A finite state machine (FSMs) is a theoretical model for a computing system capable of recognizing *regular languages*, which we can informally describe as the set of languages expressible with regular expressions. In this lab we will not delve into the more theoretical aspects of FSMs, and will simply provide an abstract description of what an FSM is and how it functions. You will write a Python program that simulates a FSM (Exercise 1), then re-implement it in C (Exercise 2). The goal of this lab is for you to take an abstract description of a process and then "translate" it into a programming language.

Finite State Machines

A FSM is composed of the following:

- A set of *states*, including:
  - A start state
  - A set of accepting states (or "end" states)
- A set of *input symbols*.
- A set of transitions between states or, more formally, a transition function that, given a state and an input symbol, returns the next state the machine must transition to.

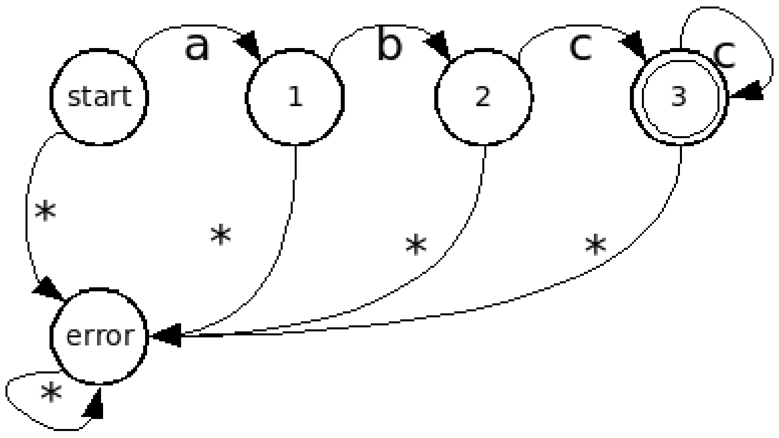
Given a string of input symbols, the FSM starts with the start state as its *current state*, reads in the first input symbol, and then update the current state with the state returned by the transition function. It continues to do this until no more symbols remain in the input string.

For simplicity, we make the following assumptions:

- *The FSM functions as a sequence detector*. In other words, we are only interested in checking whether, after processing the input string, the current state is an accepting state. (Compare with Moore and Mealy FSMs, where the purpose of the FSM is to produce an output in each state or transition, and the final state is generally irrelevant.)
- *The FSM is deterministic*. The transition function maps each unique (state,symbol) pair to a single state. (Compare with nondeterministic, where an input symbol could result in a transition to more than one state at once.)
- Our alphabet of input symbols is the set of lowercase alphabetic characters (a-z)

Example

Consider the following FSM:



This FSM has the following components:

- States: start, 1, 2, 3, error
  - Start state: start
  - Accepting state: 3 (denoted by a double circle)
- Input symbols: a-z. We use the character '*' to denote all other characters not captured by other transitions originating in this state. Note that this is just syntactic sugar and an implementation would need to specify all possible transitions.
- Transition function:
  - (start,a) -> 1
  - (start,*) -> error
  - (1,b) -> 2
  - (1,*) -> error
  - (2,c) -> 3
  - (2,*) -> error
  - (3,c) -> 3
  - (3,*) -> error
  - (error,*) -> error

By inspection, we can easily see that the above FSM corresponds to the regular expression “abc+”.

FSM file format

For this lab, we will specify FSMs using text files. For example, the following corresponds to the above FSM:

```
[states]
names: start,1,2,3,error
start: start
end: 3

[transitions]
start:  a:1 , *:error
1:      b:2 , *:error
2:      c:3 , *:error
3:      c:3 , *:error
error:  *:error
```

You are not provided with a formal specification of this format, and should be able to infer it simply by looking at the provided example files provided. Your code will be considered valid if it can correctly read these files.

Exercise 1: Python FSM simulator

You are to write a Python program that simulates a FSM, such that:

- The script's name is **`fsm.py`**
- The script accepts two command-line parameters: the FSM file and an input string. For example:

```
fsm.py example1.fsm abccccccc
```

- If the FSM ends in an accepting state, the script prints out "Accept". Otherwise, it prints out "Reject. Ended in state XX" (where XX is the current state when the FSM finishes reading the input string).
- The script should read in the FSM file at the beginning of the program and load its contents into whatever data structures you find appropriate (lists, dictionaries, etc.). Your program must *not* read the file during the simulation itself.
- The simulator should detect the following error conditions:
    - The FSM file specifies a nondeterministic automaton.
    - The FSM file does not provide an exhaustive transition function (i.e., there are state+input combinations for which the FSM would not have a transition to follow). For simplicity, you are only required to detect this if the FSM is given an input string that is affected by this missing transition.

Your simulator will be evaluated based on whether it can do the following:

- Reading in the FSM file, and storing its contents using Python data structures. (*Note*: You can read the FSM file line-by-line and manually parsing each line. However, there is a much easier way of reading the FSM file which will save you a lot of time.)

