COMP11212 - Fundamentals of Computation

Todd Davies

August 8, 2015

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.

Licence and contrubution

Contents

1 Terminology **3**

2 Describing languages **3**
2.1 Describing languages through patterns . . . 3
2.2 Regular expressions . . . 3
2.2.1 Discarding brackets . . . 4
2.2.2 Matching a regular expression . . . 4
2.3 Languages described by regular expressions . . . 4
2.3.1 Regular languages . . . 4

3 Finite State Automata **5**
3.1 Example automata . . . 5
3.2 Formalising the automata . . . 5
3.3 Deterministic and non-deterministic automata . . . 5

4 Algorithm one **6**
4.1 Algorithm one example . . . 6

5 Algorithm two **7**
5.1 Numbering states . . . 7
5.2 Applying algorithm two . . . 7

6 Algorithm three **8**

7 Algorithm four **10**

8 Combining automata **11**
8.1 The product of two automata . . . 11
8.2 Finding the complement of an automata . . . 12
8.3 The Equivalence of Automata . . . 12
8.3.1 Minimization . . . 13
8.3.2 Complementation and Intersection . . . 13
8.3.3 Simlulations . . . 13

9 The limitations of regular languages **13**

10 Other types of languages **13**
10.1 Context Free Grammars . . . 14
10.2 Converting a DFA into a CFG . . . 15
10.3 Parse trees of CFG's . . . 15
10.3.1 Ambiguous parse trees . . . 16
10.4 Backus-Naur form . . . 16
10.5 Properties of context free languages . . . 16

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.
- **Concatenation** 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 concatenated 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.

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, $abab$, $abab \ldots ab$.

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^*).

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 a 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.

2.3 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:

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

2.3.1 Regular languages

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

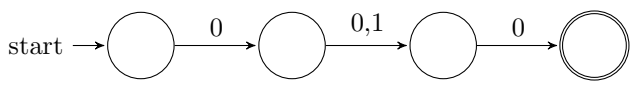
3 Finite State Automata

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:

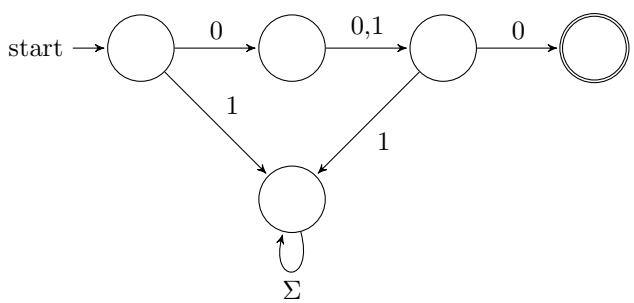
- One or more states.
- Arrows between states with labels directing the required conditions to traverse the arrow.
- A start state (indicated by an arrow with it's tail free).
- One or more accepting states. If we end up here after traversing the automation, then it will match our word.

3.1 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:



3.2 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 $s_{out} \in Q$.

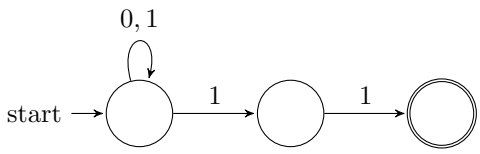
$$\delta : \{(s_{in}, c) | s_{in} \in Q, c \in \Sigma\} \to Q$$

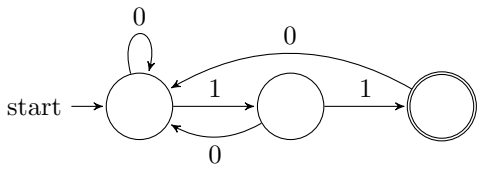
When an automata is defined like this, it can be seen as a *finite state machine*

3.3 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 $(0|1)^*(11)$:



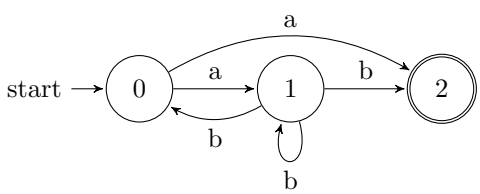


4 Algorithm one

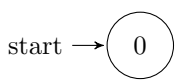
Algorithm one is a method for converting an NFA to a DFA. In order to do this, algorithm one walks through an NFA, in a recursive manner, finding all the possible collections of states that can be reached from each state. Unfortunately, this behaviour is hard to describe in a general and abstract manner,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

