THE UNIVERSITY OF CAMBRIDGE
PART II AUTOMATA AND FORMAL LANGUAGES (C)
24 LECTURES, MICHAELMAS 2018–19

Prepared by Maurice Chiodo

Contents

0. INTRODUCTION

These notes have been prepared for students studying the Part II (C) course *Automata and Formal Languages*. They serve as the *backbone* of the course. As such, they contain all the essential definitions, theorems, and proofs. However, they do **not** form the body of the course; that is best obtained from attending the lectures. In particular, there are very few examples in these notes. The reason that they have been omitted is because examples should be built and worked through in front of the reader for maximum effect. Merely seeing the final product does not impart the same intuition and insight as seeing the example being worked through in real time. With this in mind, students are encouraged to keep a copy of the notes on-hand to ensure that they have transcribed definitions and theorems correctly, but *not* to learn from. These notes are a memory aid, not a learning tool. Treat them as such.

These notes contain material on all three sections of the course. To remove any ambiguity, the course content (i.e., examinable material) will be hereby defined as the material that is *lectured.*

For those of you who are reading this in digital format, you can click on items in the table of contents to go directly to that part in the text. Also, when you click on a definition number, you will go directly to the statement. The same applies for lemmata, theorems, etc.

For those of you reading this in printed format, yet yearning to enter the digital age, the pdf notes reside in an encrypted container called `Automata2018.tc`, which can be found by following the teaching link from my homepage at

`https://www.dpmms.cam.ac.uk/~mcc56/`

To open the container, you'll need to use the cryptographic software *TrueCrypt* or *VeraCrypt* (the latter might be better for Mac users; if you do use VeraCrypt, then you need to select 'TrueCrypt mode' when decrypting). Downloads for most operating systems, and instructions, can be found at

`https://www.grc.com/misc/truecrypt/truecrypt.htm`
`https://www.veracrypt.fr/en/Home.html`

You will also need the following password[1] to open the container:

`HandTakeAlsoFreight2018CableNorthCloud`

[1]Or a lot of spare CPU power.

0.1. **List of reference texts.**

Each of the three sections of this course was designed from a different reference text[2]. These are:

(1) Register machines and computability theory:
P.T. Johnstone, *Notes on logic and set theory* (Chapter 4), Cambridge University Press, 1987.

(2) Regular languages and finite-state automata:
J.E. Hopcroft, R. Motwani and J.D. Ullman, *Introduction to automata theory, languages and computation* (Chapters 2–4), 2nd edn, Addison-Wesley, 2001. (Note that this is **not** the edition stated in the course description).

(3) Pushdown automata and context-free languages:
D.C. Kozen, *Automata and computability* (Lectures 19–25), Springer, 1997.

If you insist on obtaining only one text for the course then the book by Kozen will serve you best, but by no means fully.

0.2. **Shorthand conventions.**

Throughout the lectures I will make use of certain shorthand conventions. Some of these are standard, others are not. For clarity, here is a list of those which I will be using.

b/c = because
c/f = comes from
w/ = with
w/o = without
wts = want to show
wlog = without loss of generality
thm = theorem
defn = definition
lem = lemma
cor = corollary
pf = proof
eg = example
ex = exercise
RM = register machine
PC = partial computable
PR = partial recursive
Prim R = primitive recursive
s.t. = such that
rec = recursive
iff = if and only if

[2]Because life is never as straightforward as it should be.

1. REGISTER MACHINES AND COMPUTABILITY THEORY

To explore what is *incomputable*, we first need a robust definition of what it means for a problem to be *computable*. There are many equivalent ways to do this; we present one of them here.

1.1. **Register machines and computable functions.**

Definition 1.1 (Register machines)**.**
A *register machine* consists of two parts: a sequence of *registers* $R_1, R_2, \ldots$, and a finite *program.*

A *register* is a place to hold, at any time, an arbitrary natural number (including 0). Think of these as 'buckets', each holding a single integer.

A *program* (often denoted P) is defined by specifying a finite number of *states* $S_0, S_1, \ldots, S_n$, and, for (not necessarily all) $i \in \{0 \ldots, n\}$, an instruction to be carried out when the machine is in state i. These instructions are of two types:

(1) add 1 to register R_j and move to state S_k (written $S_i : (j, +, k)$);
(2) test whether R_j holds the integer 0: if it does, move to state S_l; otherwise, subtract 1 from it and move to state S_k (written $S_i : (j, -, k, l)$).

The *input* of a register machine is a finite (ordered) set of registers $(R_1, \ldots, R_n)$, each containing a non-negative integer, and by convention we set all other registers $\{R_j\}_{j>n}$ to contain 0.

S_1 is the *initial state*, and it is from this state that we begin applying the program to the registers. S_0 is the *terminal state*; upon reaching it, the machine ceases to operate, so there is no need to have an instruction associated to S_0.

The machine is permitted to move to a (non-terminal) state with no instruction associated to it (written $S_i : \emptyset$). If this occurs, the machine simply sits in limbo forever; neither terminating, nor performing further computation.

Of course, a register machine can only ever change the entries in finitely many of the registers, as there are only finitely many states in a program, and each state can only modify at most one specific register. We continue to use an infinite sequence of registers $R_1, R_2, \ldots$ to save having to specify how many are needed.

We can describe a register machine in many ways. Two such ways, which we give here, are via a *sequence of instructions* or a *program diagram.*

Definition 1.2 (Sequence of instructions)**.**
A *sequence of instructions* for a register machine with program P is simply the collection of instructions of P, written $S_i : (j, +, k)$ or $S_i : (j, -, k, l)$. By writing out the triples $(j, +, k)$ and quadruples $(j, -, k, l)$ in an ordered list, is it assumed that S_1 corresponds to the first instruction in the list, S_2 to the second, and so on, thus we usually omit the prefix S_i on each instruction. This completely describes the register machine with program P.

Definition 1.3 (Program diagrams)**.**
A *program diagram* for a register machine with program P is a graph Γ with directed edges (some of which are labelled) and labelled vertices. The vertex set of Γ consists of the states of P. We then generate edges as follows: For each instruction of type (1) (when in state S_i, add 1 to register R_j and move to state S_k), we include a directed edge from S_i to S_k, with label $R_j + 1$. That is,

$$S_i \xrightarrow{R_j+1} S_k$$

For each instruction of type (2) (when in state S_i, test whether R_j holds the integer 0: if it does, move to state S_l; otherwise, subtract 1 from it and move to state S_k), we include two edges. One is a directed edge from S_i to S_k with label $R_j - 1$. The other is an unlabelled directed edge (often written as a dotted edge) from S_i to S_l. That is,

$$\begin{tikzcd} S_l & S_i \arrow[l, dashed] \arrow[r, "R_j-1"] & S_k \end{tikzcd}$$

This completely describes the register machine with program P.

