# Machines and their Languages

Michael B. Gale

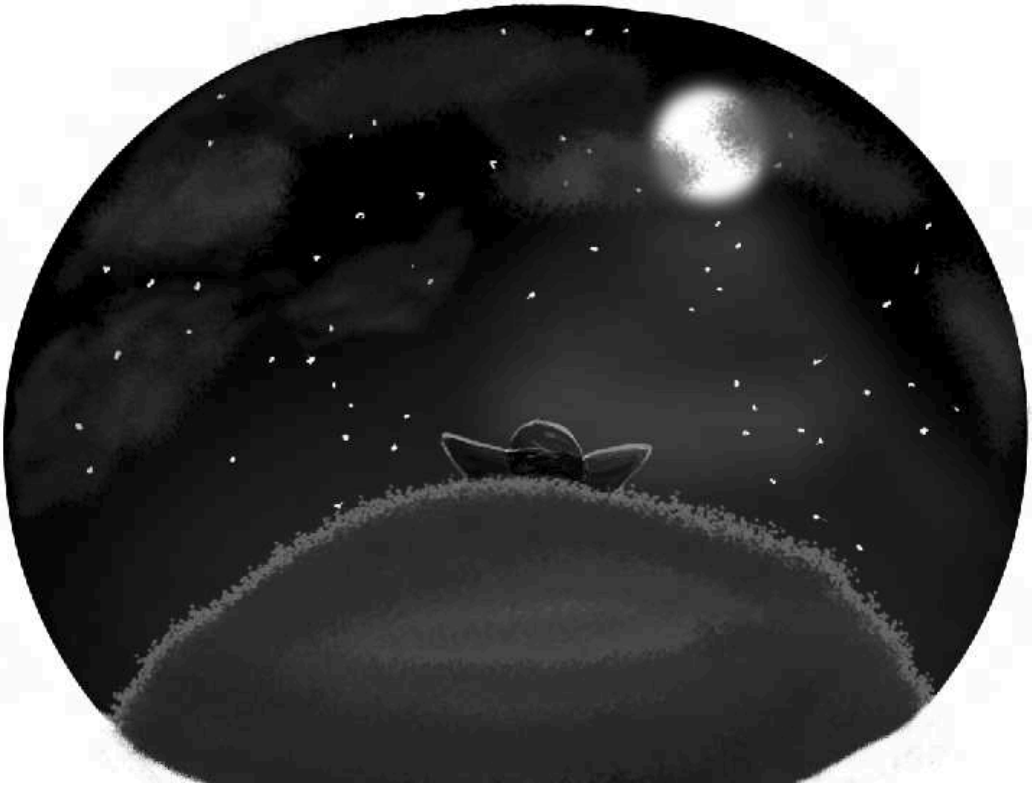


Illustration by Joe Nash

# Preface

This introduction to automata theory was originally written for the *Machines and their Languages* module at the University of Nottingham in the 2011/2012 academic year. It has since undergone various improvements and changes such as the addition of exercises, chapters which were missing in the first edition as well as extended examples for the various types of automata.

Disclaimer: this guide is still very much a draft and some content may be contain errors or may even be missing. You should only use this document in addition to other sources. If you find a mistake, by all means contact the authors.

*Michael B. Gale, January 2013*

## Recent changes

This section contains an overview of recent changes to this document.

**3 June 2013** Fixed some typographic errors in the chapter about subset construction and the pumping lemma.

**25 February 2013** We added more explanations to the examples for regular expressions and added formal definitions to the section which explains how regular expressions can be converted to NFAs.

**7 February 2013** We added a section about the *principle of induction* to the first chapter, a few notes on different notations and a proof of the equivalence between NFAs and DFAs.

# Contents

**1 Formal Languages** **5**
1.1 Sets . . . 5
1.1.1 Powerset . . . 7
1.1.2 Subset . . . 7
1.1.3 Union . . . 7
1.1.4 Intersection . . . 9
1.2 Sets of Functions . . . 9
1.3 Cartesian Product . . . 10
1.4 Quantifiers . . . 11
1.4.1 Universal Quantifier . . . 11
1.4.2 Existential Quantifier . . . 11
1.5 Inductive Definitions . . . 11
1.6 Principle of induction . . . 12
1.7 Alphabets . . . 14
1.7.1 Words . . . 15
1.8 Languages . . . 15
1.8.1 Concatenation . . . 16
1.8.2 Kleene Star . . . 17
1.8.3 Chomsky Hierarchy . . . 18
1.9 Exercises . . . 18

**2 Deterministic Finite Automata** **20**
2.1 Definition . . . 20
2.2 Example . . . 22
2.2.1 Language Membership . . . 23
2.3 The Natural Progression... . . . 26
2.4 Exercises . . . 27

**3 Nondeterministic Finite Automata** **28**
3.1 Definition . . . 28
3.2 Intuition . . . 29
3.3 Example . . . 30

3.4 Subset Construction . . . . . 33
3.4.1 Definition . . . . . 34
3.4.2 Lazy Subset Construction . . . . . 35
3.4.3 Proof . . . . . 37
3.5 Examination relevance . . . . . 38

**4 Regular Expressions** **39**
4.1 Definition . . . . . 39
4.1.1 Syntax of Regular Expressions . . . . . 39
4.1.2 Examples . . . . . 41
4.2 Converting Regular Expressions to NFAs . . . . . 44
4.2.1 $N(\emptyset)$ . . . . . 44
4.2.2 $N(\varepsilon)$ . . . . . 45
4.2.3 $N(x)$ . . . . . 45
4.2.4 $N(\alpha + \beta)$ . . . . . 45
4.2.5 $N(\alpha\beta)$ . . . . . 45
4.2.6 $N(\alpha^*)$ . . . . . 47

**5 Table-Filling Algorithm** **48**

**6 Pumping Lemma** **54**
6.1 Proof . . . . . 55
6.2 Example: Palindromes . . . . . 55
6.3 Example: $n \neq m$ . . . . . 57

**7 Pushdown Automata** **59**
7.1 Instantaneous Descriptions . . . . . 60
7.2 Language Membership . . . . . 62
7.3 Example . . . . . 62
7.4 The Language of a PDA . . . . . 65
7.4.1 Acceptance by Final State . . . . . 66
7.4.2 Acceptance by Empty Stack . . . . . 66

**8 Context-free grammars** **67**

**9 Further Resources** **68**
9.1 Turing Machines . . . . . 68
9.2 Recursive Descent Parsers . . . . . 69

# Chapter 1

# Formal Languages

In a nutshell, this book is concerned with *formal languages* and their different representations. Before we introduce what formal languages are and how they can be represented, let us first revise some concepts from other modules which we will need throughout this guide.

If you are already happy with sets, quantifiers and inductive definitions, then you can skip forward to section 1.7.

## 1.1 Sets

To start off, let us recall what we learned about *sets* in G51MCS (**?**). A set is a collection of distinct objects that is denoted using curly brackets. For example, a set with the integers 1, 2 and 3 in it can be written as

$$\{ 1, 2, 3 \}$$

We can also assign a name to this set. By convention, we name sets using upper-case characters which makes it easier to distinguish them from other variables.

$$A = \{ 1, 2, 3 \}$$

Sets don't have to contain integers. For example, we could define a set which contains different fruits.

$$B = \{ apple, banana, strawberry \}$$

Or we could define a set which contains colours.

$$C = \{ red, blue, green, yellow \}$$

If we want to say that an object is an element of a set, we use the $\in$ symbol which is read as *in* or *element of*. For example $2 \in A$, *straberry* $\in B$ and *yellow* $\in C$. On the other hand, if something isn't an element of a set we write *orange* $\notin C$.

We can also define sets using *set comprehensions* which are similar to list comprehensions in Haskell. In other words, instead of listing all elements of a set individually, we use a series of expressions to define how the members of a set may be generated or what their properties are.

$$D = \{\, n \mid n \in \mathbb{N}, n < 4 \,\}$$

In this example, we say that $D$ consists of elements $n$ such that $n \in \mathbb{N}$ and $n < 4$. Or in other words, we take each element of the set of natural numbers, refer to it as $n$ and if $n$ is smaller than 4, we put it in the set. I.e. we read the expressions after the | from the left to the right and just do the obvious thing.

We can evaluate that set comprehension if we wish to do so, such that

$$\begin{aligned} D &= \{\, n \mid n \in \mathbb{N}, n < 4 \,\} \\ &= \{\, 0, 1, 2, 3 \,\} \end{aligned}$$

However, we cannot always do this. For instance, if a set has so many elements that it would take us a long time to name all of them, it is simply more practical to come up with a set comprehension for it. In fact, some sets have infinitely many elements so that it becomes impossible to list all of them. The set of natural numbers $\mathbb{N}$ is one such set. List comprehensions allow us to define our own sets with infinitely many elements:

$$E = \{\, n \mid n \in \mathbb{N}, n \geq 4 \,\}$$

We could start to enumerate the elements in $E$, but we will obviously never be able to finish listing all of them

$$\begin{aligned} E &= \{\, n \mid n \in \mathbb{N}, n \geq 4 \,\} \\ &= \{\, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, \ldots \,\} \end{aligned}$$

When we talk about the number of elements in a set, we usually call this the *cardinality* of a set which is denoted using bars on the left and right of a set. Let us look at the cardinalities of some of the sets we defined previously

$$|A| = 3 \qquad |C| = 4 \qquad |E| = \infty$$

The set which has no elements is referred to as the *empty set* and is denoted using the $\emptyset$ symbol. Note that this simply means that $\emptyset = \{\}$ and both $\emptyset$ and $\{\}$ may be used interchangeably to denote the empty set. It should be obvious that $|\emptyset| = 0$.

### 1.1.1 Powerset

Throughout this module we will define functions which return a combination of elements from a set. Suppose that you have a set

$$A = \{ 1, 2, 3 \}$$

then all possible combinations of this set are expressed using $\mathcal{P}(A)$ where $\mathcal{P}$ is referred to as the *powerset* so that we can read $\mathcal{P}(A)$ as "the powerset of A".

$$\mathcal{P}(A) = \{ \emptyset, \{ 1 \}, \{ 2 \}, \{ 3 \}, \{ 1, 2 \}, \{ 1, 3 \}, \{ 2, 3 \}, \{ 1, 2, 3 \} \}$$

As we can see, $\mathcal{P}(A)$ is a set of sets where each set represent one distinct combination of elements from $A$. Recall that sets are not ordered and therefore $\{ 1, 2 \}$, for example, is the same as $\{ 2, 1 \}$, $\{ 2, 3 \} = \{ 3, 2 \}$ and so on which is why we only include one way of writing each set.

It is very easy to determine the cardinality of the powerset of any set using the following equality

$$|\mathcal{P}(X)| = 2^{|X|}$$

So for our previous example, the cardinality of $A$ was 3 or formally $|A| = 3$. If we then calculate $2^3$, we get 8 which is the cardinality of $\mathcal{P}(A)$.

### 1.1.2 Subset

We have learnt that we can express that an object is an element of a set using the $\in$ symbol. Similarly, we can express that a all elements from a set are elements of another set using the $\subseteq$ symbol which is read as "is a subset of".

For example, suppose that you have the set of even numbers

$$A = \{ n \mid n \in \mathbb{N}, n \bmod 2 = 0 \}$$

then $A \subseteq \mathbb{N}$ or in other words "A is a subset of the natural numbers".

### 1.1.3 Union

Now that we are familiar with the basics of sets, we can introduce some operators on sets. Let us define two sets to work with

$$A = \{ apple, banana \}$$
$$B = \{ banana, strawberry \}$$

The first operation on sets we need to know is called *union* and is denoted using the $\cup$ symbol. For instance, the union of $A$ and $B$ is written as

$$A \cup B$$

If we evaluate this operation, then we take all the elements from $A$ and all the elements from $B$ and put them into one new set. However, all elements need to be unique so that *banana* will only occur once in the resulting set

$$\begin{aligned} A \cup B &= \{\, apple, banana \,\} \cup \{\, banana, strawberry \,\} \\ &= \{\, apple, banana, strawberry \,\} \end{aligned}$$

Note that whenever we transform expressions from now own, we will use *hints* to explain each step. You should be familiar with this notation from modules such as G51APS, G51FUN and others. Let us add hints to the previous example and note that the curly brackets which are on the same lines as the equal signs do not denote sets, but instead surround the hints:

$$\begin{aligned} & A \cup B \\ = & \quad \{ \quad \text{definitions of } A \text{ and } B \quad \} \\ & \{\, apple, banana \,\} \cup \{\, banana, strawberry \,\} \\ = & \quad \{ \quad \text{set union} \quad \} \\ & \{\, apple, banana, strawberry \,\} \end{aligned}$$

This makes it easier to understand what is going on when there is more than one step involved in an example. You should try to use the hint notation in the same way whenever you transform expressions.

