COMP11212 - Fundamentals of Computation

Todd Davies

April 6, 2014

Introduction

The building of real-life computing systems, e.g. mobile phone, tv/video remote control, internet shopping, air-traffic control, internet banking, etc., is always a complex task. Mistakes can be very annoying, costly and sometimes life threatening. Methods and techniques to support the building and understanding of such systems are essential. This course unit provides an introduction to the basic computer science ideas underlying such methods. It is also a part of, and an introduction to, the Modelling and Rigorous Development theme.

Aims

This course unit provides a first approach to answering the following questions. What methods are there that can help understanding complicated systems or programs? How can we make sure that a program does what we intend it to do? How do computers go about recognizing pieces of text? If there are two ways of solving the same problem, how can we compare them? How do we measure that one of them gives the solution faster? How can we understand what computers can do in principle, and are there problems that are not solvable by a computer?

Additional reading

None.

Contents

1 Terminology

In order to talk about Strings in any meaningful way, we must first define terminology that we can use to describe exactly what we mean. What follows is a list of the terminology used throughout the course:

- A **symbol** is basically a letter. They are the basic component of all the data we use in the course. Examples include: *a, A, (, $, 7*.
- An **alphabet** is a collection of symbols that we can think of as a set. Example alphabets may include binary (*0, 1*), Latin letters (a,...,z,A,...,Z) etc.
- A **String** is a collection of symbols from an alphabet grouped together, sometimes called a word. Examples include *ababa* and *100101*.
- The *empty word* is a String consisting of no symbols. It is denoted by the letter ϵ.
- **Concentration** an operation that takes two Strings and combines them to create one longer String. For example concentrating *t* and *he* would create *the*. We can use the power notation to concentrate a String with itself any number of times. For example, ho^3 would give us *hohoho*.
- A **language** is a collection of Strings that can be thought of as a set. Examples of languages could be $\{\emptyset\}$, $\{\epsilon\}$, $\{hot, hotter, hottest\}$ or $\{a^n | n \in \mathbb{N}\}$.

We also have notation for describing generic instances of such entities:

Entity	Generic notation
Symbol	x, y, z
Alphabet	Σ
String	s, t, u
Language	$\mathcal{L}$

Languages as sets Since languages are thought of as sets, we can perform all the usual set operations on them (see my `COMP11120` notes for more information on these operations). Languages can be concentrated as described above.

Sometimes, we may want to define any finite number of concatenations of a language, and this is known as the **Kleene star**. The notation is $(\mathcal{L})^*$.

An interesting case of this is when a languages is subject to concatenation zero times ($\mathcal{L}^0$), since this would return the empty word ϵ.

> Note:
>
> *Kleene* is pronounced like *genie*.

2 Describing languages

2.1 Describing languages through patterns

A pattern describes a generic form that a set of Strings can take. If any String from the set is compared with the pattern then it will match the pattern. Any String from outside the set will not match the pattern.

For example, the pattern $(ab)^*$ would match Strings such as ϵ, ab, $abab$, $abab \ldots ab$.

> **Note:**
>
> ϵ is matched here since it satisfies the pattern $(ab)^0$. This comes about because, the Kleene star matches all concatenations of a String.

2.2 Regular expressions

The terms *pattern* and *regular expression* are pretty much synonymous. The operators allowed in a regular expression are:

Empty pattern	The character $\emptyset$ is a pattern.
Empty word	The character ϵ is a pattern.
Letters	Every letter in Σ is a pattern.
Concatenation	If x and y are patterns, then so is (xy).
Alternative	If x and y are patterns, then so is $(x\|y)$.
Kleene Star	If x is a pattern, then so is (x^*).

> **Note:**
>
> If we were to analyse a pattern in a recursive fashion, then in order to end our analysis, we would eventually have to find one of a selection of *base cases*.The highlighted rows here represent *step cases* (that is to say that at least another level of recursion is needed to finish our analysis), while the un-highlighted lines are base cases.

2.2.1 Discarding brackets

Just like in high-school mathematics, brackets can be discarded when unnecessary. For example, the pattern ((0|1)*0) is equivalent to (0|1)*0, and (2|(1|0)) is the same as (2|1|0).

2.2.2 Matching a regular expression

As was implied above, we can define a regular expression by recursively applying more operators to a 'base case' until we have the desired pattern.

The following patterns match the following words:

Empty word	The empty word only matches the pattern ϵ.
Base case	A pattern x will match a character x where x is a member of Σ
Concatenation	If p_1 is a pattern and p_2 is a pattern, then p_1p_2 will match any word matched by p_1 prepended to and word matched by p_2
Alternative	$(p_1\|p_2)$ will match a word from either p_1 or p_2.
Kleene star	The pattern (p^*) will match any number of words that are matched by p concatenated with each other. It also matches the empty word.

Languages described by regular expressions

A language $\mathcal{L}$ described by a regular expression p is one made up of every word s that is matched by p:

$$\mathcal{L}(p) - \{s \in \Sigma^* | s \ matches \ p\}$$

Obviously, we can describe the same language using different patterns, for example:

$$\mathcal{L}((1|1)(0|0)) = \mathcal{L}(10)$$

We can find out the exact language defined by a pattern in the following manner:

$$\begin{aligned}
\mathcal{L}((0|1)(1)^*) &= \mathcal{L}(0|1) \cdot \mathcal{L}(1^*) \\
&= \mathcal{L}(0) \cup \mathcal{L}(1) \cdot \mathcal{L}(1^*) \\
&= \mathcal{L}(0) \cup \mathcal{L}(1) \cdot \mathcal{L}(1)^* \\
&= \{0\} \cup \{1\} \cdot \mathcal{L}(1)^* \\
&= \{0\} \cup \{1\} \cdot \{1\}^*
\end{aligned}$$

Regular languages

We say that a language is regular if we can describe the language by defining a regular expression.

