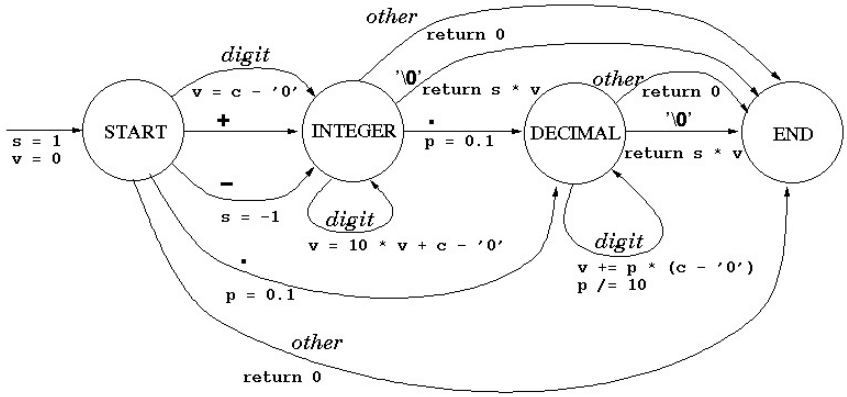
# Table Driven Methods

## 1 Background

We saw in class that the standard design pattern for implementations of state machines in object oriented languages has one big flaw: the behavior of the machine is scattered across many classes, so it is very difficult to compare the implementation to a diagram in a design document. In this lab, we are going to play with table driven methods as an alternative strategy for implementing state machines.

We will be building implementations of Mealy machines (where a behavior is associated with each edge - as opposed to Moore machines where a behavior is associated with each state). In particular, we are going to build this machine:[1]




Input symbols are written in **bold** above the transitions,
Symbol classes (such as *digit*) are written in *italic*.
Actions performed by the program are in `courier` below the transitions.
The variable `c` denotes the input character.

This machine is taking a string that represents a real number and calculating the value of the represented number. This is a standard strategy for how to solve this problem.

[1] Gehringer, Edward, http://people.engr.ncsu.edu/efg/210/s99/Notes/fsm/ retrieved on 4/6/2021.

## 2 Task #1 Table Driven Java Implementation

Create a NEW Java project for this part. Our goal is to build a table driven implementation of this machine. The table will contain representations of every edge in this machine. In particular, each edge is defined by:

- The current state (the source of the arrow)
- The input we consume by following the edge
- The action associated with following the edge
- The next state (the terminus of the arrow)

We will then use that table to drive an algorithm that will perform the appropriate calculation.

Note: I am giving you pieces of this code: Step2StartingCode.zip. My recommendation is that you do not put them in your project until you get to their descriptions in these instructions. Before that point, they will just clutter things with syntax errors.

For the this implementation, we will make three small changes to the diagram:

- we don't really need an "End" state - we can just return the result on the transition to that state
- the end of string will not be '\0' - our loop that parses the input just has to go to the length of the string and then return s∗v as the transition to the final state.
- if there is a poorly formatted input, we will throw NumberFormatException (yes, you had better have lots of tests for this)

### 2.1 Storing the Machine

We are going to store the machine in a table of edges. The hard thing about that is two of those things are behaviors (something that checks the input to see if it matches and something that is the action on the edge). For each of those, we will build those things much like the strategy pattern: we will build an interface that all of the behaviors must implement and then concrete classes that implement that interface for each required behavior. I have given you the interfaces for these (InputVerifier and Action).

With those in place, we can build our converting machine. I have given you the beginnings of the class to show you how the table can be encoded (ConvertingMachine). Be sure you understand how the table is constructed and check to make sure that it matches our diagram. If you have any questions, ASK!!!!

In order to get ConvertingMachine to compile, you are first going to have to build the concrete classes for every verifier and action you need. The verifiers just need to return a boolean reflecting whether the given character meets the criteria that verifier is checking for. Be sure to write complete tests for these (and everything else). The actions require a little more work (but, don't worry, I did it for you). We need the actions to return the result of their calculations, but that result contains three pieces of information (p, s, and v). Therefore, we need a class to encapsulate that so we can return it. I have given you a class that encapsulates those, InterimResult and its tests. Be SURE you understand what it contains and what the test does. When you have built (and tested) all of the verifiers and actions, ConvertingMachine should compile.

All that is left is to build the method that actually runs the machine. It must walk through a given string. At each character, it must search for the appropriate edge (matching the current state and the input). When that edge is found, execute the action on the edge and change the current state to the next state. I found it easier to make a method that searched for an edge and then use that within the loop that walked through the string. If no edge matches, you throw a NumberFormatException because the input is invalid.

With all of that complete, you should be finished. I have given you ConvertingMachineTest that verifies that every edge of the graph is encoded correctly.

## 3 Task #2 The C Implementation

Having completed the Java implementation, the C implementation is really just a language change. The primary difference is that we won't use separate objects as the input verifiers and the actions (since we don't have objects!). Instead, we will use function pointers. Follow the example I gave you in class. In addition, the C version doesn't throw an exception. It just returns 0 if the input is invalid. Essentially, you are building the C90 implementation of the atof function that already exists in C and it returns 0.0 on invalid input (perhaps you should read about that . . .)

While there are TDD tools for C, I won't require a full set of tests. Feel free to create a main method that demonstrates that all of the cases we tested in the Java version work in the C version.

## 4 Submission

You must submit a screen shot that demonstrates that each of your solutions works and meets the requirements I gave (show that you implemented it using the required technique). Gradescope has questions about each implementation that I will grade.

In addition, to make sure you really understand, you should be able to answer these questions.

1. What do you have to do to be sure that the table driven implementation matches the state diagram?
2. What do you have to do to add a state to the table driven java implementation?
3. What do you have to do to add a state to the table driven C implementation?