Notice that we have not specified the registers at all in the above two definitions. This is because they are not needed in order to define the register machine. We give the machine an input (a finite sequence of 'filled' registers), and the rest are set to 0 by default. However, the machine itself does not care which registers we pre-set to non-0 entries. The machine could, in fact, commence will all registers set to 0 (so with no external input).

Each of the above two definitions can be used to completely describe a register machine. A program diagram is more intuitive, but harder to write down. A sequence of instructions is easy to write down, but difficult to follow. From one such description, we can always convert to the other.

We have specified an initial state S_1 and a final state S_0 in our register machines. However, it might be the case that the machine never reaches the final state S_0; an easy example of this is a program with two states S_0, S_1, and the instruction for S_1 is 'add 1 to register R_1, then move into state S_1 again' (i.e., with sequence of instructions $S_1 : (1, +, 1)$). This machine just keeps adding 1 to register R_1, and never reaches state S_0. The key idea here is that we care about the inputs on which the register machine eventually reaches S_0.

Definition 1.4 (Halting sets).
A register machine with program P is said to *halt* on input $(m_1, \dots, m_k) \in \mathbb{N}^k$ if, when given an input of registers $(R_1, \dots, R_k)$ with each R_i holding integer m_i (and all others holding 0), the machine eventually reaches state S_0 after application of finitely many instructions. We write this as $P(m_1, \dots, m_k) \downarrow$. The *halting set* of P, written $\Omega(P)$, is the set of inputs on which P halts. That is,

$$\Omega(P) := \bigcup_{k>0} \{(m_1, \dots, m_k) \in \mathbb{N}^k \mid P(m_1, \dots, m_k) \downarrow\}$$

If P does not halt on input $(m_1, \dots, m_k)$, then we write $P(m_1, \dots, m_k) \uparrow$.

Definition 1.5 (Upper register of a program).
There will be some finite k, for each program P, for which all registers after the k^{th} are 'ignored' (or 'not touched') by P. That is, if we set

$$\text{upper}(P) := \max\{i \in \mathbb{N} \mid (i, +, j) \text{ or } (i, -, j, l) \text{ is in } P\}$$

then no register R_j with index $j > \text{upper}(P)$ will ever be modified by P. We call this index upper(P), the *upper register index* of P.

So each program P has a maximum number of input registers it can process. Of course, we want our machines to actually take an input *and* give an output; that is, we want them to compute something.

Definition 1.6 (Partial computable functions).
A partial function $f : \mathbb{N}^k \to \mathbb{N}$ is said to be *partial computable* by a program

P if, for all $(m_1, \dots, m_k) \in \mathbb{N}^k$ where $f(m_1, \dots, m_k)$ is defined, we have that $P(m_1, \dots, m_k) \downarrow$ with $f(m_1, \dots, m_k)$ in register R_1 when it halts.

We allow our definition to include *partial functions* (those defined on a subset of $\mathbb{N}^k$). When this happens, we require that $P(m_1, \dots, m_k) \uparrow$ for the inputs on which $f(m_1, \dots, m_k)$ is undefined. Thus, every program P for a register machine defines an n-variable partial function for each $n > 0$ (though different programs can define the same function).

We now introduce some basic techniques for register machine operations. We will use these several times in later proofs, so it helps to establish their existence now.

Lemma 1.7 (Addition of registers). *We can write a program P (or a subroutine of a program) to add the contents of R_i to R_j, leaving R_i unchanged at the end.*

Proof. A program that does this is given by the sequence of instructions:
$S_1 : (i, -, 2, 4)$
$S_2 : (n, +, 3)$ where n is larger than both i and j
$S_3 : (j, +, 1)$
$S_4 : (n, -, 5, 0)$
$S_5 : (i, +, 4)$

If instead we were wanting to add the above subroutine to a program P, we would need to choose n to be larger than both the upper register index for P, and the largest input register for P. □

Note that, in the above proof, we have specified which instruction corresponds to which state. Usually we just write out the instructions in a list, and take the i^{th} instruction in the list to correspond to state S_i.

The process described in Lemma 1.7 would be much easier if we just wanted to *transfer* the contents of R_i to R_j (that is, emptying R_i in the process). We leave this as an exercise.

Lemma 1.8 (Emptying registers). *We can write a program P (or a subroutine of a program) to empty the register R_i.*

Proof. A program that does this is given by the sequence of instructions:
$S_1 : (i, -, 1, 0)$. □

1.2. **Partial recursive functions.**

We can now give a large class of functions which are partial computable.

Theorem 1.9 (Closure properties of partial computable functions).
a) (Basic functions) *For each $i \le k$, the* projection function $(n_1, \dots, n_k) \mapsto n_i$ *is partial computable.*
b) (Basic functions) *The* constant function *with value* 0 *(that is, $n \mapsto 0$), and the* successor function $n \mapsto n + 1$, *are partial computable.*
c) (Composition) *If f is a partial computable function on k variables, and $g_1, \dots, g_k$ are partial computable functions each on l variables, then the function h on l variables given by*

$$h(n_1, \dots, n_l) := f(g_1(n_1, \dots, n_l), \dots, g_k(n_1, \dots, n_l))$$

is also partial computable. Note that we take h as being defined on $(n_1, \dots, n_l)$ when each g_i is defined on $(n_1, \dots, n_l)$ and *f is defined on $(g_1(n_1, \dots, n_l), \dots, g_k(n_1, \dots, n_l))$.*

d) (Recursion) *If f and g are partial computable functions on k and $k+2$ variables respectively, then the following function h on $k+1$ variables defined inductively by*

$$h(n_1,\ldots,n_k,0) := f(n_1,\ldots,n_k)$$
$$h(n_1,\ldots,n_k,n_{k+1}+1) := g(n_1,\ldots,n_k,n_{k+1},h(n_1,\ldots,n_k,n_{k+1}))$$

is partial computable (and defined if and only if f and g are appropriately defined).

e) (Minimalisation) *If f is a partial computable function on $k+1$ variables, then the following partial function g on k variables defined by*

$$g(n_1,\ldots,n_k) := \begin{cases} n & \text{if } f(n_1,\ldots,n_k,n)=0 \text{ and } f(n_1,\ldots,n_k,m)>0\ \forall m<n \\ \text{undefined} & \text{otherwise} \end{cases}$$

is partial computable.

Parts a) and b) of this theorem say that the *basic functions* are partial computable, part c) says that partial computable functions are closed under *composition*, part d) says that partial computable functions are closed under *recursion*, and part e) says that partial computable functions are closed under *minimalisation.*

Our proof will make use of the subroutines given in Lemmata 1.7 and 1.8, so we won't write them out again explicitly.

