
|       balance -= price
|       true
|     } else {
|       false
|     }
|   }
| }
defined class VendingMachine

> def vend2(input: String) {
|   val machine = new VendingMachine(25)
|   for (coin <- input) {
|     val candy = coin match {
|       case 'N' => machine.deposit(5)
|       case 'D' => machine.deposit(10)
|       case 'Q' => machine.deposit(25)
|     }
|     if (candy) println("Candy!")
|   }
| }
vend2: (input: String)Unit

> vend2("NNNNQDNND") // Should print Candy! three times
Candy!
Candy!
Candy!
```