The University of Chicago
Department of Computer Science

CMSC 15200 – Introduction to Computer Science 2
Summer Quarter 2007
Homework #8 (08/17/2007)
Due: 08/22/2007 @ 1:30pm

Name:

Student ID:

Instructor: Borja Sotomayor

Do not write in this area

1	2	3	**TOTAL**

Maximum possible points: 20 + 40

Exercise 1 <<20 points>>

You are provided with an XML file with information about products in the store. This is a sample file:

```
<?xml version="1.0"?>
<shop name="Uncle SNAFU's Computer Store">
       <item id="1" stock="7" discounted="y">
              <name>FOOBAR keyboard</name>
              <price>39.99</price>
       </item>
       <item id="2" stock="15" discounted="y">
              <name>Grouchobyte hard drive</name>
              <price>149.99</price>
       </item>
       <item id="3" stock="81">
              <name>Wumpus repellant</name>
              <price>49.99</price>
       </item>
</shop>
```

For each product, the following information is stored:

- *Product ID*: Each product is assigned a unique id.
- *Items in stock*: The number of products immediately available in the store.
- *Name*: The product's name.
- *Price*: The product's price.
- *Discounted product*: Indicates whether this product is currently offered at a discounted price (-15% discount).

Write a Python program that reads an inventory file and determines the total value of all the items in stock. In other words, for each product, multiply its price by the stock (applying a discount if necessary), and sum all the values. The program's output should be the total value.

The output for the above sample file should be:

```
6199.50
```

The output for the provided **`inventory.xml`** file should be:

```
34202.42
```

Exercise 2 <<Extra credit: 10 points>>

Write a postfix notation (also called Reverse Polish Notation) arithmetic evaluator. Your program must ask the user for a postfix expression, evaluate it, and then allow the user to write another expression. The program will run infinitely (i.e., don't include a "Do you want to enter another expression?" prompt).

For example:

```
borja@classes:~$ python2.5 eval.py
Type your expression: 3 5 +
8
Type your expression: 2 3 + 7 *
35
Type your expression: 2 3 + 4 5 + *
45
```

You are allowed to assume that operands and operators are separated by a single space. You only need to support the addition, subtraction, multiplication, and division operators.

How to evaluate a postfix expression was (briefly) discussed in class. Hint: You will need to use a stack.

Exercise 3 <<Extra credit: 30 points>>

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 then have to write a Python program that simulates a FSM. The goal of this exercise 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.

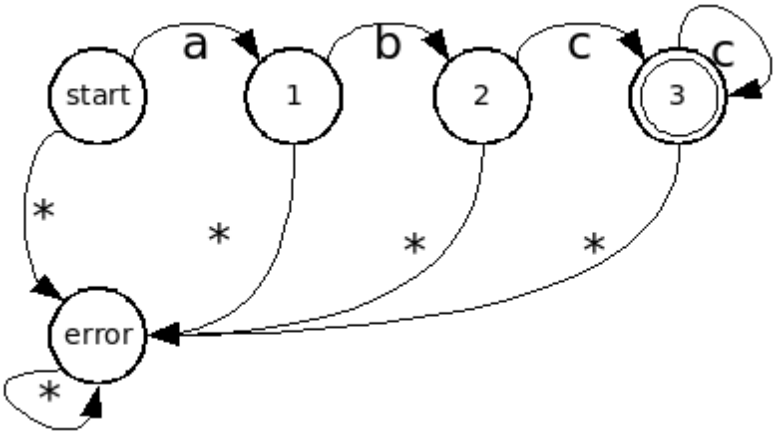
Given a string of input symbols, the FSM will start with the start state as its *current state*, will read in the first input symbol, and then update the current state with the state returned by the transition function. It continues to do this until no more symbols remain in the input string.

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 we would strictly 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 exercise, 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.

FSM simulator

You must write a Python program that simulates a FSM, with the following characteristics:

- The script's name will be **`fsm.py`**
- The script will accept 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 will write out "Accept". Otherwise, it will write out "Reject. Ended in state XX" (where XX will be the current state at the time when the FSM finished reading the input string).
- You must 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 must detect the following error conditions:
  - The FSM file specifies a nondeterministic automata.
  - 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.

The points for this exercise are the following:

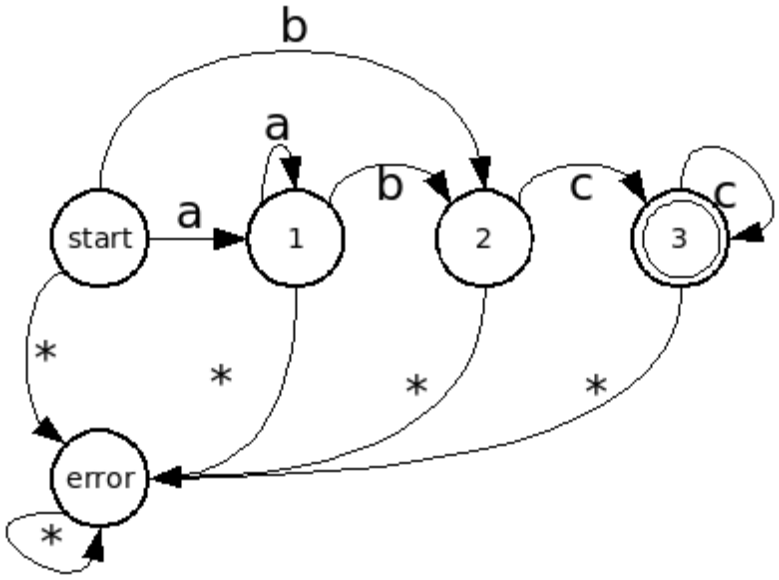
- <<10 points>> Reading in the FSM file, and storing its contents using Python data structures. *Note*: You can read the FSM file line-by-line and manually parsing each line. However, there is a *much easier* way of reading the FSM file which will save you a lot of time.
- <<10 points>> Simulating the FSM
- <<10 points>> Detecting the error conditions

Example files

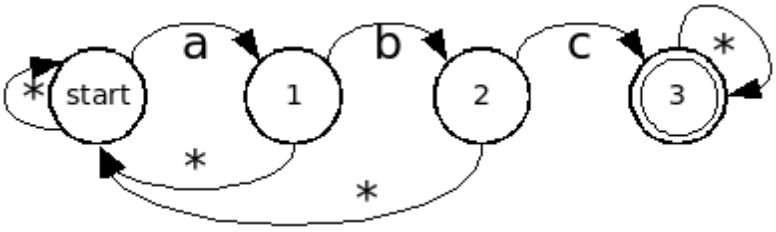
You are provided with five example files.

`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`** so its transition table will be 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)