Proof.

a) For $i=1$, the projection function can be computed by a program which does nothing to register R_1, and enters the halt state immediately. For example, the 1-line program $(2,+,0)$ will suffice. For $i>1$, we take the program which first empties R_1, then transfers R_i to R_1, then halts. For example, the following program will suffice: $(1,-,1,2),(i,-,3,0),(1,+,2)$.

b) These two functions can be computed by the one-line programs $(1,-,1,0)$ and $(1,+,0)$ respectively.

c) We construct a program to compute h as follows. First, let M_1 be the supremum of the upper register indices of the g_j's and of f. Now set $M = M_1+k+l$ (so none of the registers after R_M will ever be non-zero). Let $n=(k+1)M$, and now transfer the contents of $R_1,\ldots,R_l$ to $R_{n+1},\ldots,R_{n+l}$ respectively, setting each $R_1,\ldots,R_l$ to 0 in the process; this is to 'store them safely' so that they are not affected by the rest of our computation. Now, for each $1\le i\le k$, we copy (without deletion) the registers $R_{n+1},R_{n+2}\ldots,R_{n+l}$ to $R_{iM+1},R_{iM+2},\ldots,R_{iM+l}$, then perform the computation of g_i with all registers shifted iM places to the right, then store the output in R_i (Observe that M was chosen sufficiently large so that at no time in the computation will we reach, or pass, register $R_{(i+1)M}$. Moreover, the computation will not touch any register below R_{iM+1}, as the original computation never tries to touch a register below R_1). Note that, starting with registers containing $(n_1,\ldots,n_l,0,\ldots)$, we now have registers with contents

$$(g_1(n_1,\ldots,n_l),\ldots,g_k(n_1,\ldots,n_l),\overbrace{0,\ldots,0}^{M-k \text{ 0's}},\ \text{other entries }\,)$$

Finally, perform the computation for f. Note that upper$(f)\le M_1$, and both $M_1,l\le M$. Thus the entries beyond register R_M will be irrelevant to the computation of f. So, after computing f we will be left with $f(g_1(n_1,\ldots,n_l),\ldots,g_k(n_1,\ldots,n_l))$ in R_1, as required.

d) This is similar to the previous case. We first copy (without deletion) the entries in registers $R_1, \dots, R_{k+1}$ to $R_{n+1}, \dots, R_{n+k+1}$, with n chosen to 'store these entries safely' as follows: let M_1 be the supremum of the upper register indices of f and g, and set $n = M_1 + k + 2$. Now, set register R_{k+1} to 0, run the computation for f, take the output (the contents of R_1) and copy it to register R_{n+k+2}. (*) Subtract 1 from register R_{n+k+1} (which counts the number of steps of the recursion remaining). Now set registers $R_1, \dots, R_n$ to 0, then copy $R_{n+1}, \dots, R_{n+k}$ to $R_1, \dots, R_k$, copy R_{n+k+3} (which counts the number of steps j done so far in the recursion) to R_{k+1}, and copy R_{n+k+2} (the current value $h(n_1, \dots, n_k, j)$) to R_{k+2}. Now run g, to end up with the value $g(n_1, \dots, n_k, j, h(n_1, \dots, n_k, j)) = h(n_1, \dots, n_k, j+1)$ in R_1. Empty R_{n+k+2} and replace it with the contents of R_1, and add one to register R_{n+k+3}. Then go back to (*) and repeat. If, however, R_{n+k+1} was already 0 when we tried to subtract 1 from it, then instead just enter state S_0; R_1 will contain $h(n_1, \dots, n_{k+1})$ at this point.

e) This is similar to the previous case. We first copy the entries in registers $R_1, \dots, R_k$ to $R_{n+1}, \dots, R_{n+k}$, with n chosen to 'store these entries safely' as follows: set $n = \text{upper}(f)+k+1$. Now enter a subroutine where we empty the registers $R_1, \dots, R_{k+1}$ and then replace them with the contents of $R_{n+1}, \dots, R_{n+k+1}$ respectively, and then perform the computation of f. At the end of this subroutine, we obey $(1, -, j, j')$; from S_j we add 1 to R_{n+k+1} and return to the beginning of the subroutine; from $S_{j'}$ we empty R_1 and then copy R_{n+k+1} to R_1, and then halt. □

In fact, by using the above functions, we can define an interesting class:

Definition 1.10 (Partial recursive functions)**.**
We define the class of *partial recursive functions* as the smallest class of partial functions from $\mathbb{N}^n \to \mathbb{N}$ (for all n) which is closed under the properties of Theorem 1.9. That is, such a function f can be constructed from the basic functions and applications of composition, recursion, and minimalisation a finite number of times. If f can be constructed without minimalisation, then we say that it is *primitive recursive.*

Lemma 1.11. *Every primitive recursive function $f : \mathbb{N}^n \to \mathbb{N}$ is* total *(that is, defined on all of $\mathbb{N}^n$).*

Proof. The projection, constant, and successor functions are obviously total. The composition of total functions is again total. Finally, performing primitive recursion on total functions is again total. □

We point out (without proof) that not every total recursive function is primitive recursive; the *Ackermann function*[3] is one such example.

Example 1.12. *Addition and multiplication of integers are both primitive recursive functions.*

Proof. For addition, we can use the recursive definition

$$\begin{aligned} h(n_1, 0) &:= n_1 \\ h(n_1, n_2 + 1) &:= g(n_1, n_2, h(n_1, n_2)) = h(n_1, n_2) + 1 \end{aligned}$$

[3]A very interesting function, which we do not have time to cover.

where $g(n_1, n_2, h(n_1, n_2)) := h(n_1, n_2) + 1$ is projection onto the 3rd factor, followed by successor. Thus we see that h is primitive recursive, and $h(n_1, n_2) = n_1 + n_2$.

For multiplication, we can use the recursive definition

$$h(n_1, 0) := 0$$
$$h(n_1, n_2 + 1) := g(n_1, n_2, h(n_1, n_2)) = h(n_1, n_2) + n_1$$

where $g(n_1, n_2, h(n_1, n_2)) := h(n_1, n_2) + n_1$ is projection onto the 3rd factor, followed by addition of n_1. Thus we see that h is primitive recursive, and $h(n_1, n_2) = n_1 * n_2$. □

Example 1.13. $(n_1, n_2) \mapsto n_1^{n_2}$ *is primitive recursive.*

Proof. We can use the recursive definition

$$h(n_1, 0) := 1$$
$$h(n_1, n_2 + 1) := g(n_1, n_2, h(n_1, n_2)) = h(n_1, n_2) * n_1$$

where $g(n_1, n_2, h(n_1, n_2)) := h(n_1, n_2) * n_1$ is projection onto the 3rd factor, followed by multiplication by n_1. Thus we see that h is primitive recursive, and $h(n_1, n_2) = n_1^{n_2}$. □

Theorem 1.9 shows that all partial recursive functions are partial computable. We will soon show the converse. First, we need to introduce the following notation.