Now to move on, suppose that you have a set of sets

$$S = \{\, \{\, red, green, blue \,\}, \{\, yellow, blue \,\}, \{\, orange, purple \,\} \,\}$$

Later on, we will want to put all the elements of the subsets of $S$ into a single set. In other words we want

$$\{\, red, green, blue \,\} \cup \{\, yellow, blue \,\} \cup \{\, orange, purple \,\}$$

However, we will not always know what the elements of a set of sets are. For instance, this might happen if we define a function which takes a set of sets as argument or if we generate the powerset of a set whose elements we don't know. For this reason, we define a new operator on sets of sets which

performs the union on all sets in the set it is applied to

$$\begin{aligned}
&\bigcup S \\
=\quad &\{\quad \text{definition of } S \quad\} \\
&\bigcup \{\, \{\, red, green, blue \,\}, \{\, yellow, blue \,\}, \{\, orange, purple \,\} \,\} \\
=\quad &\{\quad \text{union on multiple sets} \quad\} \\
&\{\, red, green, blue, yellow, orange, purple \,\}
\end{aligned}$$

In addition to the basic understanding of how the set union works, you will also need to know some reasonably obvious properties about this operator which are summarised below.

| | |
|---|---|
| **Commutativity:** | $A \cup B = B \cup A$ |
| **Associativity:** | $A \cup (B \cup C) = (A \cup B) \cup C$ |
| **Identity:** | $A \cup \emptyset = A$ |
| | $\emptyset \cup A = A$ |

### 1.1.4 Intersection

Another binary operation on sets is called *intersection* and is denoted using the $\cap$ symbol. If we evaluate the intersection of two sets, we receive a new set which contains only the elements that occur in both sets

$$\begin{aligned}
A \cap B &= \{apple, banana\} \cap \{banana, strawberry\} \\
&= \{banana\}
\end{aligned}$$

## 1.2 Sets of Functions

Functions are mappings between the elements of two sets. Suppose that you have two sets, $A$ and $B$, and that you are defining a function $f$ which takes arguments that are elements of $A$ and returns elements of $B$ then we can then write this formally as

$$f \in A \to B$$

where $A \to B$ is the set of all functions which map elements from $A$ to elements from set $B$. As an example, consider the set of boolean values

$$Bool = \{\, True, False \,\}$$

then we can define a function $not \in Bool \to Bool$

$$not(True) = False$$
$$not(False) = True$$

Of course *not* isn't the only function in $Bool \to Bool$. As an example, we could define a function *id* which is also an element of $Bool \to Bool$

$$id(True) = True$$
$$id(False) = False$$

## 1.3 Cartesian Product

Often, we will define functions which take more than one argument, return more than one thing or both. In such cases we will need *cartesian products*. This is actually quite a straight-forward concept which we denote using the $\times$ symbol. For example, we could define a function

$$and \in Bool \times Bool \to Bool$$

with the following definition

$$and(True, True) = True$$
$$and(True, False) = False$$
$$and(False, True) = False$$
$$and(False, False) = False$$

As an example of a function which returns multiple things, let us define a function which maps an integer argument to it's predecessor and successor

$$ps \in \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$$

Recall that $\mathbb{Z}$ is the set of all integers. We define this function as

$$ps(n) = (n - 1, n + 1)$$

Of course, the type product of two sets is in itself a set. We could define the type product of two sets using a set comprehension

$$A \times B = \{ (x, y) \mid x \in A, y \in B \}$$

We can also produce the type product of more than two sets. For example, where elements in the type product of two sets are two-tuples, elements in the type product of three sets are three-tuples, elements in the type product of four sets are four-tuples and so on.

## 1.4 Quantifiers

Throughout this revision guide, we will often look at properties and in order to generalise these we need *quantifiers*. There are *two* types of quantifiers that you need to be familiar with.

### 1.4.1 Universal Quantifier

Let us consider a well-known property of addition: whenever we add 0 to some variable $n$, then the result will be just $n$.

$$n + 0 = n$$

However, this equation does not mention the fact that $n$ can be any integer. For all we know, $n$ could be a set. We can get around this problem by expressing the fact that $n$ can be any integer formally using the *universal quantifier* which is denoted using $\forall$ and read as "for all"

$$\forall n \in \mathbb{N}, n + 0 = n$$

So now we are saying that "for all $n$ where $n$ is an element of the set of natural numbers, $x + 0 = x$" and now there is no longer any ambiguity over what $n$ can be.

### 1.4.2 Existential Quantifier

The existential quantifier, written as $\exists$ and read as "there exists", is used to indicate that there exists *one or more* element in a set such that some expression is true.

For example, we could say that

$$\exists n \in \mathbb{N}, x * n = x$$

or in other words, we are saying that "there exists at least one element $n$ in $\mathbb{N}$ such that $x * n = x$".

## 1.5 Inductive Definitions

If you have taken G52IFR, you will be familiar with *inductive definitions* which are used to define new objects in terms of themselves. For example, the natural numbers are defined inductively

1. $0 \in \mathbb{N}$

2. $\forall n \in \mathbb{N}, n + 1 \in \mathbb{N}$

So rather than enumerating all natural numbers (which as you know, is something we can't do because there are infinitely many natural numbers), we say that 0 is a natural number and that the successor of every natural number is also a natural number.

If you are familiar with the Coq proof assistant, you could express the inductive definition above as

```
Inductive Nat : Set :=
  | Zero : Nat
  | Succ : Nat → Nat.
```

Or in Haskell, this could be expressed in a similar style using GADTs

```
data Nat where
  Zero :: Nat
  Succ :: Nat → Nat
```

Note, however, that you will not be asked to write any Coq or Haskell in the exam and that the examples above are purely for practical illustration and to aid your understanding if you feel more comfortable with Coq or Haskell notation.

## 1.6 Principle of induction

In the previous sections we have encountered finite sets such as *Bool* and sets with infinitely many elements such as $\mathbb{N}$. Suppose that we want to prove a property of a function whose domain is a finite set, then this is easy because we can either perform *case analysis* or simply rewrite the expression. For example, if we wanted to prove that

$$\forall b \in Bool, not(not(b)) = b$$

then we can simply perform case analysis on $b$. In other words, we will try to prove the property for $b$ substituted with every element in *Bool*. Let us

start with $b = True$:

$$\begin{array}{ll} & not(not(True)) \\ = & \quad \{ \quad \text{applying } not \quad \} \\ & not(False) \\ = & \quad \{ \quad \text{applying } not \quad \} \\ & True \end{array}$$

We have proven that $not(not(True) = True$. Now let us do the same for $b = False$:

$$\begin{array}{ll} & not(not(False)) \\ = & \quad \{ \quad \text{applying } not \quad \} \\ & not(True) \\ = & \quad \{ \quad \text{applying } not \quad \} \\ & False \end{array}$$

It is now obvious that the property holds for all elements of $Bool$. However, in many cases, we will be working with non-finite sets. How can we prove properties about functions whose domain is a set with infinitely many elements? We cannot use case analysis, because obviously we would never finish as there would be infinitely many cases to consider.

The answer to this problem is *induction* and it is closely related to the inductive definitions we have encountered in the previous section. We can note that sets with infinitely many elements are defined inductively using base cases and recursive (or inductive) cases. The principle of induction states that if we can prove a property for all base cases and we can then prove the property for all recursive cases, under the assumption that the property holds, then the property holds. This may sound a little confusing, but it is actually quite easy.

Let us consider an example using the definition of the natural numbers from the previous section (note that we will write $\mathbb{N}$ in place of $Nat$). Proving properties just using the definition we have is a bit boring, so let us first define an addition function:

$$\begin{array}{lcl} add & : & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ add(Zero, n) & = & n \\ add(Succ\ m, n) & = & Succ\ add(m, n) \end{array}$$

Now we can prove the following property using induction:

$$\forall x \in \mathbb{N}, x = add(x, Zero)$$

As stated above, we will first attempt to prove this property for all the base cases of $x$. Of course, the definition of $Nat$ only contains one such base case: namely, we begin with $x = Zero$:

$$\begin{aligned} & add(Zero, Zero) \\ = \quad & \{ \quad \text{applying } add \quad \} \\ & Zero \end{aligned}$$

We have shown, using the definition of $add$, that $Zero = add(Zero, Zero)$. Now that we have exhausted all base cases, we move on to the inductive cases. Remember that, for these cases, we will assume $\forall x \in \mathbb{N}, x = add(x, Zero)$ to be true. We will call this the *induction hypothesis*.

The only recursive case in the definition of $\mathbb{N}$ is $\forall n \in \mathbb{N}, Succ\ n \in \mathbb{N}$. Let us prove our property for $x = Succ\ n$ then:

$$\begin{aligned} & add(Succ\ n, Zero) \\ = \quad & \{ \quad \text{applying } add \quad \} \\ & Succ\ add(n, Zero) \\ = \quad & \{ \quad \text{induction hypothesis} \quad \} \\ & Succ\ n \end{aligned}$$

In case you find this confusing, remember that we assumed $\forall x \in \mathbb{N}, x = add(x, Zero)$ to be true. After we applied $add$ and we were left with $Succ\, add(n, Zero)$, we could observe that part of this expression is $add(n, Zero)$. Of course we have shown that this is equal to $n$ in our induction hypothesis, so we can rewrite the term to exactly that, which leaves us with $Succ\ n$.

## 1.7 Alphabets

As mentioned in the introduction to this chapter, this module is all about formal languages. Before we can define such languages, however, we need to be able to define *alphabets* over which we can then define the languages.

An alphabet is simply a *finite* set whose elements are referred to as *symbols* or sometimes as characters. This is very similar to the way natural languages work. For instance, words in the English language use characters from the Latin alphabet.

By convention, we use the Greek symbol $\Sigma$ (*Sigma*) to denote an alphabet

$$\Sigma = \{ 0, 1 \}$$

As you can see, $\Sigma$ here is simply the set consisting of 0 and 1. Note, however, that the 0 and 1 here are not numbers, but symbols in the alphabet. We can choose anything to be a symbol. For example, we could define an alphabet

$$\Sigma = \{ a, b, c \}$$

or an alphabet

$$\Sigma = \{ \clubsuit, \diamondsuit, \spadesuit, \heartsuit \}$$

### 1.7.1 Words

*Words* are *sequences* of symbols. Suppose that you have an alphabet $\Sigma$, then the set of all words using symbols from $\Sigma$ is denoted as $\Sigma^*$. We can define $\Sigma^*$ inductively (see 1.5) as

1. $\varepsilon \in \Sigma^*$
2. $\forall x \in \Sigma, \forall w \in \Sigma^*, xw \in \Sigma^*$

Note that $\varepsilon$ is referred to as the *empty word* and that we only write it when it occurs on its own as it is part of every element in $\Sigma^*$. We have also introduced a new operator called *concatenation* which allows us to concatinate two words. Unintuitively, this operator is represented by writing two words next to each other. In the second definition above, $x$ is a symbol and $w$ is a word so that $xw$ is the concatenation of $x$ and $w$.

For example, if $\Sigma = \{ 0, 1 \}$, then

$$\Sigma^* = \{ \varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots \}$$

Also note that while we may have an infinite number of words, all words have a finite length.

## 1.8 Languages

Now that we know how to define alphabets and words, we can define *formal languages*. A language is a set of words or in other words, a language is a subset of $\Sigma^*$. For example, let us define a language over the alphabet $\Sigma = \{ 0, 1 \}$

$$L_a = \{ 0110, 101, 1001 \}$$

We can define formal languages in the same ways we can use to define sets. For example, we could use a set comprehension to define a language over the alphabet $\Sigma = \{ 0, 1 \}$

$$L_b = \{ 0^n 1^n \mid n \in \mathbb{N} \}$$

Note that when we define formal languages, terms like $0^n$ mean $n$-many repetitions of the symbol 0 rather than exponentiation, because as we stated before 0 is not a number but a symbol from the alphabet $\Sigma$.

Some examples of words in $L_b$ include 01, 0011, 000111 and so on. It should be obvious that the cardinality of $L_b$ is $\infty$.

### 1.8.1 Concatenation

In section 1.7 we have learnt about the concatenation of words. There is also a concatenation operator for languages (which are, of course, just sets). Once again, we denote this type of concatenation by writing two sets or languages next to each other. For each element of the first set we take each element of the second set and concatinate the two.

$$\begin{aligned} & \{ 01, 1 \} \{ 0, 1 \} \\ = \quad & \{ \quad \text{concatenation of languages} \quad \} \\ & \{ 010, 011, 10, 11 \} \end{aligned}$$

Before we continue, let us consider another example using $\Sigma = \{ a, b, c \}$ where

$$\begin{aligned} L_a &= \{ ab, c \} \\ L_b &= \{ bb, b, cc \} \end{aligned}$$

Now we concatinate the two languages

$$\begin{aligned} & L_a L_b \\ = \quad & \{ \quad \text{concatenation of languages} \quad \} \\ & \{ abbb, abb, abcc, cbb, cb, ccc \} \end{aligned}$$

There is an important property of concatenation you need to be aware of: if we concatinate the empty set and something else, then we will always receive the empty set as result

$$\begin{aligned} \emptyset L &= \emptyset \\ L \emptyset &= \emptyset \end{aligned}$$

It should be obvious why this is the case if you look at the description of concatenation above. For the first case, we would try to go through all elements in the first set, but since it's the empty set, there are none and therefore we end up with nothing in the resulting set. Similarily, for the second case, we can go through all the elements in the first set, but then we won't find any in the set on the right hand side, so that we end up with the empty set once again.

### 1.8.2 Kleene Star

We have already encountered terms like $0^n$ in section 1.8 where 0 is an element of $\Sigma$ that mean $n$-many repetitions of 0. The same operator exists for languages. However, in the case of languages we **do not** mean repetitions, but instead we mean *number of concatenations to itself*. Let us define a language over the alphabet $\Sigma = \{ 0, 1 \}$ which we can use to demonstrate this

$$L = \{ 01, 10 \}$$

Now if we write a term such as $L^2$, we mean

$$\begin{aligned} & L^2 \\ = & \quad \{ \quad \text{meaning of } L^2 \quad \} \\ & LL \\ = & \quad \{ \quad \text{definition of } L \quad \} \\ & \{ 01, 10 \}\{ 01, 10 \} \\ = & \quad \{ \quad \text{concatenation} \quad \} \\ & \{ 0101, 0110, 1001, 1010 \} \end{aligned}$$

We can define this operator recursively for any language $L$ as

$$\begin{aligned} L^0 &= \{ \varepsilon \} \\ L^1 &= L \\ L^{n+1} &= LL^n \end{aligned}$$

Note that $L^0$ is never used for any $n \geq 1$. Where would we actually ever use $L^0$ then? The answer lies in another operator you need to know called *kleene star*. This operator represents the union of infinitely many concatenations of a language. For a language $L$ it is denoted using $L^*$ which we define as

$$L^* = \bigcup_{n=0}^{\infty} L^n$$

This expression means nothing other than

$$
\begin{aligned}
& L^* \\
= \quad & \{ \quad \text{kleene star} \quad \} \\
& \bigcup_{n=0}^{\infty} L^n \\
= \quad & \{ \quad \text{union on } L^n \text{ for } n = 0 \text{ to } n = \infty \quad \} \\
& L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots
\end{aligned}
$$

We can observe that $\varepsilon \in L^*$ for any language $L$.

### 1.8.3 Chomsky Hierarchy

You should have a reasonable understanding of what formal languages are and how we can define them using the help of set operators. Now we want to ask ourselves, what capabilities does a *machine* need to understand these languages?

In the remaining chapters of this guide, we will introduce different kinds of machines or *automata* which are capable of recognising the words in a formal language. However, we will also observe that not all automata can recognise all languages and that certain languages have certain characteristics.

Specifically, we will introduce different classes of languages from the *Chomsky hierarchy* and show which of them can be recognised by which type of automaton (**?**).

## 1.9 Exercises

1. Given the following languages, compute the resulting languages for parts (a) to (e):

$$
\begin{aligned}
L_1 &= \{B, C, F, L, M, S, T\} \\
L_2 &= \{\varepsilon\}
\end{aligned}
$$

   (a) $L_1\{ake\}$

   (b) $L_2\{ake\}$

   (c) $\emptyset\{ake\}$

   (d) $\emptyset \cup L_1$

(e) $L_1^* \cap L_2$

2. State which of the following properties hold for all languages which are defined over $\Sigma = \{0, 1\}$. If a property doesn't hold, explain why.

   (a) $\{\varepsilon\}L_1 = L_1\{\varepsilon\}$

   (b) $(L_1L_2)L_3 = L_1(L_2L_3)$

   (c) $L_1 \cup \emptyset = \emptyset L_1$

   (d) $L_1^*L_1^* = (L_1L_1)^*$

   (e) $L_1^* \subseteq L_1^*L_1^* \wedge L_1^*L_1^* \subseteq L_1^*$

# Chapter 2

# Deterministic Finite Automata

Albeit living in the 21st century, we unfortunately still live in a world where the resources that computers rely upon are limited. As such, it is reasonable to have a formal model which has access to a limited amount of information. This notion is captured precisely by *finite automata*, in both deterministic and nondeterministic flavours (which, as we shall see, are equivalent).

## 2.1 Definition

A *deterministic finite automaton* (DFA) is a mathematical representation of an operational semantics – or, a virtual machine – it is advised to at least be passingly familiar with what the former means, be it through Wikipedia or otherwise. There are five components needed to fully define a DFA:

**Deterministic Finite Automaton:**
$(Q, \Sigma, \delta, q_0, F)$

1. $Q$ is a *finite set*, the elements of which represent 'states' of the automaton. These states correspond to 'scenarios', or 'situations' (note I'm trying to be consistent with my usage of words starting with S!) that we find ourselves in at any given point.
2. $\Sigma$ is the alphabet used by the automaton (see section 1.7).
3. $\delta$ (*delta*) is an element of $Q \times \Sigma \to Q$ and it is known as the *transition function* (note that we use the symbol $\delta$ over anything else because in physics it is the letter used to denote 'change').

A single application of the $\delta$ function corresponds to a single transition within the automaton. This corresponds, for example, to the execution of a single machine code instruction, if you think of a state as a permutation of registers and the alphabet as the set of instructions.

This function $\delta$, whilst important, needs to be generalised so that the automaton can process *words* (see section 1.7.1) instead of *symbols*. The generalisation is called the *extended transition function*, which in this guide we denote using $\hat{\delta}$. It is an element of $Q \times \Sigma^* \to Q$ and its definition is as follows:

$$\begin{aligned} \hat{\delta}(s, \varepsilon) &= s \\ \hat{\delta}(s, xw) &= \hat{\delta}(\delta(s, x), w) \end{aligned}$$

$\delta^*$ is, in effect, the virtual machine which takes an initial state (see below!) and a word $w$ constructed using symbols from $\Sigma$ and computes the final state which results from applying the $\delta$ function to each element of $w$ in turn. Returning to the machine code example, the initial state is the set of empty registers and $w$ is a sequence of low-level instructions: or, a program!

Also note that $\delta$ is different for each DFA, but $\hat{\delta}$ is the same for every DFA which is why $\delta$ is included in the tuple that defines a DFA, but $\hat{\delta}$ isn't.

4. $q_0$ is an element of $Q$ which represents the *initial state*. This is the state which is passed to $\hat{\delta}$ as an initial argument.

5. $F$ is a subset of $Q$ and it is the set of *final states*. After running a word through $\hat{\delta}$, we test whether the resulting state is an element of $F$. If it is in $F$, then the word belongs to the language $L(A)$ which the automaton $A$ represents, recalling that $L(A) \subseteq \Sigma^*$.

$$\forall w \in L(A), \exists f \in F, \hat{\delta}(q_0, w) = f$$

In the language of mortals, this reads "*for all words in the language accepted by the automaton, there's a final state which is reached by running those words on the extended transition function from the initial state*".

One especially important connection to make is this

$$q_0 \in F_A \Rightarrow \varepsilon \in \mathrm{L}(A)$$

or in other words, if the initial state is a member of the set of final states, the empty word is accepted by the automaton.

A DFA can be represented by listing all of the above formally, but also via a *state transition diagram*. You're familiar with those, right? If not, there's one in the next section.

## 2.2 Example

In an exam setting, you may be given the definition for a DFA and you will be asked to construct a *transition table* for it. Let us consider the following example from the first coursework:

$$\Sigma = \{ b, c \}$$
$$A = (\{ q_0, q_1, q_2 \}, \Sigma, \delta, q_0, \{ q_2 \})$$

where $\delta$ is defined as

$$\delta(q_0, b) = q_1$$
$$\delta(q_0, c) = q_0$$
$$\delta(q_1, b) = q_2$$
$$\delta(q_1, c) = q_0$$
$$\delta(q_2, b) = q_2$$
$$\delta(q_2, c) = q_2$$

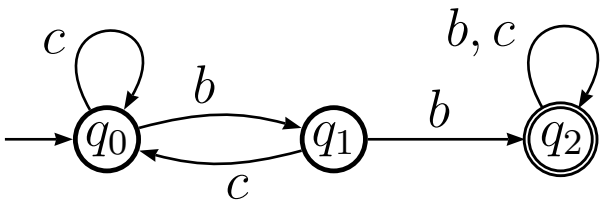
Constructing the transition table for this is very straight-forward. We simply create a column for each symbol in $\Sigma$ and a column for each state in $Q$. You may note that these two things correspond to the inputs of our $\delta$ function, so logically we fill the cells with the return values in the definition of $\delta$ for each pair of arguments:

| | $\delta$ | b | c |
|---|---|---|---|
| $\rightarrow$ | $q_0$ | $q_1$ | $q_0$ |
| | $q_1$ | $q_2$ | $q_0$ |
| $*$ | $q_2$ | $q_2$ | $q_2$ |

We also marked the initial state using $\rightarrow$ and the final state using $*$.

Now that you have the transition table, you may be asked to draw a transition diagram for this DFA. Note that in exam you may be not be asked

to construct a transition table as it is just as easy to construct a transition diagram from the definition of the DFA straight away.




### 2.2.1 Language Membership

In an examination setting, you're generally going to be asked one of two types of question: either one relating to the *metatheory* of the $\delta$ function (i.e. define $\hat{\delta}$ in terms of $\delta$) or you'll be given a bevy of words and must identify which ones are members of the automaton-generated language. If a word is in a language, you'll be expected to provide the derivation. If not, it generally suffices to say "nope, not gonna work" and move on. You've seen truckloads of these, but just for the above automaton, consider the following words:

1. $\varepsilon$
2. $cb$
3. $cbbcb$

Recall that we said earlier that $q_0 \in F_A \Rightarrow \varepsilon \in \text{Ł}(A)$. However, in this case $q_0 \notin F$ for the automaton. Therefore, the first word $\varepsilon \notin$ the language of the automaton $L(A)$.

Unfortunately, it not always that easy to conclude whether a word is in the language of an automaton. Most likely, we will need to make a formal derivation using the extended transition function $\hat{\delta}$.

For the second word, $cb$, we need to do exactly that. However, this is an extremely simple process based around looking up the definitions of $\delta$ and $\hat{delta}$. Let us begin by supplying $\hat{\delta}$ with the initial state $q_0$ and the word $cb$

$$\hat{\delta}(q_0, cb)$$

Our objective here is to see which state this function evaluates to, because then we can test whether that state is an element of the set of final states $F$. If it is in $F$, then the word is accepted by the automaton and otherwise it is rejected.

Recall that "evaluate" means the same as "simplify", so how can we simplify the expression above? Typically, if we have a function like $\hat{\delta}$, we can look up its definition. In the case of $\hat{\delta}$, there are *two* definitions. One is used if the second argument is $\varepsilon$, but $cb \neq \varepsilon$ and therefore we will use the second definition.

$$
\begin{aligned}
& \hat{\delta}(q_0, cb) \\
= \quad & \{ \quad \text{definition of } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(q_0, c), b)
\end{aligned}
$$

Note how we have taken the second definition and replaced the first parameter $s$ with $q_0$. The second parameter undoes concatenation and takes the first symbol in the second argument and binds it to $x$ so that we replace $x$ with $c$ and it binds the remaining word to $w$ so that we replace $w$ with $b$.

What can we do now? Well, we have two choices: we can either look up the definition of $\hat{\delta}$ again or we can look up the definition of $\delta$. Either choice would work, but recall that we are trying to *simplify* the expression. If we *expand* $\hat{\delta}$, the expression will actually become bigger for now. If we expand $\delta$, however, it will become smaller.

$$
\begin{aligned}
& \hat{\delta}(q_0, cb) \\
= \quad & \{ \quad \text{definition of } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(q_0, c), b) \\
= \quad & \{ \quad \text{definition of } \delta \quad \} \\
& \hat{\delta}(q_0, b)
\end{aligned}
$$

We managed to get rid of $\delta$, because it is defined as $q_0$ for the pair of arguments we supplied it with. Now we can repeat the same pattern until we

have processed all symbols in the word.

$$
\begin{aligned}
& \hat{\delta}(q_0, cb) \\
= \quad & \{ \quad \text{definition of } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(q_0, c), b) \\
= \quad & \{ \quad \text{definition of } \delta \quad \} \\
& \hat{\delta}(q_0, b) \\
= \quad & \{ \quad \text{definition of } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(q_0, b), \varepsilon) \\
= \quad & \{ \quad \text{definition of } \delta \quad \} \\
& \hat{\delta}(q_1, \varepsilon) \\
= \quad & \{ \quad \text{definition of } \hat{\delta} \quad \} \\
& q_1
\end{aligned}
$$

You may be wondering how we ended up with $\varepsilon$ in $\hat{\delta}(\delta(q_0, b), \varepsilon)$. The answer is simple if you recall that words are defined inductively (see section 1.7.1). To summarise: all words are defined by concatinating symbols from $\Sigma$ to other words. To do that, we need one word to start off with which is the empty word $\varepsilon$ and therefore if we keep removing symbols from a word, we will always end up with $\varepsilon$ in the end.

Now that we have finished evaluating the expression and ended up with $q_1$ as the resulting state, we can look up whether it is an element of $F$. It turns out that $q_1 \notin F$ and therefore $cb \notin L(A)$.

In this case we constructed a derivation and found out that the resulting state isn't a final state, therefore the whole word isn't accepted by the automaton. This example may well have been an easy one to spot for membership, but in an examination setting – when panic mode is at DEFCON 5 – it helps to learn to do these quickly, or even in your head.

Lastly, we can do the derivation for the third word, *cbbcb*, using the same

approach.

$$
\begin{aligned}
& \hat{\delta}(q_0, cbbcb) \\
= \quad & \{ \quad \text{definition of } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(q_0, c), bbcb) \\
= \quad & \{ \quad \text{definition of } \delta \quad \} \\
& \hat{\delta}(q_0, bbcb) \\
= \quad & \{ \quad \text{definition of } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(q_0, b), bcb) \\
= \quad & \{ \quad \text{definition of } \delta \quad \} \\
& \hat{\delta}(q_1, bcb) \\
= \quad & \{ \quad \text{definition of } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(q_1, b), cb) \\
= \quad & \{ \quad \text{definition of } \delta \quad \} \\
& \hat{\delta}(q_2, cb) \\
= \quad & \{ \quad \text{definition of } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(q_2, c), b) \\
= \quad & \{ \quad \text{definition of } \delta \quad \} \\
& \hat{\delta}(q_2, b) \\
= \quad & \{ \quad \text{definition of } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(q_2, b), \varepsilon) \\
= \quad & \{ \quad \text{definition of } \delta \quad \} \\
& \hat{\delta}(q_2, \varepsilon) \\
= \quad & \{ \quad \text{definition of } \hat{\delta} \quad \} \\
& q_2
\end{aligned}
$$

We can now see that $q_2 \in F$ which means that *cbbcb* is accepted by the automaton $A$.

## 2.3 The Natural Progression...

Unfortunately, deterministic finite automata can be remarkably clunky in some cases. Thankfully, **?** introduced *nondeterministic* finite automata

(NFAs), which represent the same class of languages as DFAs do, albeit in a more 'efficient' manner (where we judge 'efficiency' based on the number of states required to implement the automata). These languages are the *Type 3* or *regular* languages in the context of Chomsky's hierarchy.

## 2.4 Exercises

1. You and one of your friends play a matchstick game with $n$ matchsticks where each player is allowed to remove at most $m$ (but at least one) matchsticks from the pile during each turn. If there are no more matchsticks left at the beginning of a turn, then the player whose turn it is loses the game.

   Let the remaining number of matchsticks in the game be $r$, then an invariant of this game is that if $r \mod m + 1 = 0$ it is impossible for the current player to win against an opponent who plays the game perfectly.

   (a) Define a DFA which, given the number of remaining matchsticks in base 2, decides whether it is possible to win against an opponent who plays the game perfectly for $m = 2$. For example, suppose the input word is 110, then your DFA should accept the word while it should reject, for example, 10.

   (b) Draw the transition table for your DFA.

   (c) Draw the transition diagram for your DFA.

2. Given the following specification, implement the DFA from question 1 in Haskell.

$$
\begin{array}{lcl}
\textbf{data}\ Q & = & \ldots \\
\textbf{data}\ \Sigma & = & \ldots \\
\textbf{type}\ Word & = & \ldots \\
 & & \\
q_0 & :: & Q \\
final & :: & Q \to Bool \\
\delta & :: & (Q, \Sigma) \to Q \\
\hat{\delta} & :: & (Q, Word) \to Q \\
accept & :: & Word \to Bool
\end{array}
$$

# Chapter 3

# Nondeterministic Finite Automata

As mentioned in the previous chapter, nondeterministic finite automata represent the same class of languages as handled by DFAs, albeit in a 'cleaner' way. There are two key differenced compared to DFAs: the result type of the transition function $\delta$ allows us to return a set of states rather than just one and rather than an initial state we have a set of initial states. However, let us define this formally to be clear.

## 3.1 Definition

An NFA is given by a 5-tuple

**Nondeterministic Finite Automaton:**
$(Q, \Sigma, \delta, S, F)$

1. $Q$ is a finite set of states, precisely as in the definition for a DFA.

2. $\Sigma$ is a finite set of symbols constituting the alphabet, as for a DFA.

3. The $\delta$ function for NFAs differs from DFAs in that the function can return *multiple* states, corresponding to the result of taking each of the potential paths through the automaton based on the sequence given thus far (hence the nondeterminism). Formally, $\delta \in Q \times \Sigma \to \mathcal{P}(Q)$

and it is defined as

$$\hat{\delta}(P, \varepsilon) = P$$
$$\hat{\delta}(P, xw) = \hat{\delta}(\bigcup\{\, \delta(q, x) \mid q \in P \,\}, w)$$

The main implication of this definition is that there can be *multiple transitions* from any given state on the same input.

4. $S \subseteq Q$ is a finite set of initial states. A word can be processed using $\hat{\delta}$ taking any $s_0 \in S$ as the initial state as described in the previous chapter on DFAs.

5. $F \subseteq Q$ is the finite set of final states, as for DFAs.

*Note for the 2012/2013 academic year*: In lectures, you may use $\vartheta$ as the name for transition functions and $\hat{\vartheta}$ as the name for extended transition functions of NFAs. In this guide, we will use $\delta$ for the transition functions and $\hat{\delta}$ for extended transition functions, just like with DFAs. If we need to disambiguate between different transition functions, we will attach different subscripts to them. For example, we may use $\delta_D$ for the transition function of a DFA and $\delta_N$ for the transition function of a NFA. Remember that it doesn't matter what these functions are called, as long as it's clear what we mean.

## 3.2 Intuition

As mentioned, a nondeterministic automata accepts the same language as some deterministic one, however there's an intrinsic notion of 'guessing' when to make a certain transition at play here (I have, in the past, described NFAs as "just a DFA with a fairy in it").

By this, we mean that if a state has two possible transitions leading to different states predicated on the same input drawn from $\Sigma$, the transition which – some internal oracle reckons – is most likely to result in an accepting state is taken (for those of you who remember any content from G51APS, this is referred to as *angelic nondeterminism*). The presence of this oracle neatly circumvents the necessity of introducing additional states to a DFA to handle such differing transitions, as we are safe in the knowledge that we 'know' we can take the one that is needed in the case of ambiguity.

## 3.3 Example

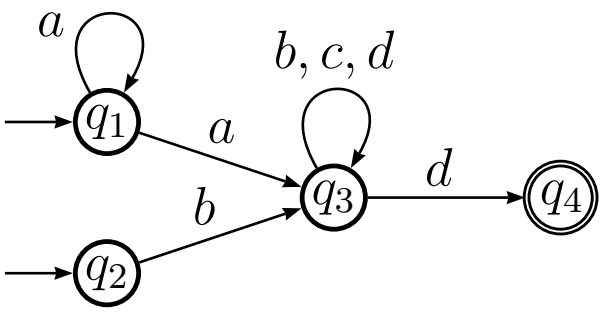
Consider the following definition for an NFA

$$N = (\{ q_1, q_2, q_3, q_4 \}, \Sigma, \delta, \{ q_1, q_2 \}, \{ q_4 \})$$

where $\Sigma = \{ a, b, c, d \}$ and the transition function $\delta$ is given by

$$\begin{aligned}
\delta(q_1, a) &= \{ q_1, q_3 \} \\
\delta(q_2, b) &= \{ q_3 \} \\
\delta(q_3, b) &= \{ q_3 \} \\
\delta(q_3, c) &= \{ q_3 \} \\
\delta(q_3, d) &= \{ q_3, q_4 \} \\
\delta(_, _) &= \emptyset
\end{aligned}$$

Just like with DFAs, we can draw a transition diagram to represent this NFA




As you can clearly see in the transition diagram, states are allowed to have multiple transitions for the same symbol which is something that we cannot do with DFAs, because each state there is only allowed to have one transition for each symbol in the input alphabet.

We can express the language of this NFA using a set comprehension

$$L(N) = (\{\, a^{n+1} \mid n \in \mathbb{N} \,\} \cup \{\, b \,\})\{\, \{\, b, c, d \,\}^i \mid i \in \mathbb{N} \,\}\{\, d \,\}$$

Consider the words *abdcd* and *abba*. The first is a member of the language, but the latter isn't. Just like with DFAs (see section 2.2.1), we can use the extended transition function to determine the state in which an automaton will end up in for a given word. Recall that the initial states in our example are $\{\, q_1, q_2 \,\}$, so that we initially have the following expression for *abdcd*

$$\hat{\delta}(\{\, q_1, q_2 \,\}, abdcd)$$

As you can see, the NFA allows us to be in multiple states at the same time so that we are initially in both state 1 and state 2. Now we can apply $\hat{\delta}$ to its arguments

$$\begin{aligned} & \hat{\delta}(\{\, q_1, q_2 \,\}, abdcd) \\ = \quad & \{ \quad \text{applying } \hat{\delta} \quad \} \\ & \hat{\delta}(\bigcup\{\, \delta(q, a) \mid q \in \{\, q_1, q_2 \,\} \,\}, bdcd) \end{aligned}$$

Unlike with DFAs, we can't apply $\delta$ just yet. Instead, we need to expand the set comprehension first.

$$\begin{aligned} & \hat{\delta}(\bigcup\{\, \delta(q, a) \mid q \in \{\, q_1, q_2 \,\} \,\}, bdcd) \\ = \quad & \{ \quad \text{set comprehension} \quad \} \\ & \hat{\delta}(\bigcup\{\, \delta(q_1, a), \delta(q_2, a) \,\}, bdcd) \end{aligned}$$

Now we can apply the transition function $\delta$ twice.

$$
\begin{aligned}
& \hat{\delta}(\bigcup\{\, \delta(q_1, a), \delta(q_2, a) \,\}, bdcd) \\
= \quad & \{ \quad \text{applying } \delta \quad \} \\
& \hat{\delta}(\bigcup\{\, \{\, q_1, q_3 \,\}, \emptyset \,\}, bdcd)
\end{aligned}
$$

$\delta$ for NFAs returns a set, so that we now have a set of sets of states. However, we only need a single set with all the states, so we use set union to achieve that.

$$
\begin{aligned}
& \hat{\delta}(\bigcup\{\, \{\, q_1, q_3 \,\}, \emptyset \,\}, bdcd) \\
= \quad & \{ \quad \text{set union} \quad \} \\
& \hat{\delta}(\{\, q_1, q_3 \,\}, bdcd)
\end{aligned}
$$

And from now on, we just repeat this process until we have processed all symbols in the input word.

$$
\begin{aligned}
& \hat{\delta}(\{\, q_1, q_3 \,\}, bdcd) \\
= \quad & \{ \quad \text{applying } \hat{\delta} \quad \} \\
& \hat{\delta}(\bigcup\{\, \delta(q, b) \mid q \in \{\, q_1, q_3 \,\} \,\}, dcd) \\
= \quad & \{ \quad \text{set comprehension, applying } \delta\text{, set union} \quad \} \\
& \hat{\delta}(\{\, q_3 \,\}, dcd) \\
= \quad & \{ \quad \text{applying } \hat{\delta} \quad \} \\
& \hat{\delta}(\bigcup\{\, \delta(q, d) \mid q \in \{\, q_3 \,\} \,\}, cd) \\
= \quad & \{ \quad \text{set comprehension, applying } \delta\text{, set union} \quad \} \\
& \hat{\delta}(\{\, q_3, q_4 \,\}, cd) \\
= \quad & \{ \quad \text{applying } \hat{\delta} \quad \} \\
& \hat{\delta}(\bigcup\{\, \delta(q, c) \mid q \in \{\, q_3, q_4 \,\} \,\}, d) \\
= \quad & \{ \quad \text{set comprehension, applying } \delta\text{, set union} \quad \} \\
& \hat{\delta}(\{\, q_3 \,\}, d) \\
= \quad & \{ \quad \text{applying } \hat{\delta} \quad \} \\
& \hat{\delta}(\bigcup\{\, \delta(q, d) \mid q \in \{\, q_3 \,\} \,\}, \varepsilon) \\
= \quad & \{ \quad \text{set comprehension, applying } \delta\text{, set union} \quad \} \\
& \hat{\delta}(\{\, q_3, q_4 \,\}, \varepsilon)
\end{aligned}
$$

$$
\begin{array}{ll}
= & \{ \quad \text{applying } \hat{\delta} \quad \} \\
\{ q_3, q_4 \}
\end{array}
$$

$q_4 \in F$ so that we can conclude that *abdcd* is accepted by the NFA. Now we do the same thing for *abba*.

$$
\begin{array}{ll}
 & \hat{\delta}(\{ q_1, q_2 \}, abba) \\
= & \{ \quad \text{applying } \hat{\delta} \quad \} \\
 & \hat{\delta}(\bigcup\{ \delta(q, a) \mid q \in \{ q_1, q_2 \} \}, bba) \\
= & \{ \quad \text{set comprehension, applying } \delta, \text{ set union} \quad \} \\
 & \hat{\delta}(\{ q_1, q_3 \}, bba) \\
= & \{ \quad \text{applying } \hat{\delta} \quad \} \\
 & \hat{\delta}(\bigcup\{ \delta(q, b) \mid q \in \{ q_1, q_3 \} \}, ba) \\
= & \{ \quad \text{set comprehension, applying } \delta, \text{ set union} \quad \} \\
 & \hat{\delta}(\{ q_3 \}, ba) \\
= & \{ \quad \text{applying } \hat{\delta} \quad \} \\
 & \hat{\delta}(\bigcup\{ \delta(q, b) \mid q \in \{ q_3 \} \}, a) \\
= & \{ \quad \text{set comprehension, applying } \delta, \text{ set union} \quad \} \\
 & \hat{\delta}(\{ q_3 \}, a) \\
= & \{ \quad \text{applying } \hat{\delta} \quad \} \\
 & \hat{\delta}(\bigcup\{ \delta(q, a) \mid q \in \{ q_3 \} \}, \varepsilon) \\
= & \{ \quad \text{set comprehension, applying } \delta, \text{ set union} \quad \} \\
 & \hat{\delta}(\emptyset, \varepsilon) \\
= & \{ \quad \text{applying } \hat{\delta} \quad \} \\
 & \emptyset
\end{array}
$$

We have reached the empty set and since there are no states in the empty set, there are also no final states. Therefore *abba* is *not* accepted by this NFA.

## 3.4 Subset Construction

Previously, we have said that DFAs and NFAs represent the same class of languages: the regular languages. This means that each language we can

represent using an NFA can also be represented using a DFA and vice versa.

*"But how do you know that DFAs and NFAs are equivalent?"*, I pretend to hear you cry. All deterministic automata are trivially nondeterministic (the transition function only returns *singleton sets* of states), but showing that the equivalence holds the other way around is somewhat trickier.

For starters, we need to be able to convert NFAs into DFAs. But how can we do that? An NFA can be in multiple states at once while a DFA can only be in one state at a time. Well firstly, let us recall that we use sets to represent the states we are currently in for an NFA. In the example above, for example, we were initially in $\{ q_1, _q 2 \}$ before we moved to $\{ q_1, q_3 \}$ and so on. It is important to note that the number of combinations of states we can be in at a time is finite. Specifically, we are restricted to $\mathcal{P}(Q)$ so that the combinations of states we can be in for $N$ are

$$\emptyset, \{ q_1 \}, \{ q_2 \}, \{ q_3 \}, \{ q_4 \}, \{ q_1, q_2 \}, \{ q_1, q_3 \}, \{ q_1, q_4 \}, \{ q_2, q_3 \},$$
$$\{ q_2, q_4 \}, \{ q_3, q_4 \}, \{ q_1, q_2, q_3 \}, \{ q_1, q_2, q_4 \}, \{ q_1, q_3, q_4 \}, \{ q_2, q_3, q_4 \},$$
$$\{ q_1, q_2, q_3, q_4 \}$$

The trick here is that each of these sets is going to represent a state in the DFA we are constructing from an NFA. This method of constructing a DFA from an NFA is called *subset construction*.

### 3.4.1 Definition

Let us now define subset construction formally. Suppose that you have an NFA $N = (Q_N, \Sigma, \delta_N, S_N, F_N)$, then we can construct a DFA

$$D = (Q_D, \Sigma, \delta_D, S_N, F_D)$$

where each state is represented by a set of states from $N$ so that

$$Q_D = \mathcal{P}(Q_N)$$
$$F_D = \{ P \mid P \in Q_D \wedge (P \cap F_N \neq \emptyset) \}$$

or in other words, the states in $D$ are the combinations of states in $N$ and the final states in $D$ are the combinations of states in $N$ where at least one state is a final state in $N$. Now we can define the transition function $\delta_D$ for $D$ as

$$\delta_D(P, s) = \bigcup \{ \delta_N(p, s) \mid p \in P \}$$

or again in other words, given a state $P$ (which represents a set of states in $N$) and a symbol $s$, we use the transition function for $N$ on each of $N$'s states in $P$ and then return the set union of all the results.

### 3.4.2 Lazy Subset Construction

In the exam, it is more likely that you will have to apply a variant of subset construction which is known as *lazy subset construction*. The key difference here is that lazy subset construction does not produce states in the resulting DFA which are unreachable and therefore it is easier for us to construct / draw, because obviously less states means less work.