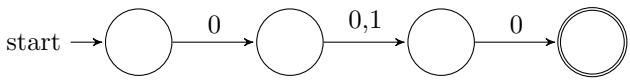
Finite State Automata

It is often useful to be able to describe languages using pictures. These are called finite state automata (FSA's). Every FSA must have the following things:

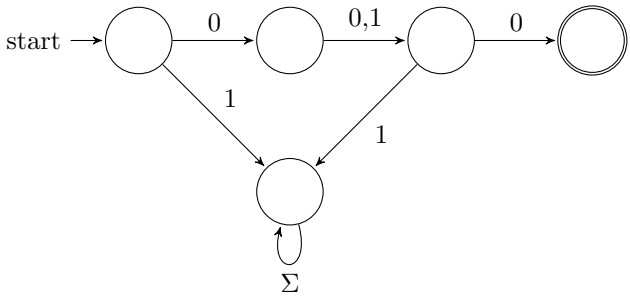
- One or more states.
- Arrows between states with labels directing the required conditions to traverse the arrow.
- A start state (indicated by an arrow with it's tail free).
- One or more accepting states. If we end up here after traversing the automation, then it will match our word.

Example automata

Here is an automata that will match the pattern 0(0|1)0:



If we wanted to, we could include all the *dump states* in the automata. Dump states are states that once entered into, it is impossible to reach an accepting state. The same automata with a dump state in would look like:



Formalising the automata

If we wanted to, we could define the set Q as the set of states in an automata, and the set F as the set of the accepting states, where $F \subset Q$. The transitions in the automata could be represented as a function δ, which takes an input state $s_{in} \in Q$, and a character $c \in \Sigma$, and returns another state $s_{out} \in Q$.

$$\delta : \{(s_{in}, c) | s_{in} \in Q, c \in \Sigma\} \rightarrow Q$$

Note:

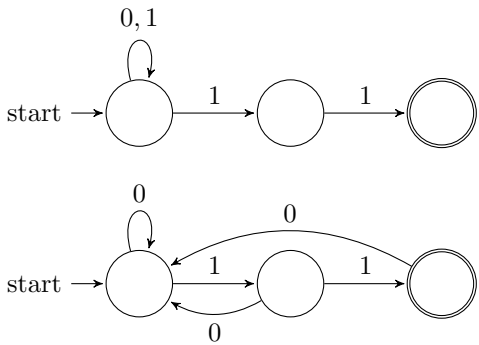
When an automata is defined like this, it can be seen as a *finite state machine*

Deterministic and non-deterministic automata

An automata is said to be deterministic if there is only one path through it for every word in $\mathcal{L}$, however, if there are multiple possible routes through an automata (i.e. there is

no unique path for one or more words), then we say that the automata is non-deterministic.

Here is an example of a deterministic and a non-deterministic automata that will accept the language of words defined by the pattern (0|1)*(11):

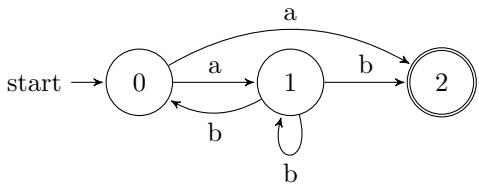


Algorithm one

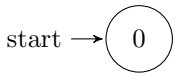
Algorithm one is a method for converting an NFA to a DFA. In order to do this, algorithm one walks through an NFA, in a recursive manner, finding all the possible collections of states that can be reached from each state. Unfortunately, this behaviour is hard to describe in a general and abstract manner, so it's probably easiest to just do an example.

Algorithm one example

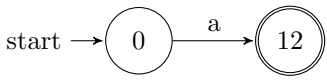
We'll do Exercise 19 in Andrea's notes, which is to convert the following NFA into a DFA:



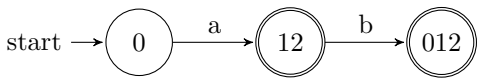
In order to apply algorithm one to this automata, we first need to write down the start state:



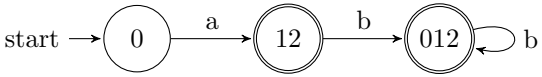
From the state 0, we can go to the state 1 or the state 2 with an a transition, therefore algorithm one dictates that we have to create a new state in our DFA called 12 that is linked to the state 0 with an a transition. Since 2 is an accepting state in the NFA, then our new state must also be an accepting state.



Now, we've dealt with the 0 state, we must concern us with where we can go from the state 12. Since this represents both the state 1 and the state 2 in the NFA, we must consider the transitions we can do from either of them. From state 1 in the NFA, we can go to either state 0, 1 or 2 with a b transition. Concequently, we should make a new state in our DFA for this:



From the state 0, 1 and 2, we can go anywhere with a b, so we should make a transition in our DFA for that:



From the states 0, 1 or 2 in the NFA, we can go to the states 1 or 2 with an a transition: