Informatics 1 - Computation & Logic: Tutorial 6

Computation: Introduction to Finite State Machines

Week 8: 6 – 10 November 2017

Please attempt the entire worksheet in advance of the tutorial, and bring with you all work, including (if a computer is involved) printouts of code and test results. Tutorials cannot function properly unless you do the work in advance.

You may work with others, but you must understand the work; you can't phone a friend during the exam.

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is **obligatory**; please let your tutor know if you cannot attend.

A *finite state machine (FSM)*, also called a *finite state automaton (FSA)* is an abstract model of computation that can be used to design sequential logic circuits and computer programs. For example, FSMs can represent the behaviour of devices such as vending machines, elevators or traffic lights. They may take as input, for example characters from an alphabet, numbers or more abstract symbols, e.g. coins or button presses.

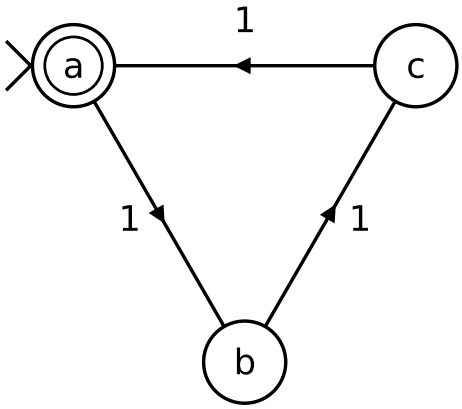
Finite state machines are often represented as a graph. Circles represent the states in which the machine can be. There is a finite number of states and the machine can only be in one state at a time (this is called the *current state*). A change from one state to another is called a *transition*. Possible transitions are represented by arrows on the graph. If a transition is triggered by particular input to the system, it is represented as an annotation next to the arrow. The state in which the machine starts operation is called the *initial state* and in this tutorial it is represented by pointing to it with an arrowhead. One or more of the states can be marked as an *accepting state*. This is a state which should be reached after processing an input sequence that the machine is designed to recognize as 'valid', for example a state in which an appropriate amount of money was inserted into a vending machine, which should then dispense a drink. If an input sequence causes the machine to end in an accepting state, we say that input sequence is accepted, else it is rejected.

For more information you can refer to the Wikipedia article on FSMs, on which this introduction was based.

You may find it useful to refer to the FSM Workbench question set which accompanies this tutorial at
homepages.inf.ed.ac.uk/s1020995/tutorial5 (the URL is correct — this exercise was last year's tutorial 5.

The link will take you to FSM simulations corresponding to each question. You can also use this site to build FSMs on your own and experiment with various inputs. A brief reference card is given at the end of this document.

1. Consider this finite state machine.



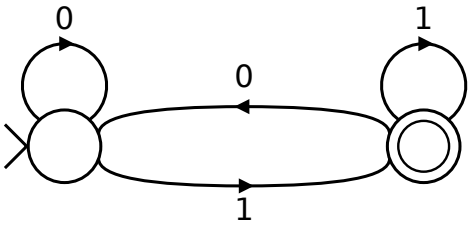
   (a) The machine accepts the sequence '111'. Give two other sequences that it will accept. ⟨⟩ (empty string), '111111', '111111111'

   (b) What do the sequences that this machine accepts have in common? All accepted strings consist of $n\times$'1', where n mod 3 = 0.

   (c) When will this machine be in state **c**? The machine will be in state **c** when n mod 3 = 2.

2. Like decimal numbers, binary numbers can be sorted into odd and even by looking at only the least significant digit. For example 12 = 1100 is even, 9 = 1001 is odd.

   (a) Design a finite state machine over the alphabet {0,1} which accepts only those sequences that form an odd binary number.



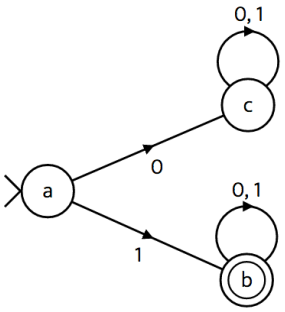
   (b) For each input sequence in the table below, record whether the sequence is accepted or rejected by your machine.