Definition 1.14. Let $n > 0$ and $i \geq 0$. We write p_i for the $(i+1)^{\text{th}}$ prime (so $p_0 = 2$), and we write $(n)_i$ for the largest power of p_i which divides n.

Lemma 1.15. *The function* $(\cdot)_i : \mathbb{N} \to \mathbb{N}$ *given above, sending* $n \mapsto (n)_i$ *if* $n > 0$ *(and* $0 \mapsto 0$*), is primitive recursive, for each fixed* $i \geq 0$*.*

The proof of this result involves showing that several intermediate functions are primitive recursive. Many of these are useful in their own right, and some will be used explicitly in later proofs.

Proof. We build this up in stages. In each stage, we define a family of functions over all $k > 1$ (note that k is an *index* of these functions, not a variable within the functions). We show that each individual function in each family is primitive recursive.

(1) Step functions:

$$\text{step}_k(n) = \begin{cases} 1 & \text{if } 0 \leq n \leq k-2 \\ 0 & \text{if } n > k-2 \end{cases}$$

We prove this inductively on k. First, we show step_2 is primitive recursive. This follows from the fact that we can define step_2 via recursion by

$$\text{step}_2(0) := 1$$
$$\text{step}_2(n+1) := 0$$

Now assume step_j is primitive recursive for all $j < k$. We can define step_k via recursion by:

$$\text{step}_k(0) := 1$$
$$\text{step}_k(n+1) := \text{step}_{k-1}(n)$$

(2) Delta functions (also defined for $k = 0, 1$):

$$\delta_k(n) = \begin{cases} 1 & \text{if } n = k \\ 0 & \text{if } n \neq k \end{cases}$$

We can define $\delta_k(n)$ via composition and product, using step_k, by:

$$\delta_k(n) := \mathrm{step}_{k+2}(n) \cdot \mathrm{step}_2(\mathrm{step}_{k+2}(n+1))$$

(3) Truncated successor functions:

$$\mathrm{slope}_k(n) = \begin{cases} n+1 & \text{if } 0 \leq n \leq k-2 \\ 0 & \text{if } n > k-2 \end{cases}$$

We can define slope_k via recursion, using step_k, by:

$$\mathrm{slope}_k(0) := 1$$
$$\mathrm{slope}_k(n+1) := \mathrm{step}_k(n+1) \cdot (\mathrm{slope}_k(n) + 1)$$

(4) Remainder functions:

$$\mathrm{rem}_k(n) = n \mod k$$

We can define rem_k via recursion, using slope_k, by:

$$\mathrm{rem}_k(0) := 0$$
$$\mathrm{rem}_k(n+1) := \mathrm{slope}_k(\mathrm{rem}_k(n))$$

(5) Floor functions:

$$\mathrm{floor}_k(n) = \left\lfloor \frac{n}{k} \right\rfloor$$

We can define $\mathrm{floor}_k(n)$ via recursion, using δ_0 and rem_k, by:

$$\mathrm{floor}_k(0) := 0$$
$$\mathrm{floor}_k(n+1) := \mathrm{floor}_k(n) + \delta_0(\mathrm{rem}_k(n+1))$$

(6) Division functions:

$$\mathrm{divide}_k(n) = \begin{cases} \frac{n}{k} & \text{if } n \equiv 0 \mod k \\ 0 & \text{otherwise} \end{cases}$$

We can define $\mathrm{divide}_k(n)$ via composition and product, using δ_0, rem_k and floor_k, by:

$$\mathrm{divide}_k(n) := \mathrm{floor}_k(n) \cdot \delta_0(\mathrm{rem}_k(n))$$

(7) Division by powers:

$$\mathrm{power}_k(n, m) = \begin{cases} \frac{n}{k^m} & \text{if } n \equiv 0 \mod k^m \\ 0 & \text{otherwise} \end{cases}$$

We can define $\mathrm{power}_k(n, m)$ via recursion, using divide_k, by:

$$\mathrm{power}_k(n, 0) := n$$
$$\mathrm{power}_k(n, m+1) := \mathrm{divide}_k(\mathrm{power}_k(n, m))$$

(8) Maximum powers dividing an integer:

$$\mathrm{maxpow}_k(n) = \begin{cases} \text{the largest power of } k \text{ dividing n, if } n \neq 0 \\ 0 \quad \text{if } n = 0 \end{cases}$$

We can define maxpow_k using δ_0 and power_k. First we define an auxiliary function h by recursion via:

$$h(n,0) := 0$$
$$h(n,m+1) := h(n,m) + \delta_0(\delta_0(\text{power}_k(n,m+1)))$$

Now we define maxpow_k via composition by:

$$\text{maxpow}_k(n) := h(n,n)$$

Observe that $h(n,n) = 0 + \delta_0(\delta_0(\text{power}_k(n,1))) + \delta_0(\delta_0(\text{power}_k(n,2))) + \ldots + \delta_0(\delta_0(\text{power}_k(n,n)))$, and that $k^n > n$ (as $k > 1$). Also observe that

$$\delta_0(\delta_0(\text{power}_k(n,j))) = \begin{cases} 1 & \text{if } k^j \text{ divides } n \text{ and } n > 0 \\ 0 & \text{otherwise} \end{cases}$$

Thus, for $n > 0$, we have $h(n,n) = \Sigma_{j=1}^{m} 1 = m$ if $m > 0$ (or 0 if $m = 0$), where m is the largest power of k dividing n. Moreover, $h(0,0) = 0$.

Finally, we take $(n)_i := \text{maxpow}_{p_i}(n)$, which is primitive recursive. □

1.3. **Equivalence of partial recursive and partial computable functions.**

We showed in Theorem 1.9 that partial recursive functions are partial computable. Now we show that partial computable functions are partial recursive.

Theorem 1.16. *Every partial computable function is partial recursive.*

In this proof we make use of the functions defined in the proof of Lemma 1.15.

Proof. Let $f : \mathbb{N}^k \to \mathbb{N}$ be a partial computable function on k variables, with program P. We define an auxiliary function $g : \mathbb{N}^{k+2} \to \mathbb{N}$ as follows:

• $g(n_1, \ldots, n_k, 0, t)$ is the number of the state of P reached after t computational steps, starting at state S_1 with input $(n_1, \ldots, n_k, 0, \ldots)$. If P halts (i.e., reaches S_0) in fewer than t steps on this input, then we take this to be 0.