4.1 Algorithm one example

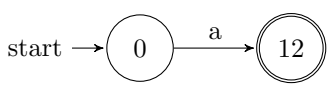
We'll do Exercise 19 in Andrea's notes, which is to convert the following NFA into a DFA:



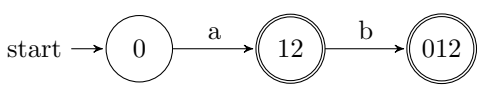
In order to apply algorithm one to this automata, we first need to write down the start state:



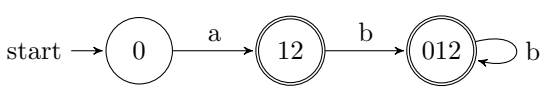
From the state 0, we can go to the state 1 or the state 2 with an a transition, therefore algorithm one dictates that we have to create a new state in our DFA called 12 that is linked to the state 0 with an a transition. Since 2 is an accepting state in the NFA, then our new state must also be an accepting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



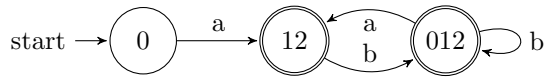
Now, we've dealt with the 0 state, we must concern us with where we can go from the state 12. Since this represents both the state 1 and the state 2 in the NFA, we must consider the transitions we can do from either of them. From state 1 in the NFA, we can go to either state 0, 1 or 2 with a b transition.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



If you want a formal definition of algorithm one, take a look at page 33 of Andrea's notes. although it's well explained, I don't think it's worth including here, since the formal definition won't be very useful in the exam.

5 Algorithm two

Algorithm two takes a DFA and converts it into a regular expression. If you want to convert an NFA into a regular expression, then you must first convert it into a DFA using algorithm one.

5.1 Numbering states

Algorithm two relies on us having named all the states in the DFA before we attempt to apply the algorithm. Although in theory, the algorithm will work with any state numbering, it is easiest to apply when the states are ordered in *reverse order of complexity*. That is to say that the states with the fewest connections will have the lowest number.

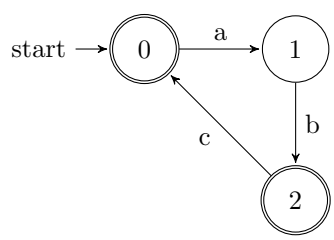
The numbering of the states isn't vital. If you get one or two the wrong way round, then it probably won't make the algorithm too complicated. Also, whether the algorithm is an acceptance state has no bearing on its number.

5.2 Applying algorithm two

First of all, we must determine which states a word from the language $\mathcal{L}$ accepted by the automation must start and end in. In order to do this, we must find the start states and all of the possible end states:

$$\mathcal{L} = \mathcal{L}_{start \to accepting_0} \cup \mathcal{L}_{start \to accepting_1} \cup \cdots \cup \mathcal{L}_{start \to accepting_n}$$

If the automation we were looking at was something like this:



Then the language described by the automation would be:

$$\mathcal{L} = \mathcal{L}_{0 \to 0} \cup \mathcal{L}_{0 \to 2}$$

We must now break down this expression into its component parts, $\mathcal{L}_{0 \to 0}$ and $\mathcal{L}_{0 \to 2}$ and evaluate them individually.