Input	Is Accepted?
0	N
1	Y
0001	Y
1111	Y
001010	N
110101	Y
$\langle\rangle$	N

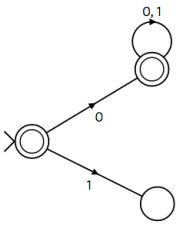
(c) What changes would you make to your machine to make recognise even numbers? Make the start state the only accepting state.

(d) What changes would you make to your machine to recognise the same sets for binary numbers presented in reverse order, with the least significant bit coming first?

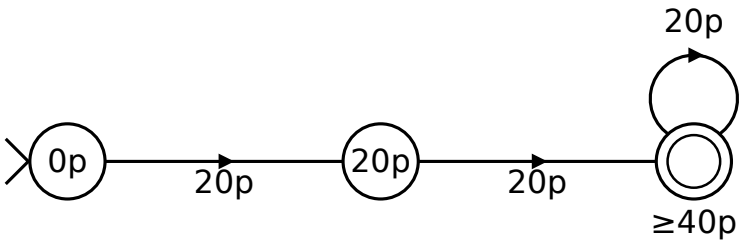
Odd



Even

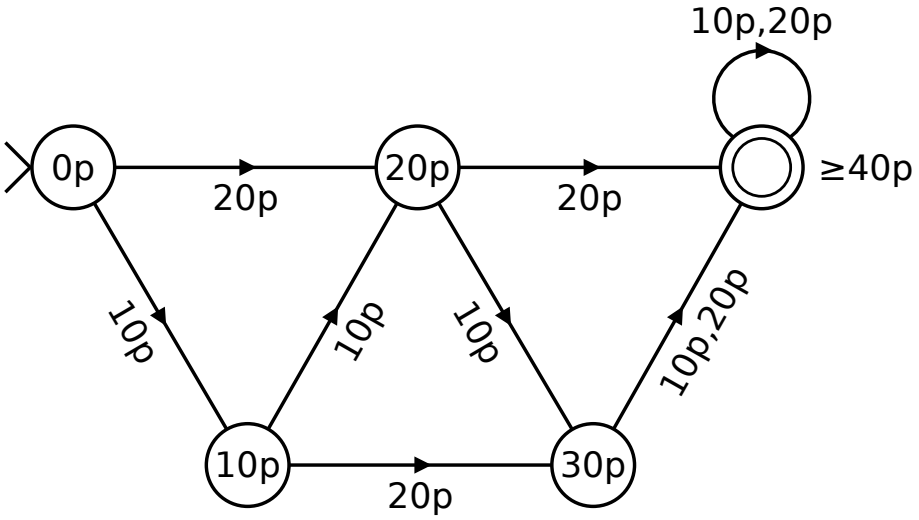


3. This finite state machine could be used as part of a vending machine. It accepts any sequence of 20p coins that add up to 40p or more.

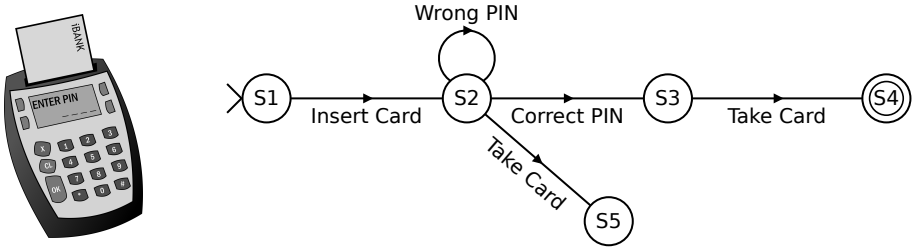


(a) How does this machine deal with input of more than 40p? After 40p has been reached, more money can be input but it will not alter the recorded total

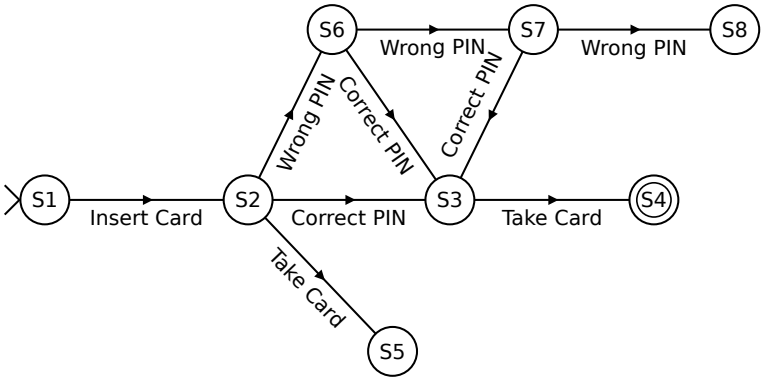
(b) Modify the machine to also allow 10p coins.