As an example for lazy subset construction, let us take the NFA $N$ from section 3.3 and convert it into a DFA $D$ using lazy subset construction. The idea here is similar to subset construction as we continue to use sets of states from $N$ as states in $D$, but rather than generating them all using the powerset, we start in $S_N$ and see where we can go from there. This process can be informally defined as

1. Group all of the start states of the NFA into a single state: e.g. $S_N = \{ q_1, q_2 \}$ is now a state in $D$.

2. Define all of the transitions of $D$ by using the lazy subset construction method: consider the set comprising $S_N$ and all sets of states reached by the transition function of the original NFA (i.e. $\{ q_3 \}$, $\{ q_1, q_3 \}$, $\{ q_3, q_4 \}$). Now, list each possible transition from each element of $\Sigma$ like so:

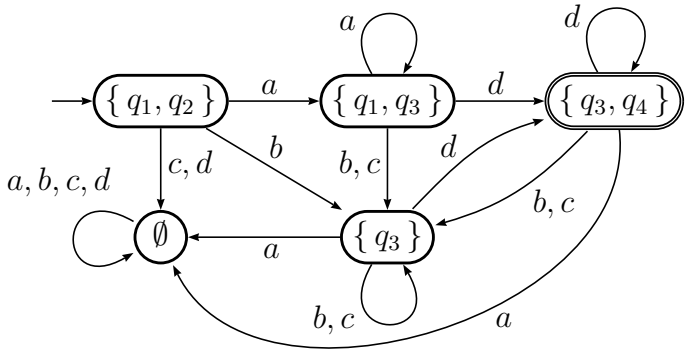
| | | $a$ | $b$ | $c$ | $d$ |
|---|---|---|---|---|---|
| $\rightarrow$ | $\{ q_1, q_2 \}$ | $\{ q_1, q_3 \}$ | $\{ q_3 \}$ | $\emptyset$ | $\emptyset$ |
| | $\{ q_1, q_3 \}$ | $\{ q_1, q_3 \}$ | $\{ q_3 \}$ | $\{ q_3 \}$ | $\{ q_3, q_4 \}$ |
| | $\{ q_3 \}$ | $\emptyset$ | $\{ q_3 \}$ | $\{ q_3 \}$ | $\{ q_3, q_4 \}$ |
| * | $\{ q_3, q_4 \}$ | $\emptyset$ | $\{ q_3 \}$ | $\{ q_3 \}$ | $\{ q_3, q_4 \}$ |
| | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |

3. Draw the resulting DFA as a state transition diagram.

4. ???

5. Profit!

Let us draw the DFA $D$ which we have just constructed from $N$ to illustrate the results. As we will see, each state is indeed named using a set of states from $N$ and there is exactly one transition for each symbol from each state so that it is obviously a valid DFA.




The immediate question is now that of equivalence: does this new DFA accept those words accepted by our original NFA, and reject those that don't?

Whilst ultimately this is a matter that can only be definitively handled using a formal proof, let us attempt to process the word *abdcd* using the derivation method for DFAs.

$$
\begin{aligned}
& \hat{\delta}(\{\, q_1, q_2 \,\}, abdcd) \\
= \quad & \{ \quad \text{applying } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(\{\, q_1, q_2 \,\}, a), bdcd) \\
= \quad & \{ \quad \text{applying } \delta \quad \} \\
& \hat{\delta}(\{\, q_1, q_3 \,\}, bdcd) \\
= \quad & \{ \quad \text{applying } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(\{\, q_1, q_3 \,\}, b), dcd) \\
= \quad & \{ \quad \text{applying } \delta \quad \} \\
& \hat{\delta}(\{\, q_3 \,\}, dcd) \\
= \quad & \{ \quad \text{applying } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(\{\, q_3 \,\}, d), cd) \\
= \quad & \{ \quad \text{applying } \delta \quad \} \\
& \hat{\delta}(\{\, q_3, q_4 \,\}, cd) \\
= \quad & \{ \quad \text{applying } \hat{\delta} \quad \} \\
& \hat{\delta}(\delta(\{\, q_3, q_4 \,\}, c), d)
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{ \quad \text{applying } \delta \quad \} \\
&\hat{\delta}(\{ q_3 \}, d) \\
&= \quad \{ \quad \text{applying } \hat{\delta} \quad \} \\
&\hat{\delta}(\delta(\{ q_3 \}, d), \varepsilon) \\
&= \quad \{ \quad \text{applying } \delta \quad \} \\
&\hat{\delta}(\{ q_3, q_4 \}, \varepsilon) \\
&= \quad \{ \quad \text{applying } \hat{\delta} \quad \} \\
&\{ q_3, q_4 \}
\end{aligned}
$$

Similarly, executing *abba* will result in a rejection. Voilà!

### 3.4.3 Proof

We shown how you can convert a NFA into a DFA using subset construction, but does this work for *all* NFAs? Let us formalise this question as:

$$\forall w \in \Sigma^*, \forall qs \in \mathcal{P}(Q_N), \hat{\delta}_N(qs, w) = \hat{\delta}_D(qs, w)$$

Or in other words, we want to show that, for any given word and any given set of initial states, the two transition functions are equal. Since $\Sigma^*$ can contain infinitely many words, this proof needs to be by induction on $w$. We begin using the base case, the empty word:

$$
\begin{aligned}
&\hat{\delta}_D(qs, \varepsilon) \\
&= \quad \{ \quad \text{applying } \hat{\delta}_D \quad \} \\
&qs \\
&= \quad \{ \quad \text{unapplying } \hat{\delta}_N \quad \} \\
&\hat{\delta}_N(qs, \varepsilon)
\end{aligned}
$$

Unsurprisingly, the base case was quite easy. Now let us assume that the extended transition functions are equivalent for all words. What if we add one more symbol to the word? *I.e.* let us consider the inductive case:

$$
\begin{aligned}
&\hat{\delta}_D(qs, xw) \\
&= \quad \{ \quad \text{applying } \hat{\delta}_D \quad \} \\
&\hat{\delta}_D(\delta_D(qs, x), w) \\
&= \quad \{ \quad \text{definition of } \delta_D \quad \}
\end{aligned}
$$

$$
\begin{aligned}
& \hat{\delta}_D(\bigcup\{\, \delta_D(p, x) \mid p \in qs \,\}, w) \\
= \quad & \{ \quad \text{induction hypothesis} \quad \} \\
& \hat{\delta}_N(\bigcup\{\, \delta_N(p, x) \mid p \in qs \,\}, w) \\
= \quad & \{ \quad \text{unapplying } \hat{\delta}_N \quad \} \\
& \hat{\delta}_N(qs, xw)
\end{aligned}
$$

As we can see, the extended transition functions are equivalent for all words and all sets of states. Thus we have proven that subset construction works as intended.

## 3.5 Examination relevance

The archetypical question that you will get in an examination setting generally follows this structure:

1. (Likely) Given the description of an extended transition function of an NFA, construct a diagram representing it. Don't forget to denote your initial and final state(s)!

2. (Guaranteed) Use the subset construction method (described elsewhere in this guide) to convert from the NFA to a DFA which accepts the same language.

3. (Optional) Use the table-filling algorithm to remove unnecessary 'dead states' from the resultant DFA.

# Chapter 4

# Regular Expressions

So far we have seen two equivalent approaches to constructing machines which can determine whether words are members of the languages represented by those automata or not. However, for practical purposes, it would make sense to come up with a simpler way of defining languages. In this chapter, we will introduce *regular expressions*.

You may have come across this method before in, for example, text editors where they are used to find and replace strings in a document. However, it is worth noting that the "regular expressions" which are commonly implemented in such programs or libraries aren't actually true regular expressions!

## 4.1 Definition

The definition of regular expressions is inductive (see section 1.5) and each case represents a language. This means that we can define simple languages using the base cases of the regular expressions and then we can use the recursive cases, which allow us to combine other regular expressions with each other, to construct more complicated languages.

### 4.1.1 Syntax of Regular Expressions

The first base case of the regular expressions is $\emptyset$. It represents the language which contains exactly nothing.

$\emptyset$ is a regular expression

Note that $\emptyset$ does not represent the empty set here, but instead it denotes a regular expression. We mentioned before that each regular expression represents a language, so let us define the language of $\emptyset$ as

$$L(\emptyset) = \emptyset \tag{4.1}$$

Unintuitively, we have used the $\emptyset$ symbol to denote both, the regular expression and the empty set, here. However, we can infer from the context which of the two we are using. It wouldn't make sense for us to use the empty set in $L(\emptyset)$ because then we would be saying "the language of the empty set". Therefore, we are obviously referring to the regular expression $\emptyset$.

The regular expression $\varepsilon$ describes the set of just the empty word. Note that this is different from the empty set: the empty word describes one word with an absence of symbols, that is it has length zero whereas the empty set describes a language containing zero words.

$\varepsilon$ is a regular expression

We mentioned that the language of $\varepsilon$ is the language consisting of just the empty word:

$$L(\varepsilon) = \{ \varepsilon \} \tag{4.2}$$

Finally we say that we can construct a regular expression using any symbol in the alphabet $\Sigma$.

$\forall x \in \Sigma, x$ is a regular expression

And the language of such a regular expression is defined as

$$\forall x \in \Sigma, L(x) = \{ x \} \tag{4.3}$$

Now that we have defined the base cases, we can start to build more complex regular expressions using the recursive cases. In the following definitions, we assume that $\alpha$ and $\beta$ are regular expressions.

The first of our recursive cases is denoted using the + symbol:

$\alpha + \beta$ is a regular expression

It corresponds to the union of two languages:

$$L(\alpha + \beta) = L(\alpha) \cup L(\beta) \tag{4.4}$$

Next, we have concatenation which we denote by writing $\alpha\beta$:

$\alpha\beta$ is a regular expression

This regular expression corresponds to the concatenation of two languages:

$$L(\alpha\beta) = L(\alpha)L(\beta) \tag{4.5}$$

The last of the recursive cases is $\alpha^*$:

$\alpha^*$ is a regular expression

Intuitively, it corresponds to Kleene Star:

$$L(\alpha^*) = L(\alpha)^* \tag{4.6}$$

Finally, we can surround a regular expression by parenthesis:

$(\alpha)$ is a regular expression

This only serves to resolve ambiguities when we write down regular expression and is not strictly a new case. It simply corresponds to the language of $\alpha$:

$$L((\alpha)) = L(\alpha) \tag{4.7}$$

### 4.1.2 Examples

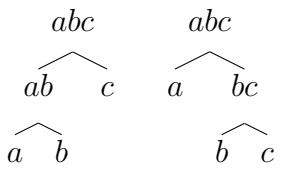
For illustration, we will show a few examples of regular expressions below and show how we can convert them into languages using the rules from the previous section. Suppose that $\Sigma = \{ a, b, c \}$ in the examples below.

We will begin with a simple example. Let us consider the regular expression $\emptyset$. It consists of only one base case and we can compute its language by simply taking the meaning of that base case:

$$\begin{array}{ll} & L(\emptyset) \\ = & \quad \{ \quad \text{language of } \emptyset \text{ (rule 4.1)} \quad \} \\ & \emptyset \end{array}$$

Next, we will look at *abc*. If we look at our definition of the regular expressions, we can easily see that this expression must be constructed using several base and recursive cases. It may help to think of expressions which

were constructed using inductive definitions as trees. There are two possible interpretations of *abc* as trees which are given below:




Note that there are two trees because we haven't specified whether concatenation is left or right associative. For larger expressions, there will likely be many different interpretations, but luckily for us, it doesn't matter which ones we use.

As you can see, the leaves of the both trees represent regular expressions which were constructed using the base cases and the nodes represent recursive cases. We can now use this knowledge to convert *abc* into a language. Suppose we are using the first tree:

$$
\begin{aligned}
& L(abc) \\
= \quad & \{ \quad \text{concatenation of } ab \text{ and } c \text{ (rule 4.5)} \quad \} \\
& L(ab)(c) \\
= \quad & \{ \quad \text{concatenation of } a \text{ and } b \text{ (rule 4.5)} \quad \} \\
& L(a)L(b)L(c)
\end{aligned}
$$

We have reached the leaves of the tree which, of course, represent expressions constructed using the base cases. It now becomes very easy for us to compute their languages and we will show how to do that in detail below:

$$
\begin{array}{ll}
 & L(a)L(b)L(c) \\
= & \quad \{ \quad \text{base case, } L(a) = \{\, a \,\} \text{ (rule 4.3)} \quad \} \\
 & \{\, a \,\}L(b)L(c) \\
= & \quad \{ \quad \text{base case, } L(b) = \{\, b \,\} \text{ (rule 4.3)} \quad \} \\
 & \{\, a \,\}\{\, b \,\}L(c) \\
= & \quad \{ \quad \text{base case, } L(c) = \{\, c \,\} \text{ (rule 4.3)} \quad \} \\
 & \{\, a \,\}\{\, b \,\}\{\, c \,\} \\
= & \quad \{ \quad \text{concatenation of } \{\, a \,\} \text{ and } \{\, b \,\} \quad \} \\
 & \{\, ab \,\}\{\, c \,\} \\
= & \quad \{ \quad \text{concatenation of } \{\, ab \,\} \text{ and } \{\, c \,\} \quad \} \\
 & \{\, abc \,\}
\end{array}
$$

In this example, we have shown each step explicitly, but it is possible to *e.g.* translate all the base cases or to concatenate all languages at once to save time. You are encouraged to combine groups of similar steps into one once you feel comfortable doing so. Note that this is merely done for convenience and doesn't differ from the more lengthy calculation. For example, we could write the above more concisely as:

$$
\begin{array}{ll}
 & L(a)L(b)L(c) \\
= & \quad \{ \quad \text{base cases} \quad \} \\
 & \{\, a \,\}\{\, b \,\}\{\, c \,\} \\
= & \quad \{ \quad \text{concatenation of languages} \quad \} \\
 & \{\, abc \,\}
\end{array}
$$

Finally, we present $ab+c^*\emptyset+\varepsilon c$ as a more advanced example which uses most of the syntax provided by the regular expressions. Note that we assume that $*$ has the highest precedence, followed by concatenation and that $+$ has the lowest precedence:

$$
\begin{array}{ll}
 & L(ab + c^*\emptyset + \varepsilon c) \\
= & \quad \{ \quad \text{plus (rule 4.4)} \quad \} \\
 & L(ab) \cup L(c^*\emptyset) \cup L(\varepsilon c) \\
= & \quad \{ \quad \text{concatenation (rule 4.5)} \quad \} \\
 & L(a)L(b) \cup L(c^*)L(\emptyset) \cup L(\varepsilon)L(c) \\
= & \quad \{ \quad \text{kleene star (rule 4.6)} \quad \}
\end{array}
$$

$$
\begin{aligned}
& L(a)L(b) \cup L(c)^* L(\emptyset) \cup L(\varepsilon)L(c) \\
= \quad & \{ \quad \text{base cases} \quad \} \\
& \{ a \}\{ b \} \cup \{ c \}^* \emptyset \cup \{ \varepsilon \}\{ c \} \\
= \quad & \{ \quad \text{property that } L\emptyset = \emptyset \quad \} \\
& \{ a \}\{ b \} \cup \emptyset \cup \{ \varepsilon \}\{ c \} \\
= \quad & \{ \quad \text{concatenation} \quad \} \\
& \{ ab \} \cup \emptyset \cup \{ c \} \\
= \quad & \{ \quad \text{set union} \quad \} \\
& \{ ab, c \}
\end{aligned}
$$

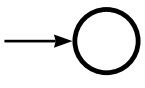
## 4.2 Converting Regular Expressions to NFAs

Since DFAs, NFAs and regular expressions all represent the same class of languages and we have already seen how to convert from an NFA to a DFA and vice versa, it would be useful if we could convert regular expressions as well. Specifically, we will introduce how to convert a regular expression into an NFA. DFAs can then be constructed from the resulting NFAs using subset construction (see section 3.4).