$$
\begin{aligned}
\mathcal{L}^{\leq 2}_{0\to 0} &= \mathcal{L}^{\leq 1}_{0\to 0} \cup \left(\mathcal{L}^{\leq 1}_{0\to 2} \cdot {\mathcal{L}^{\leq 1}_{2\to 2}}^{*} \cdot \mathcal{L}^{\leq 1}_{2\to 0}\right) \\
\mathcal{L}^{\leq 1}_{0\to 0} &= \epsilon \\
\mathcal{L}^{\leq 1}_{0\to 2} &= \mathcal{L}^{\leq 0}_{0\to 2} \cup \left(\mathcal{L}^{\leq 0}_{0\to 1} \cdot {\mathcal{L}^{\leq 0}_{1\to 1}}^{*} \cdot \mathcal{L}^{\leq 0}_{1\to 2}\right) \\
&= \emptyset \cup (a \cdot \epsilon^* \cdot b) \\
&= ab \\
\mathcal{L}^{\leq 1}_{2\to 2} &= \mathcal{L}^{\leq 0}_{2\to 2} \cup \left(\mathcal{L}^{\leq 0}_{2\to 1} \cdot {\mathcal{L}^{\leq 0}_{1\to 1}}^{*} \cdot \mathcal{L}^{\leq 0}_{1\to 2}\right)^{*} \\
&= \epsilon \cup (ca \cdot \epsilon^* \cdot b)^* \\
&= \epsilon | (cab)^* \\
\mathcal{L}^{\leq 1}_{2\to 0} &= c \\
\mathcal{L}^{\leq 2}_{0\to 0} &= \epsilon | ab(cab)^* c
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{L}^{\leq 2}_{0\to 2} &= \mathcal{L}^{\leq 1}_{0\to 2} \cdot {\mathcal{L}^{\leq 1}_{2\to 2}}^{*} \\
&= \{ab\} \cdot \{(cab)^*\} \\
&= ab(cab)^*
\end{aligned}
$$

We can now just concatenate the two languages we found and simplify the resulting regular expression:

$$
\begin{aligned}
\mathcal{L} &= \epsilon | ab(cab)^* c \cup ab(cab)^* \\
&= \epsilon | ab(cab)^* (c | \epsilon)
\end{aligned}
$$

6 Algorithm three

The role of algorithm three is to construct a non-deterministic finite automata from a regular expression.

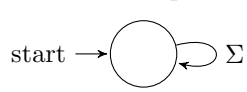
Algorithm three recursively analyses the regular expression into small chunks of NFA and then makes use of *epsilon transitions* to link these parts together.

An epsilon transition is a transition between states in the NFA that does not require a letter from the word to be traversed. This allows you to join NFA's together.

There are a number of general cases in algorithm three, where a regular expression of a particular form will create an NFA with a certain structure:

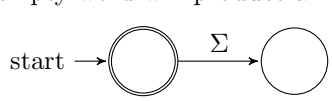
The empty pattern $\emptyset$:

A regular expression that accepts no words will produce an NFA like this:

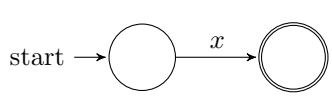


The ϵ pattern:

A regular expression that the empty word will produce an NFA like this:



A pattern accepting a word $x, x \in \Sigma$:

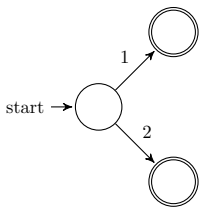


In order to produce a meaningful NFA, we must also handle cases where regular expressions implement concatenation, alternatives and the kleene star:

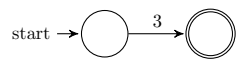
Concatenation:

In order to create an NFA that will recognise the same language as a pattern that uses concatenation (e.g. xy), we must first create two NFA's to accept each half of the concatenation, lets call these X and Y. In order to connect them, we must change all the accepting states in X to non-accepting states and make an ϵ transition from them to the start state of Y.

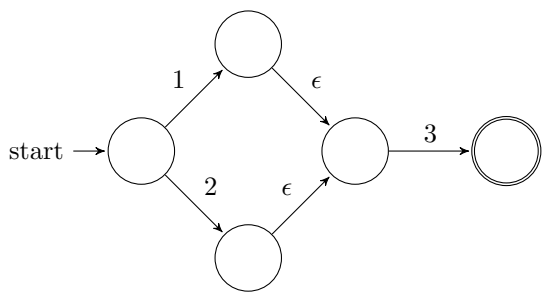
For example:



Concatenated with



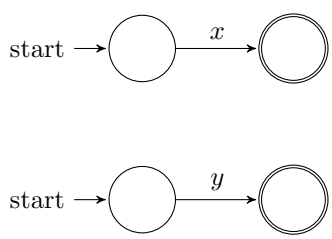
Goes to:



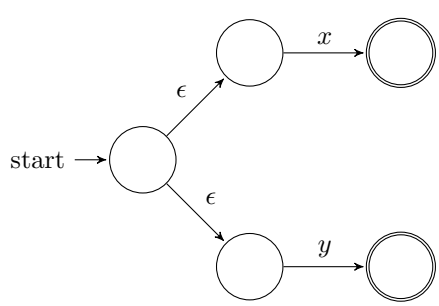
Alternative:

To create an NFA that implements an alternative such as $x|y$, then we must make a new NFA that has a start state that connects to the other NFA's that will accept x and y with ϵ transitions:

For example:



Combine to make:

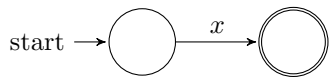


Kleene star:

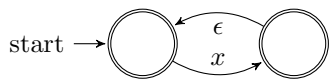
To implement a Kleene star with an NFA, you simply create an epsilon transition from each accepting state in the NFA to the start state, and make sure that the start state is

an accepting state.

For example:

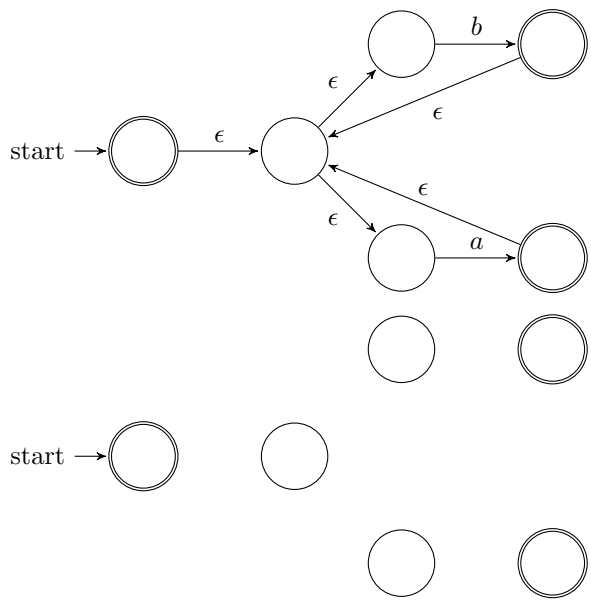


Goes to:

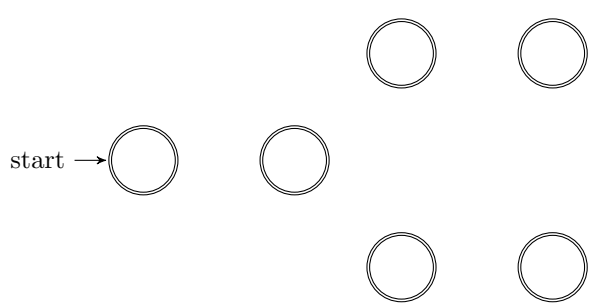


7 Algorithm four

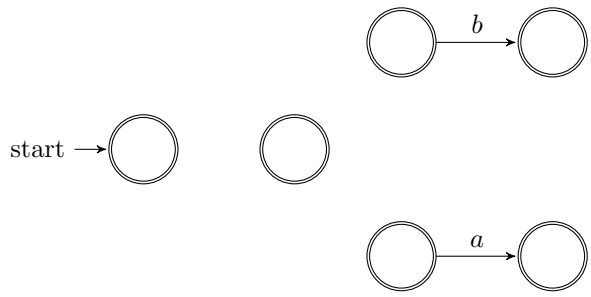
The purpose of algorithm four is to convert an NFA into another NFA without any epsilon transitions. In order to do this, you start with an NFA and then create a new automation with the same states as the NFA, but with no transitions:



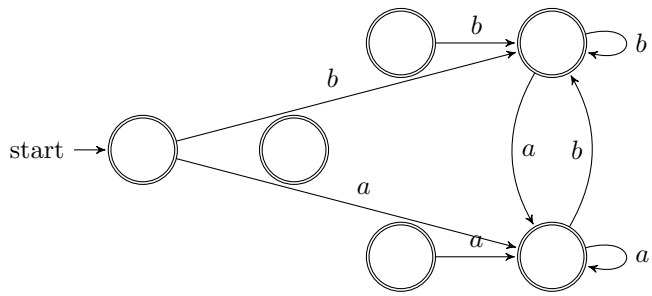
We then look at each non-accepting state, and if we can reach it using *only* epsilon transitions from an accepting state, then we convert that non-accepting state to an accepting state.



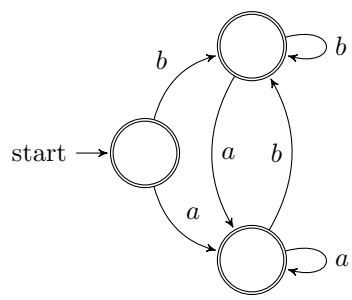
Then we copy all of the transitions with a letter (not ϵ) to the new automation.



Then we look at all the transitions where an actual letter is involved, and see if there are epsilon transitions leading to the start state for that transition, in which case we can link the state where the epsilon transition started to the end state of the letter transition.



Then we can take out the states that have no input transitions:



You can see that this automata accepts all the words matching the pattern $(a|b)^*$, which obviously could be represented by a different, more concise NFA. However, the goal of algorithm four isn't to produce efficient an efficient NFA, only to get rid of the epsilon transitions while still maintaining the same set of accepted words.

8 Combining automata

If you regard automata as a set of all the words that they accept, then it is logical that you would be able to perform set operations on them. For some operations, this is easy. For example, combining two automata to create a new automata that accepts the union of their individually accepted languages is easy; just create a new start state and have epsilon transitions going from it to each of the separate automata.

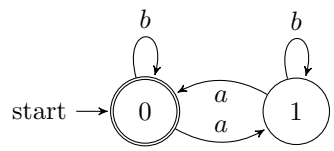
Concatenation automata is easy too, just have an epsilon transition going from the accepting states of the first automata to the start state of the second one.

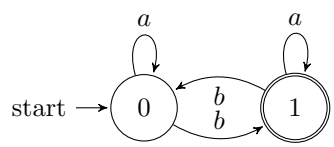
However, not all operations are so straight forward. What if we want to find the intersection of two automata?

8.1 The product of two automata

In order to produce an automata that accepts words that are only accepted in both of two other automata (i.e. their intersection), then we must apply an algorithm to find the product of the two automata.

Say if we wanted to combine these two automata, which accept words that have even numbers of a's and odd numbers of b's respectively:





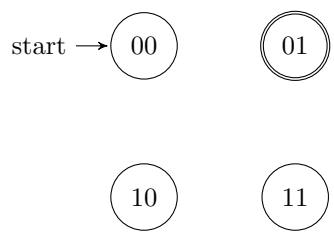
We first need to decide what states our new automata will need. This can be found by finding the powerset of the states of the two 'parent' automata; $\{0,1\} \times \{0,1\} = \{00, 01, 10, 11\}$.

Now we need to decide the start state for the automata, which is going to be the state that is the start state in both the parent automata, in our case 00.

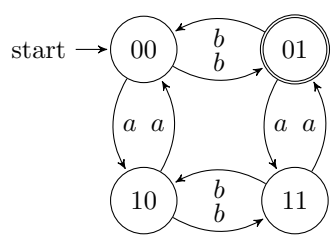
Since there is only one start state in every automata, there can only be one state in the powerset of the parent automata states that is made of two start states.

We then need to decide the accepting states, which are just the states in the powerset that are composed of two accepting states, in our case, this is just 01.

At the moment, our automata might look like this:



Now we need to decide where the state transitions need to go. In order to do this, we need to put a transition $(a, b) \to (c, d)$ if and only if, there is a transition from $a \to c$ in the first parent automation, and another one from $b \to d$ in the second parent automation. Henceforth:



This automata will now accept all the languages that are accepted by both its parent automata.

8.2 Finding the complement of an automata

In order to find an automata that will accept all the words not accepted by another automata, and reject all the ones that are, we must first find the DFA of the automata, and the we can simply swap the accepting states with the non-accepting states in order to find an automata that accepts the complement of the language previously accepted.

8.3 The Equivalence of Automata

It is obvious that one can create a number of distinct automata that will recognise the same language. However, if there are two different automata, how do we decide, with any element of certainty that they do in fact accept exactly the same language?

In fact, there are three different ways of doing so; *minimization*, *complementation and intersection* and *simlulations*.

8.3.1 Minimization

There is an algorithm that when given a Deterministic Finite Automaton, will minimize the number of states in the DFA, so that it is efficient as possible. If we apply the algorithm to two different DFA's, and get the same result, then we will know that they accept the same language.

However, this algorithm is non-trivial and is not required to be known for this course.[1]

8.3.2 Complementation and Intersection

In subsection 8.2, we saw how it is possible to find the complement of an automata. If we can do this, to one DFA, we can then apply what we learnt in subsection 8.1 to find the intersection of the two automata.

In Andrea's notes, this is not the preferred way of finding homogeneous automata, since finding the product of the two automata takes a long time. I disagreed, until I tried to apply the product algorithm on two of my own automata, and promptly changed my mind.

8.3.3 Simlulations

Both 'minimization' and 'complementation and intersection' only work on DFA's, which is annoying, since if you wanted to use them on an NFA first, you'd have to apply Algorithm One from section 4 first.

Simulations, on the other hand, work for both NFA's and DFA's.

Check Andrea's notes (page 56 at the time of writing) for how to construct a simulation.

9 The limitations of regular languages

One limitation of regular languages is that they cannot count beyond any pre-defined number. For example, I could create a DFA to find if a string is of length 5, but I couldn't create an automation to count how long any given string is. What if the string was infinite? Since DFA's are finite, I could never make enough states when I was drawing my DFA.

This may seem like only a minor inconvenience, but what if we want an automaton to see if there are the same number of 1's and 0's in an automation? We can define a language that will produce these words - $\mathcal{L} = \{0^n 1^n | n \in \mathbb{N}\}$, but we can't produce an automaton that will accept that language.

10 Other types of languages

Up to now, we have looked at definitions of languages that work by recognising different words; If the word is recognised then it is part of the language. However, what if we were to flip this on its head and have our definition of a language *generate* the words instead of merely recognising them?

We can do this by defining a recursive grammar that allows us to iteratively generate a string. Such grammars will always have a start symbol that we begin with, ususally S. Such an example may be:

[1] But you can read more about it here: `http://en.wikipedia.org/wiki/DFA_minimization`

$$
\begin{aligned}
S &\to A \\
A &\to aAa \\
A &\to B \\
A &\to \epsilon \\
B &\to bBb \\
B &\to C \\
B &\to \epsilon \\
C &\to cCc \\
C &\to \epsilon
\end{aligned}
$$

This language defines all the words of the form $a \ldots ab \ldots bc \ldots cb \ldots ba \ldots a$. In order to generate a word using the grammar, you simply start with S and then replace the capital letters with whatever corresponding values you can find in the grammar. This is called a **derivation** from the grammar.

The capital letters are referred to as *non-terminal* symbols since we can never have finished generating a word when there are still these letters in the word. On the other hand, the lower case letters are called *terminal* symbols, since when the word consists only of these, then it is in its final form and we cannot operate on it any more.

For example, to derive *abbccbba* you would do:

$$
\begin{aligned}
S &= A \\
&= A \\
&= aAa \\
&= aBa \\
&= abBba \\
&= abbBbba \\
&= abbCbba \\
&= abbcCcbba \\
&= abbc\epsilon cbba \\
&= abbccbba
\end{aligned}
$$

We often combine multiple definitions of the same non-terminal symbol into one, so the grammar defined above would be:

$$
\begin{aligned}
S &\to A \\
A &\to aAa|B|\epsilon \\
B &\to bBb|C|\epsilon \\
C &\to cCc|\epsilon
\end{aligned}
$$

10.1 Context Free Grammars

What we have just described is a *context free grammar* (CFG). Formally, it is defined as:

- an alphabet Σ of terminal symbols
- an alphabet Ξ of non-terminal symbols where $\Sigma \cap \Xi = \emptyset$
- A start symbol $S \in \Xi$
- A set of rules to convert terminal symbols into other terminal and non-terminal symbols.

In the example we have previously covered:

$$\Sigma = \{a, b, c\}$$
$$\Xi = \{S, A, B, C\}$$

Such grammars are referred to as context free since for any valid CFG, we can *always* replace any non-terminal symbol using some of the rules of the grammar, no matter what the state of the generated word is.

In a set theoretical notation, we could say that a string $X \in (\Sigma \cup \Xi)^*$ is a string generated by a grammar G if there is a sequence of strings:

$$S \to X_0 \to X_1 \to \cdots \to X$$

Such that each step denoted by $\to$ is an application of one of G's production rules.

10.2 Converting a DFA into a CFG

Converting a DFA into a CFG is easy, just follow these rules:

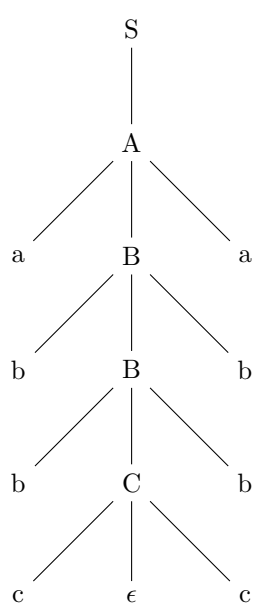
- Use Σ as the alphabet of terminal symbols.
- Make sure every state in the DFA is labelled, and use the set of states (Q) as Ξ.
- Use the start state in the DFA as the start symbol in the CFG.
- For every transition $A \xrightarrow{y} B$ in the DFA have a corresponding production rule $a \to yB$ in the CFG.
- For every accepting state E in the DFA, add a production rule $E \to \epsilon$ in the CFG.

Because we can convert every DFA into a CFG, we know that CFG's can be used to represent every regular language.

Every CFG generated from a DFA in this manner will be **right linear**. This means that it only has production rules with terminal symbols on the left and non-terminal symbols on the right.

10.3 Parse trees of CFG's

It is possible to create a parse tree of a word generated by a CFG in order to show how the word was generated. For example, the parse tree for *abbccbba* we generated in section 10 would look like:



In order to read off the word generated by a parse tree of a CFG, then just walk the leaves of the tree from left to right.

10.3.1 Ambiguous parse trees

Sometimes, it is possible to produce a word using a CFG in multiple ways. If that is the case, then multiple parse trees will be created that all have the same order of leaves.

We can modify our CFG's so that they cannot be used in an ambiguous way. There's no standard algorithm to do this, however, if you create more non-terminating symbols to differentiate between different ways of producing content, then it will make the parse trees differ and therefore become unambiguous.

See page 72 in Andreas notes for an example.

10.4 Backus-Naur form

Backus-Naur form is an alternative way of expressing CFG's. Instead of having capital letters for non-terminal symbols, they are represented by any string inside angular brackets (so $\langle number \rangle$ instead of N). In addition, the arrow ($\rightarrow$) is replaced by the symbol ::=.

10.5 Properties of context free languages

Not all of the properties such as (intersection and concatenation etc) that we can make use of with regular languages are also applicable to context free languages too. Here is a table listing which ones are and aren't:

Concatenation	This does work for context free languages. To concatenate two CFG's, just all the non-terminating symbols a subscript 1 or 2 depending on what language they are from, then add another production rule $S \rightarrow S_1 S_2$.
Kleene star	In order to implement a kleene star in a CFG, just add two new production rules; $S \rightarrow SS$ and $S \rightarrow \epsilon$.
Reversal	In order to reverse the words generated by a CFG, just reverse the right hand sides of all the production rules. So $A \rightarrow 0R1$ would become $A \rightarrow 1R0$.
Union	Similar to concatenation, make each non-terminating symbol in the two CFG's subscripted by a 1 or 2, and then add two new production rules; $S \rightarrow S_1$ or $S \rightarrow S_2$.
Intersection	Intersection is not possible with CFG's.