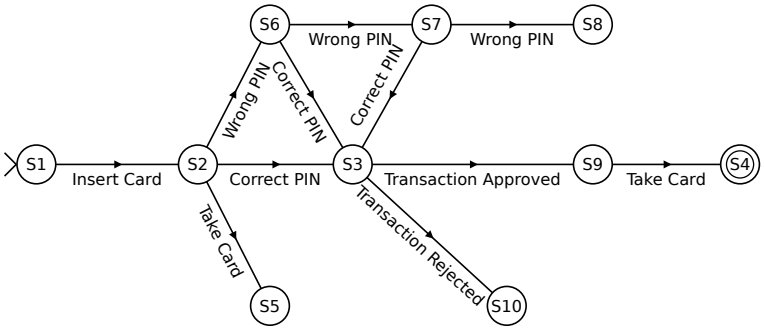
4. The FSM below models the control logic of a chip & PIN card payment terminal. It represents a single transaction, ending in an accepting state if the transaction is successful.



(a) Note that the transaction will fail if the card is removed too early. Modify the machine so that the transaction will also fail if the wrong PIN is entered three times.



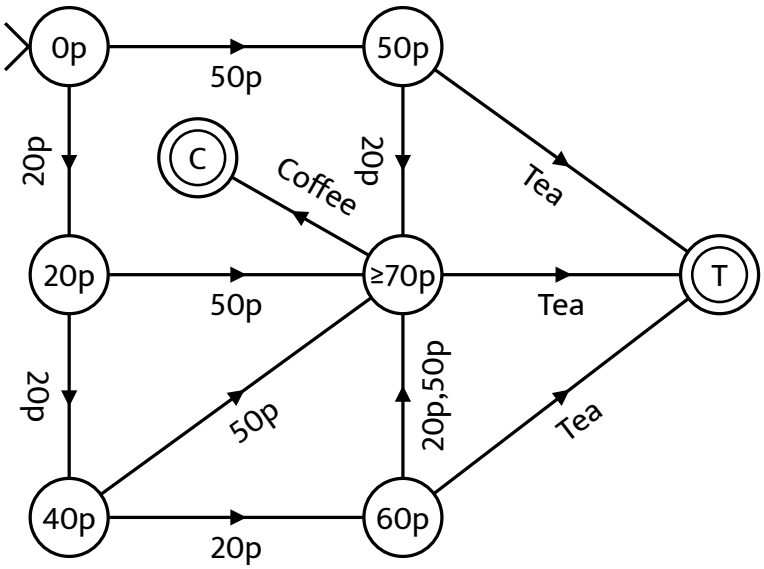
(b) This machine only verifies the PIN. Modify it to represent the process of checking with the bank for approval. The machine should only accept if the transaction is approved. The modified machine should take inputs 'Transaction Approved' and 'Transaction Rejected'.



It could be argued that there should be 'Take Card' transitions to S5 from every state other than S1, S5, and S9. As shown, the behaviour of the machine is undefined if the card is removed early.

5. Consider a hot drinks machine. The machine takes 20p and 50p coins. It sells tea for 50p and coffee for 70p.

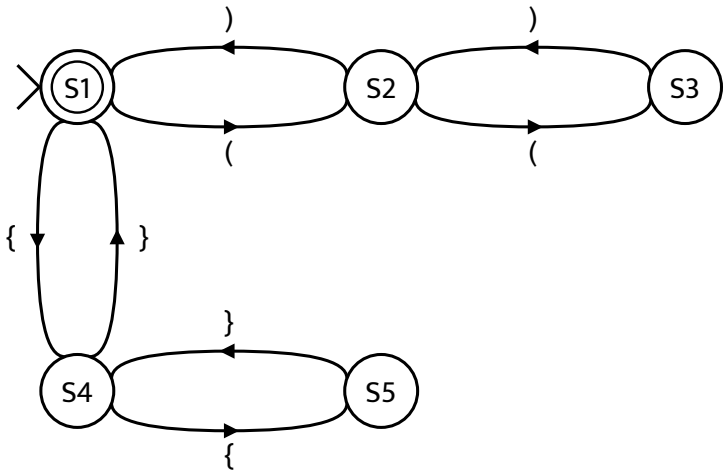
   (a) Design a FSM that could be used to control this machine. After a successful sale the FSM should be in an accepting state. The machine only needs to model a single transaction.



   The label on the transition from the $50p$ state to the $\geq 70p$ state could also allow a $50p$ input.

   (b) Consider your answer to part (a). How does your machine handle over-payment? Would it be possible to design a FSM that gives correct change? The machine ignores payment over 70p. No finite state machine can give correct change for all possible inputs, as this requires the machine to count arbitrarily high. However, by adding more states it would be possible to give change up to some fixed limit.

6. The designer of the this machine was attempting to create a system to accept strings with matching opening and closing brackets.



(a) Give a sequence of matching brackets that the machine does not accept. `((()))` or `{{{}}}` or `{()}`.

(b) What does each state of the machine represent? That is, for each state what do the sequences that end in that state have in common?
These are hard to describe with regular expressions, as parentheses are special characters. In the next two bulleted paragraphs, we use `( ) { }` as literals and use `[ ]` to group regular expressions

- S1 – We introduce two patterns of matched parentheses to describe the two branches of the FSM.
- Let $P = \texttt{[([()]*)]}^*$ be a pattern for strings containing only `( )`, and let Q be the corresponding pattern for strings using `{ }`, that is, $Q = \texttt{[{[{}]*}]}^*$

S1 accepts the language $L_1 = (P|Q)^*$.
S2 accepts L_1 followed by and extra `(`, plus any number of `( )` pairs.
S3 – accepts any string accepted by S2, plus an extra `(`.
S4 – is similar to S2. It accepts a string in L_1 plus an extra `{`, plus any number of `{ }` pairs.
S5 – accepts any string accepted by S4, plus an extra `{`.

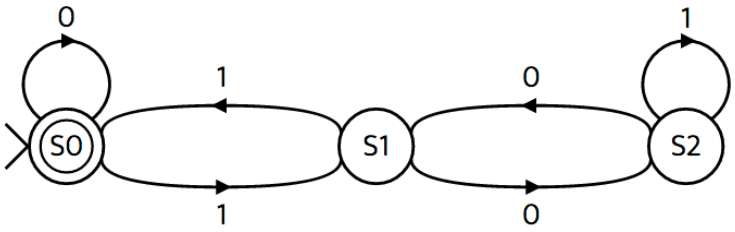
(c) Is it possible to design a finite state machine that will accept all possible sequences of matching brackets? Justify your answer.
No – this requires counting the number of open brackets. To do this for all possible inputs would require an infinite number of states.

(d) For each state in the machine, there are input symbols which do not correspond to any transition. What do you think the machine should do if it received input that did not correspond to a transition? There are (at least) two conventions for 'missing inputs'. When we are modelling systems such as a coffee machine, we often use the convention that a missing transition signifies that the corresponding action that cannot happen.
When we consider NFA the definition of a trace means that we interpret missing transitions differently. If we construct the corresponding DFA using the subset construction, then the missing transition is replaced by a transition to the empty set, which corresponds to a 'black hole' state, from which there is no escape.

The ambiguity of how to handle unexpected input arises as it is often undesirable to draw transitions for all possible inputs to each state, as this can make diagrams difficult to read. For machines modelling control systems, such as the vending machine in question 5, the convention is for machines to remain in their current state when they receive unexpected input. For string processing applications, such as bracket matching, the convention is to reject sequences that contain unexpected input by moving to a black hole state.

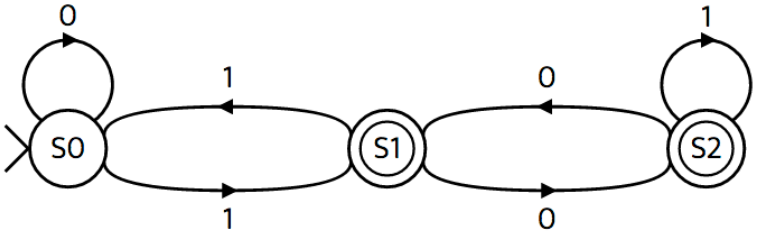
7. This is a variation on Question 2.