• $g(n_1, \ldots, n_k, i, t)$ is the contents of register R_i after P has run t computational steps, starting at state S_1 with input $(n_1, \ldots, n_k, 0, \ldots)$. If P halts in fewer than t steps on this input, then we take the contents of R_i when P halted.

Clearly g is total; we now show that it is actually primitive recursive. Set $r := \text{upper}(P) + k + 1$. Then $g(n_1, \ldots, n_k, i, t)$ can only be non-zero if $0 \le i \le r$. So, for each fixed $n_1, \ldots, n_k, t$, we can express the values of $g(n_1, \ldots, n_k, i, t)$ for $0 \le i \le r$ by the finite sequence $(g_0, \ldots, g_r)$ (where each g_i depends on $(n_1, \ldots, n_k, t)$), and then we can code this to the integer $2^{g_0} 3^{g_1} \cdots p_r^{g_r}$. Then this coding function $c : (g_0, \ldots, g_r) \mapsto 2^{g_0} 3^{g_1} \cdots p_r^{g_r}$ is primitive recursive (by Examples 1.12 and 1.13). Note that the function $(\cdot)_i$ does the following to integers of the form $2^{g_0} 3^{g_1} \cdots p_r^{g_r}$:

$$(2^{g_0} 3^{g_1} \cdots p_r^{g_r})_i = \begin{cases} g_i \text{ if } 0 \le i \le r \\ 0 \text{ if } i > r \end{cases}$$

So $(\cdot)_i$ is the component-wise inverse to c. That is, $(c(g_0, \ldots, g_r))_i = g_i$ is projection onto the $i+1$ entry. Moreover, $(\cdot)_i$ is primitive recursive, as shown in Lemma 1.15. We now proceed to define g via primitive recursion on the last variable. First, we define a function $h : \mathbb{N}^{k+2} \to \mathbb{N}$ via recursion, starting with

$$h(n_0, \ldots, n_k, 0) = 2^{n_0} 3^{n_1} \cdots p_k^{n_k}$$

So when given the index n_0 of the initial state of P (which will be 1), as well as the contents $(n_1, \ldots, n_k)$ of the first k registers, $h(n_0, \ldots, n_k, 0)$ is the integer which codes these (and is primitive recursive, just like c). Now we need a 'transition function' for these coded integers to simulate the steps of P, which will give us our recursive definition of h. This will be a function which 'computes one step of P', and gives us the code for all the new register values and the new state. We call this function $s : \mathbb{N} \to \mathbb{N}$, with $s(2^{n_0}3^{n_1} \cdots p_r^{n_r} p_{r+1}^{n_{r+1}} \cdots p_m^{n_m}) := 2^{n'_0}3^{n'_1} \cdots p_r^{n'_r} p_{r+1}^{n_{r+1}} \cdots p_m^{n_m}$ (so n_l is unchanged for all $l > r$), and where

$$n'_0 := \begin{cases} \beta & \text{if } S_{n_0} \Rightarrow (j, +, \beta) \\ \beta & \text{if } S_{n_0} \Rightarrow (j, -, \beta, \gamma) \text{ and } n_j > 0 \\ \gamma & \text{if } S_{n_0} \Rightarrow (j, -, \beta, \gamma) \text{ and } n_j = 0 \\ 0 & \text{if } n_0 = 0 \\ n'_0 & \text{if } S_{n_0} \Rightarrow \emptyset \text{ (no instruction for } S_{n_0}) \end{cases}$$

(where 'if $S_a \Rightarrow (j, +, \beta)$' means 'if the instruction for state S_a in P is $(j, +, \beta)$', and so on). And, for $1 \leq j \leq r$, we have

$$n'_j := \begin{cases} n_j & \text{if } S_{n_0} \Rightarrow (l, +, \beta) \text{ or } (l, -, \beta, \gamma), \text{ where } l \neq j \\ n_j + 1 & \text{if } S_{n_0} \Rightarrow (j, +, \beta) \\ n_j - 1 & \text{if } S_{n_0} \Rightarrow (j, -, \beta, \gamma) \text{ and } n_j > 0 \\ 0 & \text{if } S_{n_0} \Rightarrow (j, -, \beta, \gamma) \text{ and } n_j = 0 \\ n_j & \text{if } S_{n_0} \Rightarrow \emptyset \text{ (no instruction for } S_{n_0}) \end{cases}$$

Starting with $\alpha := 2^{n_0}3^{n_1} \cdots p_m^{n_m}$ we first observe that we can compute, in a primitive recursive way, $p_{r+1}^{n_{r+1}} \cdots p_m^{n_m}$ as we have

$$p_{r+1}^{n_{r+1}} \cdots p_m^{n_m} = \text{power}_{p_r}(\ldots \text{power}_{p_1}(\ \text{power}_{p_0}(\alpha, (\alpha)_0),\ (\alpha)_1\ ) \ldots (\alpha)_r)$$

Computing n_j for $0 \leq j \leq r$ is primitive recursive, as it is just the function $(\cdot)_j$. Moreover, computing n'_j (for $0 \leq j \leq r$) is also primitive recursive, because we have a finite number of 'non-trivial exceptions' in the definitions of n'_j so we can use functions like δ_k and step_k to compute these. So we can compute $n'_0, \ldots, n'_r$, and the product $p_{r+1}^{n_{r+1}} \cdots p_m^{n_m}$. Thus, as c is primitive recursive, we see that $s(\alpha) = 2^{n'_0}3^{n'_1} \cdots p_r^{n'_r} p_{r+1}^{n_{r+1}} \cdots p_m^{n_m} = c(n'_0, \ldots, n'_r) \cdot \text{power}_{p_r}(\ldots(\ \text{power}_{p_0}(\alpha, (\alpha)_0), \ldots (\alpha)_r)$ is also a primitive recursive function. Finally, we finish the recursive definition of h with

$$h(n_0, \ldots, n_k, t + 1) := s(h(n_0, \ldots, n_k, t))$$

So $h(1, n_1, \ldots, n_k, t)$ is the coded integer giving the state and registers of program P, on input $(n_1, \ldots, n_k)$, after t computational steps. So we see that $g : \mathbb{N}^{k+2} \to \mathbb{N}$ is primitive recursive, as

$$g(n_1, \ldots, n_k, i, t) = \big(h(1, n_1, \ldots, n_k, t)\big)_i \ = \sum_{j=0}^{r} \delta_j(i) \cdot \big(h(1, n_1, \ldots, n_k, t)\big)_j$$

Now let $q(n_1, \ldots, n_k)$ be the smallest t such that $g(n_1, \ldots, n_k, 0, t) = 0$, if such a t exists (interpret this as the number of steps that P takes to reach the halting state). Thus q is defined via minimalisation (on the primitive recursive function g), and is therefore partial recursive. Then we see that the partial computable function f is given by

$$f(n_1, \ldots, n_k) = g(n_1, \ldots, n_k, 1, q(n_1, \ldots, n_k))$$

which is partial recursive (but not necessarily primitive recursive). $\square$

From hereon, we will interchangeably refer to functions as 'partial recursive' or 'partial computable'; there is no difference as the two classes of functions are the same.