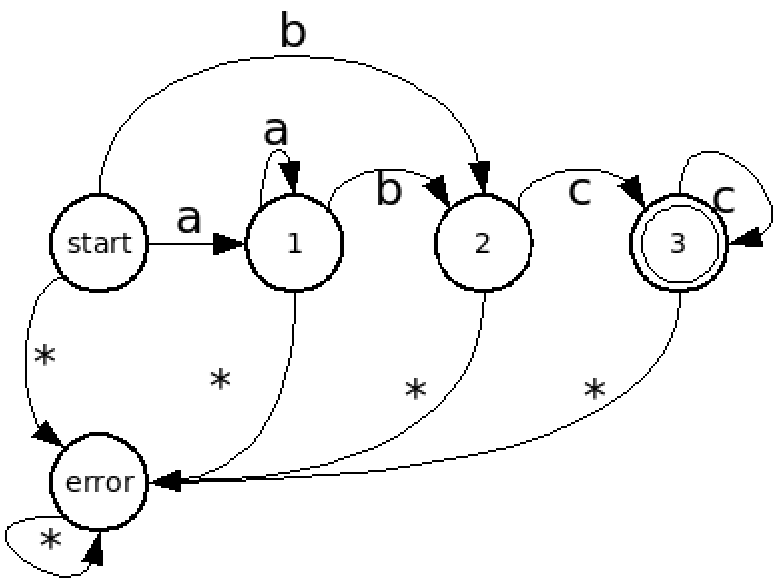
- Simulating the FSM.
- Detecting the error conditions.

Example files

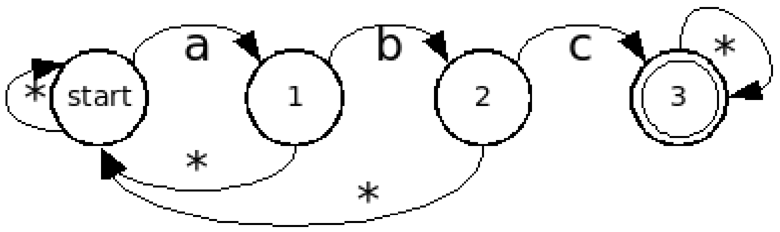
You are provided with five example files.

`example1.fsm` corresponds to the FSM shown earlier.

`example2.fsm` corresponds to the following FSM, equivalent to regular expression "`a+bc+`":



`example3.fsm` corresponds to the following FSM, equivalent to regular expression "`[a-z]*abc+[a-z]*`"



`example4.fsm` is a modification of **`example3.fsm`** to make it non-exhaustive (your implementation must detect this with input strings such as "abe" and "abu").

`example5.fsm` is a modification of **`example3.fsm`** to make it nondeterministic (your implementation must detect this).

Exercise 2: C implementation

Implement an FSM simulator in C satisfying the same conditions as in Exercise 1, except that it does *not* need to detect the error conditions. That is, you may assume that the FSM file specifies a deterministic automaton, and provides an exhaustive transition function. Thus, your simulator only needs to be able to handle **`example{1,2,3}.fsm`** (and not **`example{4,5}.fsm`**).

Your makefile should compile your program to a binary called **`fsm`**, and should take the same command-line parameters as in Exercise 1. For example:

```
./fsm example1.fsm abccccccc
```

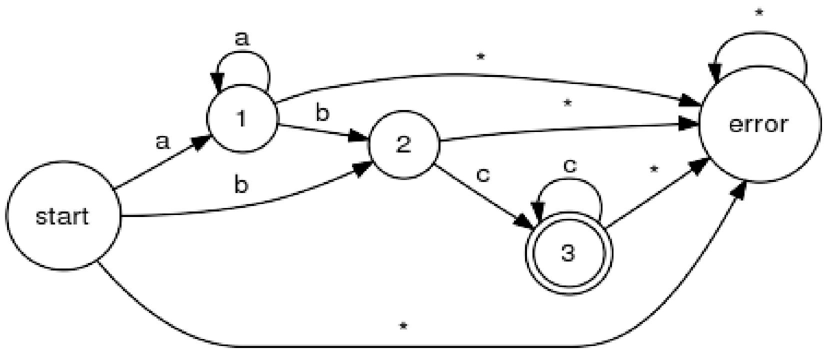
You are encouraged to represent the FSM being simulated using appropriate types, defined using struct. For example, a node could be represented by a struct with fields for whether it is an accept state and the set of transitions to other states.

Extra credit

The above FSM diagrams are ugly and unpleasant. They would look much nicer if we used graph visualization software instead of drawing them manually. In particular, we could use the tools provided by the Graphviz project (http://www.graphviz.org/) to generate better diagrams. You are asked to do the following:

- Read about the Graphviz project and, in particular, the dot command (available in the CS machines).
- Write a script **`fsm2dot.py`** that creates a dot file based on a FSM file.
- Write a simple wrapper script **`fsm2png.sh`** that takes a FSM file, converts it to a dot file (using `fsm2dot`), and then generates a PNG file using dot.

For example, the diagram for example2.fsm would look something like this:



Documentation

Make sure your code is adequately commented.

- For each function (in C and Python), explain its purpose, parameters, and return value(s).
- Describe each part of any structs (in C) or classes (in Python) you create.
- Any non-obvious code within a function or struct/class definition should also be commented.

Using **`make`**

When submitting your code, include a Makefile to compile your code for Exercise 2 to a binary called **`fsm`**.