We have tried to keep the examples in this guide simple, but we recommend that you look at the more elaborate examples in the lecture notes which clearly demonstrate all aspects of the conversion process if you struggle with the written explanations below.

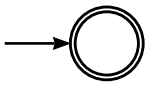
### 4.2.1 $N(\emptyset)$

We know that $L(\emptyset) = \emptyset$, so all we need to do here is to construct an NFA which rejects every word. Any NFA which has no final states will reject all words, but the simplest is the one shown below which consist of a single initial state:



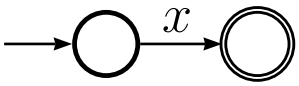
### 4.2.2 $N(\varepsilon)$

$L(\varepsilon) = \{ \varepsilon \}$ so that we only need a single, accepting state in the NFA. We start in the initial state and if the input word is the empty word, then we are already in an accepting state. If the input word is not empty, then there are no transitions for the symbols and we will reject the word:



### 4.2.3 $N(x)$

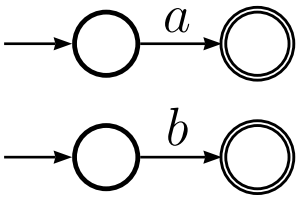
$L(x) = \{ x \}$ which means that we need to test if the first (and only) symbol in the input word is $x$. If so, we accept it, and otherwise, we will reject it:




### 4.2.4 $N(\alpha + \beta)$

It is surprisingly easy to construct the NFA for this recursive case since $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$. We simply need to construct $N(\alpha)$ and $N(\beta)$ before putting them together in one NFA.

For example, suppose you are asked to construct an NFA for the regular expression $a + b$, then you would first construct $N(a)$ and $N(b)$ before constructing $N(a + b)$ as

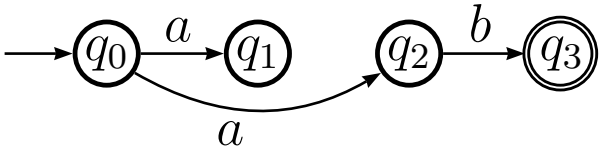



The result in this example contains the four states shown above. Feel free to test that this NFA accepts $a$ and $b$, but nothing else.

### 4.2.5 $N(\alpha\beta)$

Intuitively, the regular expression $\alpha\beta$ means "$\alpha$ followed by $\beta$" so that the resulting NFA represents exactly that. We will once again use an example to

demonstrate how this works, before explaining it formally. Suppose that you have the regular expression $ab$, then you need to construct $N(a)$ and $N(b)$ before merging them together as shown below:




Conceptually, we first put the two NFAs for the sub-expressions together. Then, all final states in $N(\alpha)$ become non-final and all initial states in $N(\beta)$ become non-initial states. Lastly, we connect all states in $N(\alpha)$ from which there exists a transition to a final state in $N(\alpha)$ to all initial states in $N(\beta)$.

We also need to be aware of one tricky case: if final states in $N(\alpha)$ are also initial states, then all the initial states in $N(\beta)$ remain as such in $N(\alpha\beta)$! The reasoning is that, if there are states in an NFA which are both initial and accepting, then the NFA accepts the empty word and we need to be able to skip straight to the words which are accepted by $N(\beta)$.

More formally, given any two regular expressions $\alpha$ and $\beta$ resulting in $N(\alpha) = (Q_\alpha, \Sigma, \delta_\alpha, S_\alpha, F_\alpha)$ and $N(\beta) = (Q_\beta, \Sigma, \delta_\beta, S_\beta, F_\beta)$, we define $N(\alpha\beta)$ to be (assuming that the states in both NFAs are distinct):

$$(Q_\alpha \cup Q_\beta, \Sigma, \delta_{\alpha\beta}, S_{\alpha\beta}, F_\beta)$$

This is pretty straight-forward except for $\delta_{\alpha\beta}$ and $S_{\alpha\beta}$. Recall that the set of initial states in the resulting NFA needs to account for the case where at least one final state in $N(\alpha)$ is also an initial state:

$$S_{\alpha\beta} = \begin{cases} S_\alpha \cup S_\beta & \text{if } \exists q \in Q_\alpha, q \in S_\alpha \wedge q \in F_\alpha \\ S_\alpha & \text{otherwise} \end{cases}$$

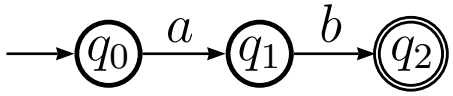
The new transition function has three cases: the last two simply use the old transition functions depending on which NFA the current state originated from.

However, we also have one special case for states in $N(\alpha)$ if the transition for them, using $x$, would lead to a final state. As described above, we then add transitions to all initial states in $N(\beta)$:

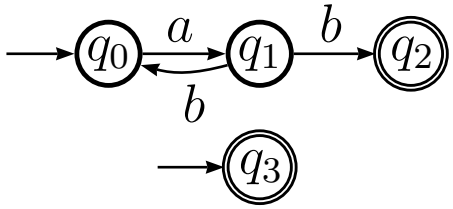
$$\delta_{\alpha\beta}(q, x) = \begin{cases} S_\beta \cup \delta_\alpha(q, x) & \text{if } q \in Q_\alpha \wedge \delta_\alpha(q, x) \cap F_\alpha \neq \emptyset \\ \delta_\alpha(q, x) & \text{if } q \in Q_\alpha \\ \delta_\beta(q, x) & \text{if } q \in Q_\beta \end{cases}$$

### 4.2.6 $N(\alpha^*)$

Suppose that we want to construct a NFA for $(ab)^*$, then we first construct the NFA for $ab$ - we have shown how to do that in the previous section. For simplicity, we have removed "useless" states from the resulting NFA and given names to the remaining ones:




Now, in order to construct $N((ab)^*)$ from this, we need to do two things. Firstly, we need to look at all the transitions which lead to final states: in this case, there is only one such transition from $q_1$ to $q_2$ for symbol $b$. For each of these transitions, we need to add new transitions from the origin state ($q_1$ here) to each initial state ($q_0$ is the only initial state here). In our example, this results in one new transition from $q_1$ to $q_0$ for $b$. Secondly, we need to remember that $L^*$ includes $\varepsilon$ for any language $L$, so that we need a new state for the empty word which is both an initial state and a final state:



# Chapter 5

# Table-Filling Algorithm

In the previous chapter we have seen how regular expressions can be converted into NFAs and these can then be converted into DFAs using subset construction (see section 3.4). However, we will often end up with very large DFAs that may be more complex than they need to be.

A DFA can be simplified using the *Table-Filling Algorithm* which is based around the idea of states which we consider to be *undistinguishable* (i.e. states which are identical to each other in terms of transitions and behaviour). The algorithm consits of the following *two* steps

1. All pairs of states where one state is a final state and the other one isn't are distinguishable.

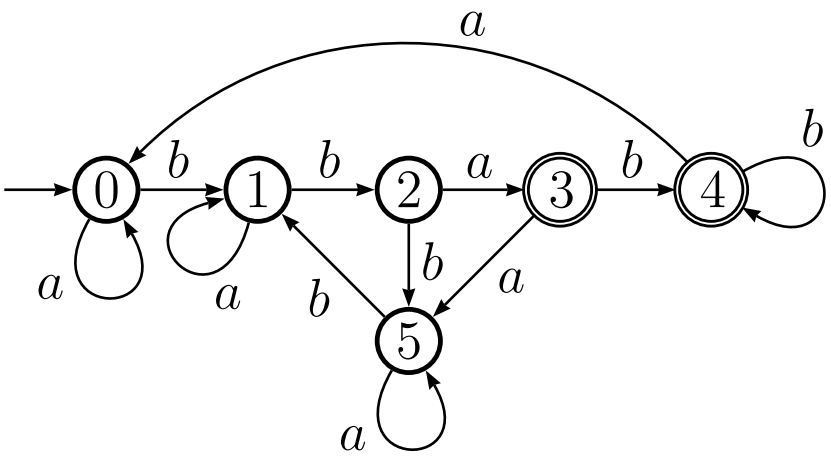
2. All pairs of states for which there is a transition to a distinguishable state is also distinguishable.

We can express this algorithm more formally as

1. $\forall p, q \in Q$ if $p \in F$ and $q \notin F$ then $(p, q)$ is distinguishable.

2. $\forall p, q \in Q$ if $\exists x \in \Sigma$ such that $(\delta(p, a), \delta(q, a))$ is distinguishable, then $(p, q)$ is also distinguishable.

As an example, let us consider the following DFA named $D$ from the fourth coursework




We can now minimise $D$ using the Table-Filling Algorithm. This can easily be done using a table of all unique pairs of states in $D$ in which when the mark pairs which are distinguishable, hence *Table-Filling* Algorithm.

Constructing the table of uniqe pairs is a fairly easy process: we simply order our states in some way (in this case we will simply use the state numbers), then we use all states except for the last as columns and all states except for the first as rows.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 5 | | | | | |
| 4 | | | | | |
| 3 | | | | | |
| 2 | | | | | |
| 1 | | | | | |

We start by marking all pairs of states $(p, q)$ for which $p \in F$ and $q \notin F$ as distinguishable as stated in the first part of the algorithm.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 5 | | | | x | x |
| 4 | x | x | x | | |
| 3 | x | x | x | | |
| 2 | | | | | |
| 1 | | | | | |

Then we can mark all pairs of states $(p, q)$ as distinguishable if $\exists x \in \Sigma$ such that $(\delta(p, x), \delta(q, x))$ is distinguishable as stated in the second part of the algorithm.

Sometimes we may not immediately be able to deduce whether a state is distinguishable, so let us do this process in detail. Let us write down a list of pairs of states for which we need to decide whether they are distinguishable or not.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 5 | | | | x | x |
| 4 | x | x | x | | |
| 3 | x | x | x | | |
| 2 | | | | | |
| 1 | | | | | |

(0,1) (0,2) (0,5)

(1,2) (1,5) (2,5)

(3,4)

Note that it does not matter in which order we look at the pairs of states, so let us simply begin with $(0, 1)$ and symbol $a$. We get $(\delta(0, a), \delta(1, a)) = (0, 1)$ which is the same pair of states as the one for which we are trying to decide if it distinguishable or not. This is no use to us. Now let us try the symbol $b$ for the same pair of states. We get $(\delta(0, b), \delta(1, b)) = (1, 2)$ which we haven't looked at yet. Let us write down this information so that if we find $(1, 2)$ to be distinguishable, we can also mark $(0, 1)$ as distinguishable.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 5 | | | | x | x |
| 4 | x | x | x | | |
| 3 | x | x | x | | |
| 2 | | | | | |
| 1 | | | | | |

~~(0,1)~~ (0,2) (0,5)

(1,2)

(1,2) (1,5) (2,5)

(3,4)

Next, we can look at $(0, 2)$. For $a$, we get $(\delta(0, a), \delta(2, a)) = (0, 3)$ which is already marked as distinguishable, so we can mark $(0, 2)$ as well. Note that we currently have no dependencies on $(0, 2)$ so we can't automatically mark any other states as distinguishable.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 5 | | | | x | x |
| 4 | x | x | x | | |
| 3 | x | x | x | | |
| 2 | x | | | | |
| 1 | | | | | |

~~(0,1)~~ (1,2) | ~~(0,2)~~ | (0,5)

(1,2) | (1,5) | (2,5)

(3,4)

For $(0, 5)$ and symbol $a$, we get $(\delta(0, a), \delta(5, a)) = (0, 5)$ which is of no use to us. For $b$, we get $(\delta(0, b), \delta(5, b)) = (1, 1)$ which is not a unique pair of states and therefore not in our table. In a case where we get a pair consisting of the same state, we can't deduce any information from it. We have also shown that both transitions for $(0, 5)$ are indistinguishable so that we can already conclude that states 0 and 5 are *not* distinguishable.

We now proceed with $(1, 2)$ and $a$ for which $(\delta(1, a), \delta(2, a)) = (1, 3)$ which is distinguishable so that $(1, 2)$ is consequently also distinguishable. Note that we wrote down earlier that $(0, 1)$ is distinguishable if $(1, 2)$ is distinguishable so that we can now mark both pairs of states as distinguishable.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 5 | | | | x | x |
| 4 | x | x | x | | |
| 3 | x | x | x | | |
| 2 | x | x | | | |
| 1 | x | | | | |

~~(0,1)~~ ~~(1,2)~~ | ~~(0,2)~~ | ~~(0,5)~~

~~(1,2)~~ | (1,5) | (2,5)

(3,4)

Let us now look at $(1, 5)$ and $b$ for which $(\delta(1, a), \delta(5, a)) = (1, 5)$ so that's no use. For $b$, we get $(\delta(1, b), \delta(5, b)) = (2, 1)$ which refers to the same pair of states as $(1, 2)$ which is already marked as distinguishable so that we can mark this pair of states as well.

