

Inf2A: Review of FSMs and Regular Languages

Stuart Anderson

School of Informatics
University of Edinburgh

October 6, 2009

Outline

- 1 Review of Finite State Machines
 - Finite State Machines
 - Operations on Regular Languages
 - Finite State Machines and Regular Grammars

- 2 Characterising Regular Languages
 - Looping Behaviour

Road map for the Formal Aspects of the Course

- The four classes of language and their corresponding grammars and machines:
 - Type 3 languages: Finite Automata, Regular Grammars, Regular Expressions
 - Type 2 languages: Pushdown automata, Context-Free Grammars
 - Type 1 languages: Linear-bounded automata, Context Sensitive Grammars
 - Type 0 languages: Turing machines, Unrestricted Grammars

Questions about languages

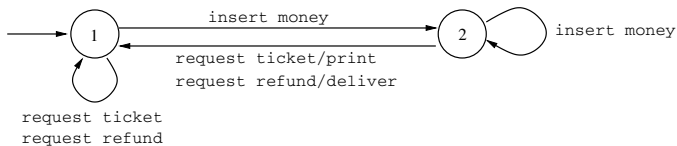
- For each of these four classes we study:
 - Methods for deciding if a string is a member of a language.
 - Designing particular grammars or machines to describe a particular language.
 - Exploring the difference between deterministic and non-deterministic machines for each class.
 - Seeing if we can decide if two machines or grammar recognize/describe the same language.
 - Seeing if we can calculate the effects of operations on languages e.g. intersection, complement, ...
 - Looking at how to characterize the power of language defining mechanisms.
 - Looking at probabilistic versions of languages.

Acceptors and Transducers

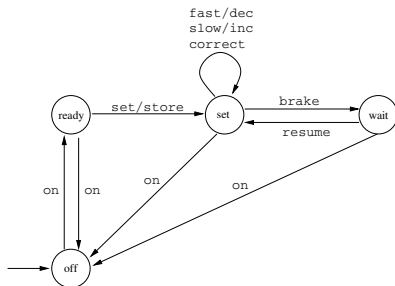
- In Inf1 we saw FSMs as descriptions of *reactive systems*.
- We describe them as a set of *states* together with *transitions* between states on receipt of an input.
- Can be either *transducers* – providing output for each input, or *acceptors* – accepting some sequences, rejecting others.

Examples of Transducers

- A parking ticket machine:

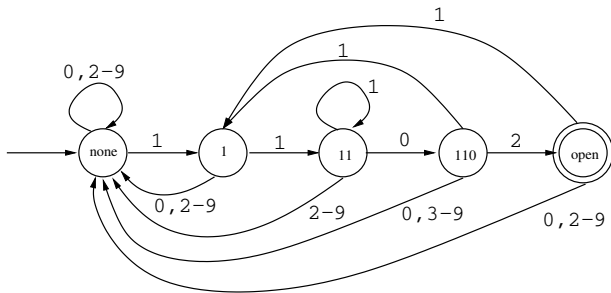


- A cruise control system:



Example of an Acceptor

- A combination lock:



- In Inf2 we mostly focus on the role of FSMs as acceptors.
- You can look on the language accepted as the collection of all possible behaviours.
- Is the sentence 1110021 accepted by the lock?

Formal Languages

- Finite *alphabet* Σ .
- Σ^* is the set of all strings with symbols drawn from Σ . Σ^* contains *empty string* ε .
- ε is the identity for *concatenation* of strings, i.e.
 $\varepsilon s = s\varepsilon = s$ for all $s \in \Sigma^*$
- A *language* over Σ is a subset of Σ^* .

Example: Binary Strings

- Alphabet: $\Sigma = \{0, 1\}$
- Strings: $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, \dots\}$
- Languages:

$$L_1 = \{\varepsilon, 0110, 1101, 111110\}$$

$$L_2 = \{0x \mid x \in \Sigma^*\} \quad (\text{strings starting with } 0)$$

$$L_3 = \emptyset \quad (\text{the empty language})$$

$$L_4 = \{\varepsilon\}$$

Example: ASCII Strings

- Alphabet: Σ = set of all ASCII symbols
- Strings: all ASCII strings, for example,

$$x_1 = \varepsilon$$

$$x_2 = \text{soa@inf.ed.ac.uk}$$

$$x_3 = \text{fac = foldr (*) 1 . enumFromTo 1}$$

- Languages:

$$L_1 = \{x_1, x_2, x_3\}$$

$$L_2 = \text{all syntactically correct JAVA programs}$$

$$L_3 = \text{all correct English sentences}$$

$$L_4 = \emptyset$$

Finite Automata as Language Acceptors

Σ alphabet

M finite automaton (acceptor) whose inputs are symbols from the alphabet Σ

- M accepts a string $a_1 a_2 \dots a_n \in \Sigma^*$ if after “reading” inputs a_1, a_2, \dots, a_n it is in an accepting state.
- M recognises (or accepts) a language $L \subseteq \Sigma^*$ if for all strings $x \in \Sigma^*$:

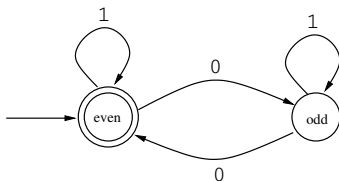
$$x \in L \iff M \text{ accepts } x.$$

Example: Binary Strings with an Even Number of Zeroes

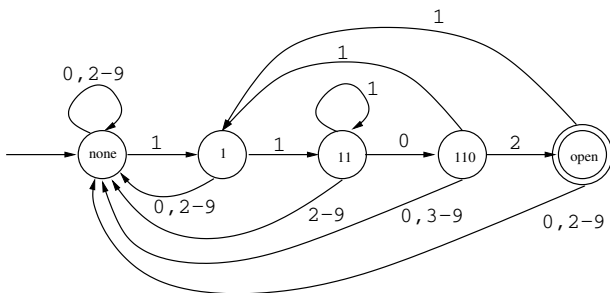
The following automaton recognises the language

$$\{x \in \{0, 1\}^* \mid x \text{ contains an even number of } 0\text{s}\}$$

over the alphabet $\{0, 1\}$.



Example: The Combination Lock Revisited



This automaton accepts the language

$$\{x1102 \mid x \in \{0, 1, \dots, 9\}^*\}$$

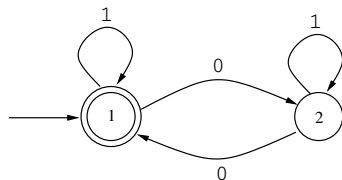
over the alphabet $\{0, 1, \dots, 9\}$.

Deterministic finite automata

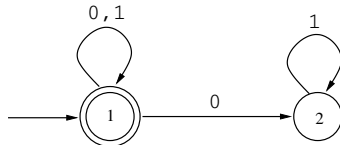
A finite automaton is *deterministic* if:

- it has a unique start state, and
- from every state there is exactly one transition for each possible input symbol.

Example



deterministic



non-deterministic

Can you draw a DFA that recognises the same language as the NFA?

Deterministic finite automata (formally)

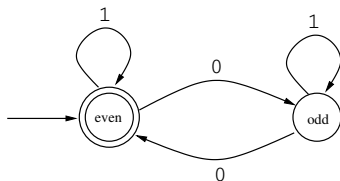
Definition: A *deterministic finite automaton* (or *DFA*) is a tuple

$$M = (Q, \Sigma, q_0, F, \delta)$$

consisting of:

- 1 a finite set Q of *states*,
- 2 a finite *alphabet*, Σ ,
- 3 a distinguished *starting state* $q_0 \in Q$,
- 4 a set $F \subseteq Q$ of *final states* (the ones that indicate acceptance),
- 5 a description δ of all the possible *transitions*, given by a table that answers the question: “Given a state q and an input symbol a , what is the next state?” There must be an answer to this no matter what q and a are.
 δ is called the *transition function* of M .

Example: Binary strings with an even number of zeros, revisited



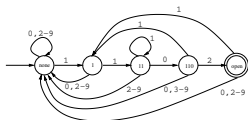
This is a DFA formally specified by:

$$\left(\{ \text{even}, \text{odd} \}, \{ 0, 1 \}, \text{even}, \{ \text{even} \}, \delta \right),$$

where the transition function δ is given by the following table:

δ	0	1
even	odd	even
odd	even	odd

Example: Combination lock, re-revisited



This is a DFA formally specified by:

$$\left(\{ \text{none}, 1, 11, 110, \text{open} \}, \{ 0, \dots, 9 \}, \text{none}, \{ \text{open} \}, \delta \right)$$

where the transition function δ is given by the following table:

δ	0	1	2	3	4	5	6	7	8	9
none	none	1	none	none			...			
1	none	11	none	none			...			
11	110	11	none	none			...			
110	none	1	open	none			...			
open	none	1	none	none			...			

Regular Expressions

- Recall from Inf1 that we developed a notation for regular languages, we called these regular expressions.
- If our alphabet is Σ , then the following are the basic expressions we build into more complex expressions:
 - φ stands for the empty language $\{\}$
 - ε stands for the language just consisting of the empty string $\{\varepsilon\}$
 - a where $a \in \Sigma$ stands for the language consisting of just one string $\{a\}$
- If r_1 and r_2 , are regular expressions standing for the languages L_{r_1} and L_{r_2} , then:
 - union: $r_1 + r_2$ is a regular expression standing for the language $L_{r_1} \cup L_{r_2}$
 - concatenation: $r_1 r_2$ is a regular expression standing for the language $\{s_1 s_2 \mid s_1 \in L_{r_1}, s_2 \in L_{r_2}\}$
 - asterate: r_1^* is a regular expression standing for $\{\varepsilon\} \cup \{s_1 \dots s_n \mid s_1 \in L_{r_1}, \dots, s_n \in L_{r_1}, \text{ for all } n \geq 1\}$

Other Operations

- Recall we were also able to define machines that recognised some more complex operations on regular sets.
- In particular, recall the construction of FSAs that found the intersection and interleaving of two regular sets.
- Recall also that these constructions resulted in significant increases in the size of the statespace because they form the product of the statespaces of two FSMs.
- Our constructions show that we can extend regular expression with the \cap and $|$ operations and hence have the means to give concise descriptions of regular sets.
- Recall also the system of equations of *regular algebra* that allowed us to reason about equality of regular sets.