   (a) Design a finite state machine over the alphabet 0,1 which accepts only those sequences that form a binary number divisible by 3.
   For example your machine should accept $0, 000, 011, 11, 110, 1001$, and 1100, which represent $0, 0, 3, 3, 6, 9$ and 12. Hint: design a machine with three states s0, s1, s2, where the machine is in state s2 whenever $n \bmod 3 = 2$, i.e. where n, the number represented by the binary sequence seen, is 2 more than some multiple of 3.
   If a binary string s represents the number n, then $s1$ represents $2n + 1$, while $s0$ represents $2n$. The three states keep track of $n \bmod 3$.



   Answer:

   (b) What changes would you make to your machine to make recognise numbers not divisible by 3?



   Answer:

   (c) What changes would you make to your machine to recognise the numbers divisible by 3 for binary numbers presented in reverse order, i.e. with the least significant bit coming first?
   The surprising answer is that we don't need to change anything. The machine for recognising 0 mod 3 can run backwards as well as forwards — if we reverse the direction of every arrow we get the same machine. This means that if the machine accepts a given binary string then it also accepts that string in reverse order. So, we have discovered that a binary string represents a number divisible by 3 iff its reverse represents a number divisible by 3!
   We can see this algebraically.[1] If n is even, then $2x^n = 1 \mod 3$, while if n is odd then $2x^n = -1 \bmod 3$. The value v is the sum

[1] Thanks to Andrew Ranicki for this argument.

of these contributions $\mod 3$. Reversing a string of odd length takes odd bits to oddd bits, and even to even. Reversing a string of even length swaps odd and even bits, so changes the sign of each contribution. In either case, if the sum is 0 mod 3 it is unchanged by the reversal.

Understanding the operation of the machine on the reversed string is more complex. Adding a zero at the start of a binary number doesn't change its value, v. However, the effect of adding a 1 to a string of length n is to move from v to $v + 2^n$. So if we add two successive 1s then $v \mod 3$ doesn't change.

Using these observations, you can check that the machine is:

- in $S0$ when $v = 0 \mod 3$
- in $S1$ when either n is even and $v = -1 \mod 3$, or n is odd and $v = 1 \mod 3$;
- in $S2$ when either n is even and $v = 1 \mod 3$, or n is odd and $v = -1 \mod 3$;

Tutorial Activity

1. Use the FSM workbench
   http://homepages.inf.ed.ac.uk/s1020995/fsmworkbench/create.html
   to build machines to recognise the following languages. For each machine give a set of test strings that you use to check your design.

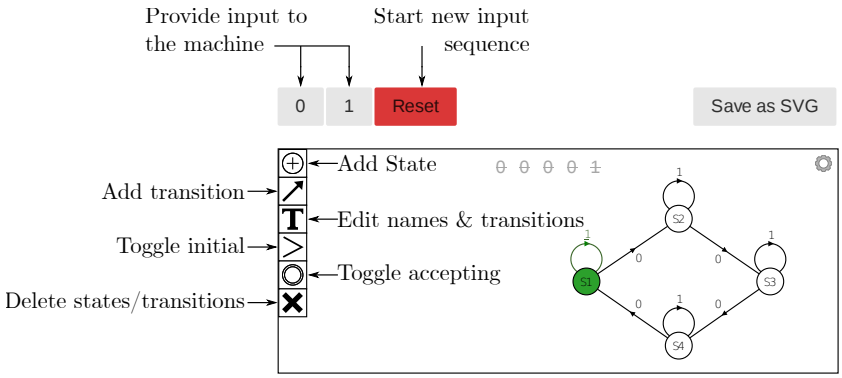
   - Binary numerals that represent numbers divisible by 5.
     Save the svg as `base2mod5`.
   - Binary numerals that represent numbers divisible by 7. Save the svg as `base2mod7`.
   - Create more examples, such as `base4mod5`, `base3mod7`, `base6mod7`, ...

2. Can you say anything special about base n mod $n + 1$?

Using the FSM Workbench

The workbench provides tools for creating, editing, and simulating finite state machines. The diagram shows the function of each tool.

You can toggle each tool on/off by clicking it. When no tools are active you can drag the states of your FSM to rearrange the layout.



This tutorial exercise sheet was written by Matthew Hepburn and Dagmara Niklasiewicz, with additions from Michael Fourman. Send comments to `Michael.Fourman@ed.ac.uk`