For $(2, 5)$ and $a$, we get $(\delta(2, a), \delta(5, a)) = (3, 5)$ which is distinguishable so that we can mark $(2, 5)$ as distinguishable as well.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 5 | | x | x | x | x |
| 4 | x | x | x | | |
| 3 | x | x | x | | |
| 2 | x | x | | | |
| 1 | x | | | | |

~~(0,1)~~ ~~(0,2)~~ ~~(0,5)~~

~~(1,2)~~

~~(1,2)~~ ~~(1,5)~~ ~~(2,5)~~

(3,4)

Lastly, we have got $(3, 4)$. For $a$, we get $(\delta(3, a), \delta(4, a)) = (5, 0)$ for which we know it is not distinguishable. For $b$, we get $(\delta(3, b), \delta(4, b)) = (4, 4)$ which is of no use to us. We have no more options left and therefore $(3, 4)$ is *not* distinguishable.

Let us take a final look at our table:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 5 | | x | x | x | x |
| 4 | x | x | x | | |
| 3 | x | x | x | | |
| 2 | x | x | | | |
| 1 | x | | | | |

~~(0,1)~~ ~~(0,2)~~ ~~(0,5)~~

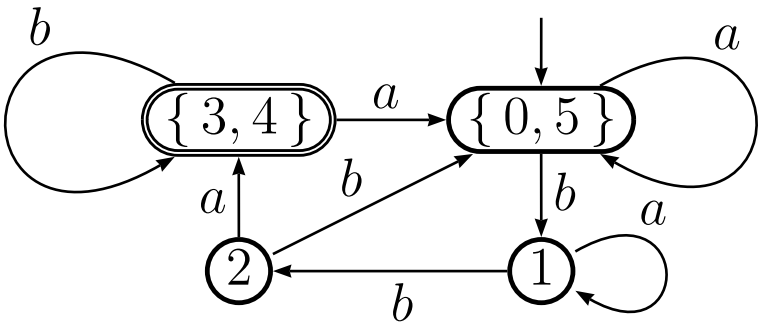
~~(1,2)~~

~~(1,2)~~ ~~(1,5)~~ ~~(2,5)~~

~~(3,4)~~

Finally we can construct a new DFA by merging all states which are not marked as distinguishable. Doing this is actually very simple if we think about what it means for two states to be indistinguishable. If we cannot distinguish between two states, it means that the transitions *originating* from them have the same targets for all symbols in the alphabet and both states are either final or normal states, but not both.

We replace each pair of states which is not distinguishable with one new state (which we label here with the identifiers of both old states). All transitions *to* the two old states now point at the new state and all transitions *from* the old states now originate from the new state. The resulting DFA for the Table-Filling Algorithm applied to $D$ is shown below:


b
a
{3, 4}
a
{0, 5}
a
b
b
a
2
1
b

# Chapter 6

# Pumping Lemma

Sometimes we will want to prove that a language $L$ is not regular. To do this, we will need to use the negated *pumping lemma*. However, before we can negate the pumping lemma, we first need to define what the pumping lemma actually is

Regular Language $\Rightarrow$ Pumping Property

We are saying that if a language is regular, then it has the pumping property which is defined as

$\exists n \in \mathbb{N},$
$\forall w \in L, |w| \geq n,$
$\exists x, y, z$ such that $w = xyz, |xy| \leq n, y \neq \varepsilon$
$\forall i \in \mathbb{N}, xy^i z \in L$

However, we want to use the pumping lemma to show that a language is not regular. In other words, we want to use the negation (contrapositive) of the pumping lemma:

$\neg$Pumping Property $\Rightarrow$ $\neg$Regular Language

In order to negate the pumping property, we simply change the quantifiers and change the last $\in$ to $\notin$.

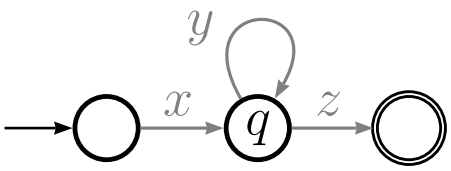
$\forall n \in \mathbb{N},$
$\exists w \in L, |w| \geq n,$
$\forall x, y, z$ such that $w = xyz, |xy| \leq n, y \neq \varepsilon$

$$\exists i \in \mathbb{N}, xy^iz \notin L$$

Now, if we want to prove that a language is not regular, we simply need to show that some word $w$ is not in the language if we pump up a word using $i$, for all $x$, $y$ and $z$.

## 6.1 Proof

Let us assume that we have a regular language $L$, then we can represent that language using a DFA. Let us refer to that DFA as $D$ so that we can then say $L = L(D)$. We will also assume that $D$ has $n$-many states. If $D$ accepts a word $w$ such that $|w| \geq n$, then we must have visited a state in $D$ at least twice. Let us refer to that state as $q$ so that we can draw $D$ as




Note that $x$, $y$ and $z$ are words rather than symbols in the representation above. State $q$ is chosen such that it is the first loop in $D$ which means that $|xy| \leq n$. We also know that $y$ cannot be $\varepsilon$ because otherwise there would be no loop. Lastly, we know that we can repeat the loop for $y$ as many times as we like or not at all and still get a word which is accepted by $D$. Therefore, $\forall i \in \mathbb{N}, xy^iz \in L$.

## 6.2 Example: Palindromes

As an example for the pumping lemma, let us consider the language of palindromes $L = \{w \mid w \in \Sigma^*, w = w^R\}$ over an alphabet $\Sigma = \{0, 1\}$.

In order to do this proof, it helps to think about it as an argument you are having with someone such as a younger sibling. Looking at the negated pumping lemma above, whenever we come across a $\forall$, our younger sibling gets to make an argument. However, he or she won't tell us what exactly their argument is. They will only tell us a little bit about it. For example,

for $\forall n \in \mathbb{N}$, they won't tell us which number $n$ is. Instead, they will only let us know that $n \in \mathbb{N}$.

When we come across an $\exists$ such as the one in $\exists w \in L, |w| \geq n$, we get to choose an argument. In other words, we can choose any $w$ as long as we can guarantee that it is at least $n$ symbols long (because of the $|w| \geq n$ constraint in the pumping lemma) and that it is a word in $L$. In this case, we could choose a word like $w = 0^n 10^n$. Let us quickly check that our choice fits the two constraints we mentioned earlier: We have $n+1+n$ symbols which is $\geq n$ so that's fine. We also know that the word must be a palindrome, regardless of what $n$ actually is because we always have the same number of 0s on both sides.

Now it is our sibling's turn again and we know that they choose three variables $x, y, z$ such that $w = xyz, |xy| \leq n, y \neq \varepsilon$. That's all we know, but we can make a few educated guesses about what $x, y$ and $z$ *might* be. We know that the first $n$ symbols in our word are all 0 and that $|xy| \leq n$. Consequently, all symbols in $x$ and $y$ must be 0s. According to the pumping lemma, $y \neq \varepsilon$ so that $y$ must consist of *at least* one 0. It can consist of more than one, but there must be at least one 0 in it.

With the knowledge about $x$ and $y$ we have gained from the previous paragraph, we can now look at the final part of the pumping lemma. Once again, it is our turn to make an argument. We want to find an $i \in \mathbb{N}$ such that $xy^i z \notin L$. Because we know that $y$ consists of at least one 0, we can easily tell that if $i > 1$, there will be more $0s$ on the left of the word than there are on the right. Similarly, if $i = 0$ then there will be more 0s on the right than there are on the left.

For illustration, let us assume that our sibling chose $x = 0^{n-1}$, $y = 0$ and $z = 0^n$. Now let $i = 0$

$$
\begin{array}{ll}
 & xy^0z \\
= & \quad \{ \quad y^0 = \varepsilon \quad \} \\
 & xz \\
= & \quad \{ \quad \text{definitions of } x \text{ and } z \quad \} \\
 & 0^{n-1}0^n
\end{array}
$$

which shows that there is one less 0 on the left than there is on the right. Consequently, the word is not a palindrome and we have proven that the language of palindromes is not regular.

## 6.3 Example: $n \neq m$

Let us consider one more example which is a bit more difficult. Suppose that $L = \{0^n 1^m \mid n \neq m\}$ and $\Sigma = \{0, 1\}$ then how do we prove that $L$ is not a regular language?

**Hey, listen!** It is very easy to make a mistake here if you do not fully understand the pumping lemma yet. To demonstrate, let us make the common mistake here so we can see why it is **not** a valid proof.

Our sibling chooses $k \in \mathbb{N}$. Note that we use $k$ this time instead of $n$, because we already have a variable named $n$ in the set comprehension for $L$. So in order to avoid confusion between the two variables, we name this one $k$.

Now we get to choose a word. This time, we only get to choose $n$ and $m$ such that $n \neq m$. Let us simply choose $n = k$ and $m = n + 1$ so that our word is $0^k 1^{k+1}$ which is obviously a valid choice.

When our sibling chooses $x, y$ and $z$, he or she will be limited to 0s for $x$ and $y$ just like in the example for the language of palindromes. So we know that $xy$ will be all 0s and $y$ will be at least one 0.

Let us assume that our sibling chose $x = 0^{k-1}$, $y = 0$ and $z = 1^{k+1}$. Now if we choose $i = 2$, everything seems fine because

$$
\begin{array}{ll}
& xy^2z \\
= & \quad \{ \quad y^2 = yy \quad \} \\
& xyyz \\
= & \quad \{ \quad \text{definitions of } x, y \text{ and } z \quad \} \\
& 0^{k-1}001^{k+1}
\end{array}
$$

which has the same number of 0s and 1s (i.e. $n = m$) and we might think that this proves that $L$ is not regular. However, our proof is not valid, because our choice for $i$ needs to work for any $x$, $y$ and $z$ our sibling can possibly choose. For example, if $x = 0^{k-2}$, $y = 00$ and $z = 1^{k+1}$ then

$$
\begin{array}{ll}
& xy^2z \\
= & \quad \{ \quad y^2 = yy \quad \} \\
& xyyz \\
= & \quad \{ \quad \text{definitions of } x, y \text{ and } z \quad \} \\
& 0^{k-2}00001^{k+1}
\end{array}
$$

which has a different number of 0s and 1s. Therefore this naive approach to proving that $L$ is not regular doesn't work.

In order to answer this question, we need to split the problem into simpler ones. Specifically, we can split the problem into two cases

$$L_a = \{\, 0^n 1^m \mid n < m \,\}$$
$$L_b = \{\, 0^n 1^m \mid n > m \,\}$$

If we can prove that neither the language for $n < m$ nor the one for $n > m$ is regular, then it implies that the language for $n \neq m$ is not regular.

*Proof.* Our sibling chooses $k \in \mathbb{N}$. Just like before, we use $k$ instead of $n$, because we already have a variable named $n$ in the set comprehension for $L$.

At this point, we get to choose a word and as above, we only get to choose $n$ and $m$ such that $n < m$. Let us simply choose $m = k + 1$ and $n = m - 1$ so that our word ends up being $w = 0^k 1^{k+1}$ which is valid because it has more or equal to $k$ symbols and $n < m$.

Just like in the example for the language of palindromes, our sibling is nailed down to choosing 0s for $x$ and $y$ and $y$ must be at least one 0. Consequently, if we pump $y$ up using $i$, we will get a number of 0s on the left which is equal or greater than the number of 1s on the right. Therefore, $L_a$ is not regular. ■

We have proven that $L_a$ is not regular, but we still need to prove that $L_b$ is also not regular.

*Proof.* Our sibling chooses $k \in \mathbb{N}$ and now we get to choose a word such that $n > m$. Let $n = k + 1$ and $m = k$ then $w = 0^{k+1} 1^k$ which satisfies both of the given constraints. This means that we have once again forced our sibling to choose $xy$ to be all 0s which also means that $y$ must consist of at least one 0. Regardless of how many 0s there are in $y$, if we choose $i = 0$, there will be less or equal as many 0s as there are 1s. Consequently, $L_b$ is also not regular. ■

Now that we have proven that both $L_a$ and $L_b$ are not regular, we can conclude that $L$ must not be regular. Also note that there is an alternative proof without case analysis in the solutions for the coursework[1].

[1] `http://www.cs.nott.ac.uk/~txa/g52mal/Exercises/solutions4.pdf`

# Chapter 7

# Pushdown Automata

We have encountered languages, such as the language of palindromes, which are not in the class of regular languages and therefore we cannot represent them using DFAs, NFAs or as regular expressions. In this chapter we will be introducing one way of representing *context-free* languages.

*Pushdown Automata* are essentially the same as NFAs (see chapter 3) which additionally come equipped with a *stack* where each element represents exactly one symbol from the *stack alphabet*. Unlike DFAs and NFAs, pushdown automata (PDAs) are defined using *two* alphabets: the *input alphabet* $\Sigma$ is the same as before and additionally we have the stack alphabet $\Gamma$ (*Gamma*).

We formally define a PDA using a 7-tuple as follows

**Pushdown Automaton:**
$(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

1. $Q$ is the finite set of states, as before.
2. $\Sigma$ is the input alphabet, as before.
3. $\Gamma$ is the stack alphabet.
4. $\delta$ is the transition function. It is an element of

$$Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma \to \mathcal{P}_{\text{fin}}(Q \times \Gamma^*)$$

   Or in plain English: $\delta$ is a function which, if given a state, a symbol from the input alphabet or the empty word and a symbol from the stack

alphabet, returns a finite set of pairs whose first element is a state and whose second element is a word using symbols from the stack alphabet.

Note that $P_{\text{fin}}$ simply means "finite powerset". It is defined as

$$\mathcal{P}_{\text{fin}}(A) = \{ X \mid X \subseteq A \wedge X \text{ is finite} \}$$

In other words: given a set $A$, it returns a set of all finite subsets of $A$.

5. $q_0$ is the initial state, as before.

6. $z_0$ is the initial stack symbol. It is an element of $\Gamma$ which is automatically put on the stack for us before we enter the initial state for the first time.

7. $F \subseteq Q$ is the set of final states, as before.

## 7.1 Instantaneous Descriptions

Note that we did not define an extended transition function for PDAs, so how can we work out whether a word is in the language of a pushdown automaton or not? Before we can answer this, let us define a way to represent the state of a PDA. Note that "state" does not refer to elements of $Q$ here, but instead we mean the situation a pushdown automata is in at a given point.

To avoid confusion between the state of the automaton and the states in the automaton, we will refer to the former as *instantaneous description*. These IDs or instantaneous descriptions are 3-tuples which are elements of

$$ID = Q \times \Sigma^* \times \Gamma^*$$

In other words, at any given point, a pushdown automaton will be in a state, there will be a word using symbols from the input alphabet and there will be a word consisting of symbols from the stack alphabet (the stack).

Initially, the state of the pushdown automata will be $(q_0, w, z_0)$ where $w$ is the input word. We can then derive changes to the state of the automaton by defining a relation

$$\underset{P}{\vdash} \subseteq ID \times ID$$

There are *two* cases for the relation. The first case is a "normal transition" which we define as

$$(q, xw, z\gamma) \underset{P}{\vdash} (q', w, \alpha\gamma) \textbf{ if } (q', \alpha) \in \delta(q, x, z) \tag{7.1}$$

A normal transition simply removes one symbol $x$ from the input word, moves from a state $q$ to state $q'$ and lets us modify the stack a bit.

Let us look at what's going on here in more detail: the $(q, xw, z\gamma)$ part on the far left is an instantaneous description for the state of the automaton we are currently in, where

- $q \in Q$ represents the state we are currently in
- $x \in \Sigma$ is the first symbol in the word we are currently processing
- $w \in \Sigma^*$ is the tail of the word we are currently processing
- $z \in \Gamma$ is the symbol on top of the stack
- $\gamma \in \Gamma^*$ is the rest of the stack

Now, if we pass $q \in Q$, $x \in \Sigma$ and $z \in \Gamma$ to the transition function for the pushdown automata (recall that $\delta \in Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma \to \mathcal{P}_{\text{fin}}(Q \times \Gamma^*)$), we will get a set of pairs which represent possible transitions away from the current state we are in. Each of those pairs $(q', \alpha)$ consits of a state $q' \in Q$ and a word using symbols from the stack alphabet $\alpha$. For such a pair, we can then move to a new ID $(q', w, \alpha\gamma)$.

Alternatively to this normal transition which is essentially the same as for types of automata we have encountered before, pushdown automata also support *silent transitions*. That is virtually the same as a normal transition, except that we don't remove a symbol from the input word. Formally, we define this as

$$(q, w, z\gamma) \underset{P}{\vdash} (q', w, \alpha\gamma) \textbf{ if } (q', \alpha) \in \delta(q, \varepsilon, z) \tag{7.2}$$

Note how this is the same as Equation 7.1, except that we do not take a symbol $x$ from the input word. Since we no longer have a symbol we can pass to the transition function $\delta$, we simply give it $\varepsilon$ instead.

## 7.2 Language Membership

For DFAs and NFAs, we decided if a word was in the language of the automaton by running it through the extended transition function $\hat{\delta}$ which returns a state or a set of states respectively. If the resulting state or a state in the set of states was also in the set of final states, then the word was accepted by the automaton.

For pushdown automata, there are *two* different ways of deciding whether a word is in the language of an automaton.

Firstly, we can define the PDA as described at the beginning of this chapter and use our previous approach by using instantaneous descriptions to derive where we can go from the initial state of the automaton using the input word and then test if we can end up in a final state.

Alternatively, we can define a PDA using a 6-tuple without the set of final states $F$ as follows

$$\textbf{Pushdown Automaton (Alt.):}$$
$$(Q, \Sigma, \Gamma, \delta, q_0, z_0)$$

and use a method called *acceptance by empty stack*. This means that a word will be accepted by the pushdown automaton if there is a derivation from the initial state such that the stack ends up being empty.

## 7.3 Example

As an example, let us consider the pushdown automaton from the fifth coursework. We are given the following definition for a pushdown automaton with acceptance by empty stack

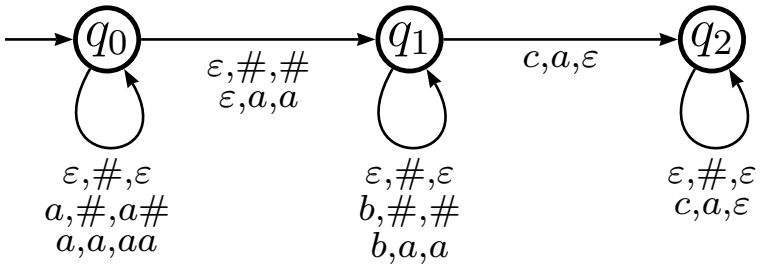
$$P = (\{\, q_0, q_1, q_2 \,\}, \{\, a, b, c \,\}, \{\, a, \# \,\}, \delta, q_0, \#)$$

where the transition function $\delta$ is defined as

$$\begin{aligned}
\delta(q_0, \varepsilon, \#) &= \{\, (q_0, \varepsilon), (q_1, \#) \,\} \\
\delta(q_0, a, \#) &= \{\, (q_0, a\#) \,\} \\
\delta(q_0, a, a) &= \{\, (q_0, aa) \,\} \\
\delta(q_0, \varepsilon, a) &= \{\, (q_1, a) \,\} \\
\delta(q_1, \varepsilon, \#) &= \{\, (q_1, \varepsilon) \,\}
\end{aligned}$$

$$\begin{aligned}
\delta(q_1, b, \#) &= \{\, (q_1, \#) \,\} \\
\delta(q_1, b, a) &= \{\, (q_1, a) \,\} \\
\delta(q_1, c, a) &= \{\, (q_2, \varepsilon) \,\} \\
\delta(q_2, \varepsilon, \#) &= \{\, (q_2, \varepsilon) \,\} \\
\delta(q_2, c, a) &= \{\, (q_2, \varepsilon) \,\} \\
\delta(_, _, _) &= \emptyset
\end{aligned}$$

Just like with the previous types of automata, we can draw a transition diagram for $P$.




Note that we represented multiple transitions from one state to another using only one arrow with multiple descriptions, but this is simply for clarity. You can draw one arrow for each transition, but it would be a lot messier than this.

For starters, let us demonstrate how we can derive that the word $ac$ is in the language $L(P)$. Recall that the state of the PDA is denoted using instantaneous descriptions consisting of a state, input word and stack. So initially we are in $q_0$, our input word is $ac$ and the stack consists of only $\#$.

$$(q_0, ac, \#)$$

Now we can use the relation $\underset{P}{\vdash}$ to derive a state we can go to from here. Note that there will often be multiple valid choices, but we will always choose the one that will eventually lead to the best result for us. Think of it as a formal proof, we are trying to prove that $ac$ is in $L(P)$, so we will obviously do what's necessary to show that.

Currently we have two choices. We could either do a silent transition to $q_1$ because $\#$ is on top of the stack at the moment or we could do a normal transition to $q_0$ because $a$ is at the beginning of the input word and $\#$ is on top of the stack. We will do the latter

$$(q_0, ac, \#) \underset{P}{\vdash} (q_0, c, a\#) \qquad 1.\ \text{with } (q_0, a\#) \in \delta(q_0, a, \#)$$

Note how we stated on the right which of the two cases for the $\underset{P}{\vdash}$ relation we used and which transition we performed using $\delta$. If you find this confusing, it helps to look back at Equation 7.1 and to bind values to the variables. We will do this in detail for the derivation we have just performed above.

The current instantaneous description is $(q_0, ac, \#)$ so that we can already bind the following values to the variables in the relation

$$\begin{aligned} q &= q_0 \\ x &= a \\ w &= c \\ z &= \# \\ \gamma &= \varepsilon \end{aligned}$$

Next, we will use $q$, $x$ and $z$ as arguments for $\delta$ which gives us

$$\delta(q_0, a, \#) = \{\, (q_0, a\#) \,\}$$

Every pair in the set returned by $\delta$ describes a state we can go to and a way of modifying the stack. In this case, there is only one possibility, so that we bind the following values to $q'$ and $\alpha$

$$\begin{aligned} q' &= q_0 \\ \alpha &= a\# \end{aligned}$$

Now we have all we need to construct a new instantaneous description

$$(q_0, c, a\#)$$

And that's all there is to it. The hard part is really just working out which of the possible transitions we should take. Recall that initially we had the choice between a normal and a silent transition, but we chose the normal transition. $\delta$ then returned only a single possibility for the normal transition so that we didn't have a choice there.

Let us continue with the derivation. The only choice we have now is to take the silent transition to $q_1$ because the next symbol in the input word is $c$ and there is no normal transition from $q_0$ for $c$, but we can ignore it and just do a silent transition because there is one for if $a$ is on top of the stack (which it is).

$$(q_0, ac, \#) \underset{P}{\vdash} (q_0, c, a\#) \qquad 1.\ \text{with } (q_0, a\#) \in \delta(q_0, a, \#)$$

$$\underset{P}{\vdash} (q_1, c, a\#) \qquad \text{2. with } (q_1, a) \in \delta(q_0, \varepsilon, a)$$

Look at Equation 7.2 if you need help, but recall that silent transitions work exactly like normal transitions (which we explained in detail above), except that we don't consume the next character in in the input word.

At this point, we once again only have one choice and that is to take a normal transition to $q_2$ because $c$ is at the beginning of the input word and $a$ is on top of the stack. If you were wondering, we cannot take the silent transition to $q_1$, because # is not on top of the stack.

$$\begin{aligned}
(q_0, ac, \#) &\underset{P}{\vdash} (q_0, c, a\#) && \text{1. with } (q_0, a\#) \in \delta(q_0, a, \#) \\
&\underset{P}{\vdash} (q_1, c, a\#) && \text{2. with } (q_1, a) \in \delta(q_0, \varepsilon, a) \\
&\underset{P}{\vdash} (q_2, \varepsilon, \#) && \text{1. with } (q_2, \varepsilon) \in \delta(q_1, c, a)
\end{aligned}$$

We have now consumed all symbols in the input word, but we are not finished yet. Recall that acceptance for $P$ is by empty stack and that we therefore don't have any final states, so we need to get rid of the # on the stack. Luckily, there is a silent transition which does exactly that

$$\begin{aligned}
(q_0, ac, \#) &\underset{P}{\vdash} (q_0, c, a\#) && \text{1. with } (q_0, a\#) \in \delta(q_0, a, \#) \\
&\underset{P}{\vdash} (q_1, c, a\#) && \text{2. with } (q_1, a) \in \delta(q_0, \varepsilon, a) \\
&\underset{P}{\vdash} (q_2, \varepsilon, \#) && \text{1. with } (q_2, \varepsilon) \in \delta(q_1, c, a) \\
&\underset{P}{\vdash} (q_2, \varepsilon, \varepsilon) && \text{2. with } (q_2, \varepsilon) \in \delta(q_2, \varepsilon, \#)
\end{aligned}$$

The stack is now empty and the word has therefore been accepted by the pushdown automaton. A word is only rejected if we get "stuck" at the end of all possible derivations from the initial state of the automaton. Here that clearly wasn't the case, because we have shown one derivation for which we can reach an empty stack.

This leads us to the question "how can we define the language of a pushdown automaton?" which we will answer in the next section.

## 7.4 The Language of a PDA

Since we have introduced two different types of pushdown automata, those that perform acceptance by final state and those that perform acceptance by empty stack, we also have two different definitions of the language of a PDA, depending on which type of PDA it is.

### 7.4.1 Acceptance by Final State

Recall that a pushdown automaton $P$ which uses acceptance by final state is defined as

$$P = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$$

Intuitively, the language of such a pushdowm automaton consists of words for which we can derive an instantaneous description whose state is a final state.

In other words, a word is accepted if there exists a derivation which results in an instantaneous description $(q', \varepsilon, \gamma)$ where we have processed all symbols in the input word and $q' \in F$. We do not care what value the stack has.

We can now formally define the language of a pushdown automaton which uses acceptance by final state as

$$L(P) = \{ w \mid (q_0, w, z_0) \overset{*}{\underset{P}{\vdash}} (q', \varepsilon, \gamma) \wedge q \in F \}$$

Note that $\overset{*}{\underset{P}{\vdash}}$ simply means "in zero or more moves" while the $\underset{P}{\vdash}$ relation represents a single "move".

### 7.4.2 Acceptance by Empty Stack

Recall that a pushdown automaton $P$ which uses acceptance by empty stack is defined as

$$P = (Q, \Sigma, \Gamma, \delta, q_0, z_0)$$

Also intuitively, the language of such a pushdowm automaton consists of words for which we can derive an instantaneous description where we have reached an empty stack after processing all symbols in the input word.

In other words, a word is accepted if there exists a derivation which results in an instantaneous description $(q', \varepsilon, \varepsilon)$ where we have processed all symbols in the input word and the stack is empty.

We can now formally define the language of a pushdown automaton which uses acceptance by empty stack as

$$L(P) = \{ w \mid (q_0, w, z_0) \overset{*}{\underset{P}{\vdash}} (q', \varepsilon, \varepsilon) \}$$

# Chapter 8

# Context-free grammars

# Chapter 9

# Further Resources

In addition to this revision guide, you are encouraged to read the lecture notes[1] as well as the recommended textbook (**?**). There is also a list of links to resources on the web which can be found at

`http://michael-gale.co.uk/teaching/g52mal`

If you find any errors in this revision guide or if you know about a useful website which might help other students as well, feel free to email the authors.

## 9.1 Turing Machines

In addition to the regular and context-free languages, you will also need to know about the *recursively-enumerable* languages and *turing machines* (or TMs) which can be used to represent these languages. These will most likely appear in at least one of the questions in the exam, so it is important that you know how they work.

Unfortunately, due to time constraints, we are unable to discuss turing machines in this revision guide. However, there are plenty of resources about TMs on the internet and in the offical lectures notes, so we recommend you have a look at those instead.

[1] `http://www.cs.nott.ac.uk/~txa/g52mal/LectureNotes/g52mal-notes.pdf`

## 9.2 Recursive Descent Parsers

Recursive descent parsers and specifically LL(1) grammars were covered in the lectures and lecture notes, but not in this revision guide. For the exam, make sure that you know what a LL(1) grammar is in case the topic shows up.