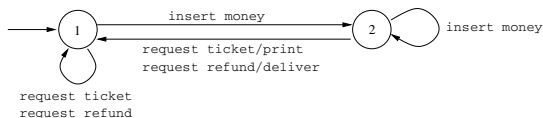
FSMs to Regular Grammars and Back Again

- Recall that in regular grammars, all productions are of the form $A \rightarrow aB$, $A \rightarrow \varepsilon$ or $A \rightarrow Ba$, $A \rightarrow \varepsilon$ (note that if you mix productions of the form $A \rightarrow aB$ and $A \rightarrow Ba$ then the grammar is no longer regular (why?).
- Given a FSM $M = (Q, \Sigma, \delta, s, F)$ we can construct a corresponding regular grammar

$$G = (\quad \{S_q \mid q \in Q\}, \Sigma, \\ \quad \{S_f \rightarrow \varepsilon \mid f \in F\} \cup \\ \quad \{S_q \rightarrow aS_{q'} \mid q' \in \delta(q, a)\}, S_s)$$

Example

- Consider the parking machine:



- Assuming state 1 is final and abbreviating the inputs as i_m, r_t, r_r , the corresponding regular grammar is:

$$\left(\begin{array}{l}
 \{S_1, S_2\}, \{i_m, r_t, r_r\}, \\
 \{S_1 \rightarrow \varepsilon, S_1 \rightarrow r_t S_1, S_1 \rightarrow r_r S_1, \\
 S_1 \rightarrow i_m S_2, S_2 \rightarrow i_m S_2, S_2 \rightarrow r_t S_1, S_2 \rightarrow r_r S_1\}, S_1)
 \end{array} \right)$$

Pigeon Holes

- In the next lecture we will consider the result that characterises the looping character of any regular language. Here we consider a preparatory result.
- The pigeon hole principle is a simple result on classification of data. It says that if we classify n things into m categories where $n > m$ then at least two things have the same classification.
- For example, if everyone in the UK only bought Ford, Nissan or Rolls Royce cars then as soon as we have seen 4 or more cars we know at least two have the same maker.
- We use this simple observation to demonstrate a useful fact about regular languages

Regular Languages that include “long” sentences are infinite: 1

- We'll show: for any regular language L there is a constant n_L such that if $s \in L$ and $|s| \geq n_L$ then the language L is infinite.
- If L is regular then there is a DFSA M_L (having no ε -transitions) that recognises L , let n_L be the number of states in M_L .
- If $s \in L$ and $s = a_1 \dots a_m$ and $m \geq n_L$ and q_0 is the initial state of M_L and the machine is in state q_j after having read $a_1 \dots a_j$, then we can see ...

Regular Languages that include “long” sentences are infinite: 2

- We have a sequence of at least $n_L + 1$ states: q_0, \dots, q_m so we classify the $m + 1$ indices by the states, i.e. j belongs to classification q_j
- Since there are only n_L states and the list of states is longer than n_L , at least two are classified by the same state (pigeon hole principle).
- If those indices are $k < l$ then $a_1 \dots a_k a_{l+1} \dots a_m \in L$ and $a_1 \dots a_l a_{k+1} \dots a_l a_{l+1} \dots a_m \in L$ and $a_1 \dots a_l a_{k+1} \dots a_l a_{k+1} \dots a_{l+1} \dots a_m \in L$ for as many repetitions as we like.
- As a consequence L is infinite.

What help is the result?

- It helps us show that some languages are not regular.
- For example we know the language $L_{par} = \{(^n)^n \mid n \geq 0\}$ is context-free, (the grammar $(\{S\}, \{(,)\}, \{S \rightarrow \varepsilon, S \rightarrow (S)\})$ does it) but it is not regular.
- We prove by contradiction so we assume L_{par} is a regular language.
- If it is regular then there is some $n_{L_{par}}$ that fits the conditions of our result,
- So if we consider the sentence $(^m)^m$ where $m > n_{L_{par}}$ then our result tells us there is some $p > 0$ such that $(^{m+ip})^m$ is in the language for every $i \geq 0$.

Summary

- We have reviewed some of the material of Inf1 and considered the link between FSMs and regular grammars.
- We have begun to characterise what languages can and cannot be defined by FSMs.
- In the next few lectures, we will:
 - Consider the relationship between FSAs and regular expressions.
 - Sharpen the results on the power of FSAs