1.4. **Algorithms and encodings.**

We now have a large class of functions, partial computable functions, definable both 'mechanically' and algebraically. It turns out that many of the functions we come across in mathematics are partial computable, or even total computable. These include:

- Arithmetic functions (addition, subtraction, multiplication, division with remainder).
- Computing reductions mod n.
- Primality testing.
- Computing gcd and lcm.

These functions are, in a sense, the 'nicest' functions, from a computational standpoint. They are the ones that we can simulate via our computational device (register machines).

Definition 1.17 (Recursive functions and recursive sets)**.**
A function $f : \mathbb{N}^k \to \mathbb{N}$ is said to be *recursive*, or *computable*, if it is total recursive. A set $X \subseteq \mathbb{N}$ is said to be *recursive*, or *computable*, or *decidable*[4], if its characteristic function

$$\mathcal{X}_X(n) := \begin{cases} 1 \text{ if } n \in X \\ 0 \text{ if } n \notin X \end{cases}$$

is a total recursive (= computable) function. This extends to subsets of $\mathbb{N}^k$.

Thus a function f is recursive (= computable) if we can always compute it, and a set X is recursive (= computable, decidable) if we can always compute whether or not a given integer (or tuple) lies in X. So such functions, and such sets, can be completely understood and 'computed' with register machines.

We will now flip this idea, and say that register machines are *the* way to do computation.

Definition 1.18 (Algorithms)**.**
An *algorithm* is any process which takes as input some recursive subset of $\mathbb{N}^k$, and which can be simulated by a register machine. A *total algorithm* is one which will always terminate on every element in its input set. A *partial algorithm* is one which may fail to terminate on some elements in its input set.

It is important that the input set of the algorithm is recursive, so that we can pre-test inputs to check that the algorithm can process them[5]. Usually (but not always), the algorithms we will describe will have $\mathbb{N}^k$ as input set. Just as it is nonsensical to input the pair $(1, 2)$ into an algorithm which takes as input a single integer, we must also be careful not to 'break' our algorithms in other ways. For example, if we have an algorithm which takes as input a square number n, and outputs the square root of n, then we cannot input 5 into this algorithm. However, the set of square numbers is recursive. In general, we need to ensure that our input is suitable for the algorithm to start 'working on'.

[4]Most of the time we are interested in *deciding* if an integer has a certain property or not, hence the alternate name *decidable*.

[5]A blender is an excellent machine for mincing food, but you wouldn't want to put a brick in it.

It turns out that not all functions are computable. Moreover, not all functions are partial computable; we call such functions *incomputable.*

Lemma 1.19 (Incomputable functions)**.**
There exists an incomputable function from $\mathbb{N}^k \to \mathbb{N}$, for each $k \geq 1$.

Proof. Each partial computable function comes from a finite program of a register machine (alternatively, from a finite number of applications of composition, recursion and minimalisation with the finite set of basic functions). Thus there are at most countably many partial computable functions, yet there are uncountably many functions from $\mathbb{N}^k \to \mathbb{N}$ for any $k \geq 1$. □

Observe that the above proof actually shows that there are *uncountably many* incomputable functions from $\mathbb{N}^k \to \mathbb{N}$, for each $k \geq 1$.

One problem with our definition of an algorithm is that it *only* takes in to account computations from $\mathbb{N}^k$ to $\mathbb{N}$. Thus, strictly speaking, we can't consider the process 'take a word in the English language, and compute the number of letters in it' as a computable function; the input is not a k-tuple of integers. Similarly for the process 'take an integer n, and compute the first word in the English dictionary with n letters', as the output is not an integer. As we will soon see, we need ways to *encode* our inputs as k-tuples, and our outputs as integers. We start with ways of encoding *tuples* as integers.

It helps to have a notion of how to produce an ordered list of the elements of $\mathbb{N}^m$. There are many ways to do this; one such way is called the *shortlex ordering.*

Definition 1.20 (Shortlex ordering)**.**
We define the *shortlex* ordering on $\mathbb{N}^m$ as follows: $(n_1, \ldots, n_m) < (n'_1, \ldots, n'_m)$ if $\Sigma_{i=1}^m n_i < \Sigma_{i=1}^m n'_i$ or $\Sigma_{i=1}^m n_i = \Sigma_{i=1}^m n'_i$ and for some k we have $n_i = n'_i$ for all $1 \leq i \leq k$ but $n_{k+1} < n'_{k+1}$.

We can use shortlex to produce an 'indexed list' of $\mathbb{N}^m$: Take all elements $(n_1, \ldots, n_m)$ with sum of entries $\Sigma_{i=1}^m n_i = 0$, and order these by shortlex. Then take all elements $(n_1, \ldots, n_m)$ with sum of entries $\Sigma_{i=1}^m n_i = 1$, and order these by shortlex. And so on. Thus, for each $n \in \mathbb{N}$, we can construct the n^{th} element of $\mathbb{N}^m$ in this list. This 'indexing' and its inverse (onto the i^{th} component for each $1 \leq i \leq m$) are all computable; we will show how in the next section with the aid of *Church's thesis.*

We can use this idea to encode words as integers. Consider the set Σ^* of all words over the finite alphabet Σ. By placing an ordering $\{\sigma_1, \ldots, \sigma_n\}$ on Σ, we can represent each letter σ_i of Σ by the integer i. By restricting the shortlex ordering to $\{1, \ldots, n\}^m$ for each m, we get an induced ordering of Σ^m (words of length m) for each m: given a word $w \in \Sigma^m$, we can associate to it an m-tuple $(i_1, \ldots, i_m)$ representing the sequence of letters in w, and then we use the shortlex ordering on these associated tuples. Now, to produce an indexed list of Σ^*, we first take all words in Σ and order them (via their tuples) by the induced shortlex. Then take all words in Σ^2 and order them by the induced shortlex, and then Σ^3, and so on. Thus, for each $n \in \mathbb{N}$, we can construct the n^{th} element of Σ^* in this list.

Using this, we may re-interpret our previous question of 'take a word in the English language, and compute the number of letters in it' as 'take the *index* for a word in the English language, and compute the number of letters in it'. So it now makes sense to ask 'is this function computable?' We will always require our inputs/outputs to be integers. Thus,

From hereon we will take it as a given that the inputs/outputs of our algorithms are given by codes of various objects, either by explicitly giving an encoding, or implicitly without going in to the details.

So when we ask for "an algorithm to count the number of letters in a word", it is clear what we mean.

We now give a way of encoding *machines* as integers, as later we will want to compute things about machines. There are various ways to uniformly encode programs for register machines as integers. Some of these are bijective (one program $\leftrightarrow$ one integer). We give an encoding here which is not bijective (in particular, there are integers which do not correspond to programs).

Definition 1.21 (Encoding programs as natural numbers)**.**
For each line in the register machine program Q (a triple or quadruple) corresponding to state S_i ($1 \leq i \leq r$; we disregard S_0), we take the triple $(j,+,k)$ and encode it to the integer $2^j \cdot 5^k$, or the quadruple $(j,-,k,l)$ and encode it to the integer $2^j \cdot 3 \cdot 5^k \cdot 7^l$. Call this integer t_i. Now take this sequence $t_1,\dots,t_r$ and form the integer $n = 3^{t_1} \cdots p_r^{t_r}$. Write P_n for the program encoded by the integer n.

Given that a register machine program P has no intrinsic arity (that is, has no intrinsic 'number of variables' which it needs to take as input), we see that if P computes a k-variable function f then it also computes the $(k-1)$-variable function f' given by

$$f'(n_1,\dots,n_{k-1}) := f(n_1,\dots,n_{k-1},0)$$

by simply inputting $(n_1,\dots,n_{k-1},0)$ into P. We now remove this ambiguity:

Definition 1.22 (Functions from register machines)**.**
We write $f_{n,k}$ for the k-variable function computed by the register machine with program P_n, if P_n exists (that is, if n actually encodes a program).

We can now adapt Cantor's diagonal argument to construct an explicit function which is not partial recursive:

Lemma 1.23 (An explicit function which is not partial recursive)**.**
Define the function $g : \mathbb{N} \to \mathbb{N}$ via

$$g(n) := \begin{cases} f_{n,1}(n)+1 & \text{if } n \text{ codes a program and } f_{n,1}(n) \text{ is defined} \\ 0 & \text{otherwise} \end{cases}$$

Then this is an explicit definition of a function which is not partial recursive.

Proof. We proceed by contradiction. Suppose g were partial recursive. Then there must be some code N for which $g = f_{N,1}$. Now observe what happens if we try and compute $g(N)$. We see that, as N is a code, if $f_{N,1}(N)$ were defined then we would have $g(N) = f_{N,1}(N) + 1 \neq f_{N,1}(N) = g(N)$. Thus $f_{N,1}(N)$ is not defined. So by the definition of g we have $g(N) = 0$, thus giving that $f_{N,1}(N) = 0$, and so $f_{N,1}(N)$ is defined; a contradiction. □

Note that we need the clause '$g(n) = 0$ if $f_{n,1}(n)$ is undefined'. If instead we had that '$g(n)$ is undefined if $f_{n,1}(n)$ is undefined', then g would indeed be partial recursive, and we will prove this later in Lemma 1.24. To understand why this is true, we first need to introduce *Church's thesis.*

1.5. **Church's thesis.**

We have given our definition of an *algorithm* in the previous section, in terms of register machines (and, equivalently, partial recursive functions). But this was not simply an arbitrary definition; it reflects some of our intuition of what an algorithm should do. We could, for example, have said that an algorithm is something that can be computed by a linear function. But it should be clear that this is far to restrictive a definition, and does not capture all the properties that we would want 'algorithms' to exhibit.

The intuitive idea we have tried to reflect when giving the definition of an algorithm is the following: an *executable process*, in the intuitive sense, is a *step-by-step deterministic process with a finite description at every step, a finite set of rules, and finite input/output.* This, of course, is not a formal definition, and so is unsatisfactory to mathematicians. But it is the idea that we want to axiomatise, by defining 'algorithms' in a suitable way.

We have seen that two seemingly independent definitions of 'functions we can compute', namely partial recursive functions and partial computable functions, actually yield the same set of functions. Alonzo Church[6] made the assertion (known as *Church's thesis*) that any *abstract theory of finite computation* (i.e., a theory of computation whose processes are executable processes, in the above intuitive sense) will always yield a set of partial computable functions which is contained in the set of partial recursive functions defined in Definition 1.10. Later in the course we will see theories of finite computation which define a strictly smaller set of partial computable functions. Church himself defined finite computation via the λ–calculus, whose set of partial computable functions is also identical to the set of partial recursive functions.

That is to say, Church asserted that the definition of an algorithm from Definition 1.18 is indeed the 'correct' definition to take, as it most accurately represents our intuitive understanding[7].

Of course, we cannot prove that *all* theories of finite computation lead to the same set of partial computable functions, as we don't know them all! However, every abstract theory of finite computation which has been proposed so far has been verified (mathematically) to compute at most the set of partial recursive functions (and this includes quantum computers; they are *faster*, but not *better*, than existing computing machines[8]).

So we now state the first (of three) parts of Church's thesis. Think of this as the "definition" part, where we have argued (philosophically) that our definition of an algorithm accurately reflects our intuitive understanding.

Church's thesis 1.
Any abstract theory of finite computation C will give at most *the set of partial recursive functions as its set of C partial computable functions from $\mathbb{N}^k$ to $\mathbb{N}$. Thus the most powerful theory of finite computation is given by register machines and their many equivalents.*
That is, the definition of an algorithm *from Definition* 1.18 *is the 'correct' definition to work with.*

[6]Alan Turing made the same assertion, hence this is often referred to as the *Church-Turing thesis*. For brevity, we shall continue to call it Church's thesis.

[7]Just like the definition of a continuous function; something that was the subject of (100 years of) debate, but is accepted now as the 'correct' definition to work with.

[8]They still use finitely-many *bits* in their computation, so are a high-tech variation on a classical theme, in the same way that a nuclear reactor is a high-tech kettle.

Next, consider the encoding of register machines as natural numbers, as done in Definition 1.21. As mentioned, this encoding is not bijective; if integer n encodes a program P_n then $(n)_i$ can only have prime divisors $2, 3, 5, 7$. But we can describe an executable process to determine if an integer represents a program; the reverse of Definition 1.21 where we break down n into products of distinct prime powers, then break down those powers and make sure they only have prime factors $2, 3, 5, 7$ (where the power of 3 is at most 1). A long and tedious exercise would be to verify, either by constructing a suitable register machine, or by composing suitable partial recursive functions, that the following (characteristic) function $f : \mathbb{N} \to \mathbb{N}$ given by:

$$f(n) := \begin{cases} 1 & \text{if } n \text{ codes a program} \\ 0 & \text{otherwise} \end{cases}$$

is indeed a computable function.

But even without seeing a full proof that such a register machine / total recursive function exists, you probably already have a good idea of how one might go about building such an algorithm, from the description of it given above: "Break down n into products of distinct prime powers, then break down those powers and make sure they only have prime factors $2, 3, 5, 7$, where the power of 3 is at most 1." In fact, you may already feel that such a description would actually be *sufficient proof* that there is an algorithm to do it; you don't need to see it written out in complete detail.

Well, Alonzo Church felt the same way. And so this brings us to the second part of Church's thesis; the idea that having a full step-by-step description of an executable process is *sufficient proof* that there exists an algorithm to carry out the process (given a suitable encoding). Think of this as the "my arguments should be enough to convince you" part of Church's thesis.

Church's thesis 2.
Any informal written description of a step-by-step deterministic process with a finite description at every step, a finite set of rules, and finite input/output, starting with some tuple in $\mathbb{N}^k$ and with only integer output, is equivalent to some register machine computation.

This may seem counter-intuitive at first; how can a *description* of an executable process be enough to prove that there is an algorithm (= register machine) that performs the computation? Surely such a 'proof' is not rigorous, as it is simply a convincing argument. But most 'proofs' that we see in mathematics are exactly that: a convincing argument. Unless you're in the habit of doing all your proofs via predicate logic[9], they are no more than 'convincing arguments'. So that is what you have seen here. The *description* of the process should be enough to convince you that it can (with some effort) be simulated by a register machine.

But we can say even more than this. Thinking back to the example above again, of checking if an integer n encodes a register machine, we see that the *verbal/written* description given doesn't just show that process can be made algorithmic; it *describes* what the algorithm does, in sufficient detail to allow us to write down the register machine program explicitly (albeit with a bit of effort). In the same way that we encoded words over alphabet Σ as integers, we

[9]*Do not* do all your proofs via predicate logic, unless you want to spend the next 1000 years completing tripos.

can also encode all finite phrases in the English language as integers (we omit the details here; it is very similar to the previous example). Thus we can now state the last part of Church's thesis; think of this as the "If I can describe it, I can build it" part of Church's thesis.

Church's thesis 3.
There is a total algorithm (= recursive function $h : \mathbb{N} \to \mathbb{N}$) that, given a code n for such a finite English description of a process as in part 2 of Church's thesis, will produce a code $h(n)$ for a register machine $P_{h(n)}$ that carries out the process so described.

So what we've said here is that having a complete verbal/written description of an executable process is as good as having the actual register machine that carries out the computation; we can always recover one from the other in a computable manner. Thus, producing the explicit register machine is equivalent to producing a verbal/written description of the process.

We summarise the three parts of Church's thesis here.

Church's thesis.

(1) *Any abstract theory of finite computation C will give* at most *the set of partial recursive functions as its set of C partial computable functions from $\mathbb{N}^k$ to $\mathbb{N}$. Thus the most powerful theory of finite computation is given by register machines and their many equivalents.*
That is, the definition of an algorithm *from Definition* 1.18 *is the 'correct' definition to work with.*

(2) *Any informal written description of a step-by-step deterministic process with a finite description at every step, a finite set of rules, and finite input/output, starting with some tuple in $\mathbb{N}^k$ and with only integer output, is equivalent to some register machine computation.*

(3) *There is a total algorithm (= recursive function $h : \mathbb{N} \to \mathbb{N}$) that, given a code n for such a finite English description of a process as in part* 2 *of Church's thesis, will produce a code $h(n)$ for a register machine $P_{h(n)}$ that carries out the process so described.*

Now we can start to appeal to Church's thesis to show that certain functions are indeed partial recursive. First, in a slight abuse of notation, and making use of part 2 of Church's thesis, when we say things like "We describe an algorithm to compute XYZ...", we really mean "We describe a step-by-step deterministic process with a finite description at every step, a finite set of rules, and finite input/output, to compute XYZ..." Observe that, by part 2 of Church's thesis and our previous discussion on encodings, we know that this is equivalent to having built a register machine to carry out the process. Thus, even though we have defined 'algorithm' in a very strict sense via register machines, we can interpret it more broadly now.

We previously showed some of the following functions to be recursive:

• Arithmetic functions (addition, subtraction, multiplication, division with remainder).
• Computing reductions mod n.
• Primality testing.
• Computing gcd and lcm.

All the above processes have deterministic step-by-step descriptions with all the necessary finiteness conditions, and so we can simply argue by Church's thesis that there exist register machines which can compute them. This is substantially simpler than explicitly constructing the necessary register machines or partial recursive functions.

Recall that we earlier stated, without proof, that recognising whether an integer represented a register machine code was computable. Well, now we can simply argue this by Church's thesis. The reverse of Definition 1.21, where we break down n into products of distinct prime powers, then break down those powers and make sure they only have prime factors $2, 3, 5, 7$ (where the power of 3 is at most 1), is a step-by-step deterministic process with a finite description at every step, a finite set of rules, and finite input/output, starting with some integer and with only integer output. Thus, by Church's thesis, we see that the following function

$$f(n) := \begin{cases} 1 & \text{if } n \text{ codes a program} \\ 0 & \text{otherwise} \end{cases}$$

is indeed a computable function.

However, Church's thesis cannot be applied to a description of a process if it contains an 'existential step'. For example, consider the description 'Take a register machine program P defining a function on 1 variable and, if it has a non-empty domain, output the smallest integer on which it halts'. Such an integer is, if it exists, well-defined. However, this does not translate into a step-by-step process for computation.

We can now use Church's thesis to clarify some of the claims we made in the previous section. For example: when we defined the shortlex ordering (Definition 1.20) to produce an indexed list of $\mathbb{N}^m$, we said that this indexing and its inverse (onto the i^{th} component for each $1 \leq i \leq m$) are all computable. This is now immediate; we gave a step-by-step deterministic process for producing the indexing, and thus by Church's thesis we see that there is thus an algorithm (= register machine) which computes this, and so the indexing function and its m inverses are all computable (and obviously total).

Here is another example, which is a variant of Lemma 1.23.

Lemma 1.24 (An explicit function which is partial recursive)**.**
Define the function $g : \mathbb{N} \to \mathbb{N}$ via

$$g(n) := \begin{cases} f_{n,1}(n) + 1 & \text{if } n \text{ codes a program and } f_{n,1}(n) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Then this give a function which is partial recursive.

Proof. To see this, we again appeal to Church's thesis: we have a description of a process to compute the values of $g(n)$ when it is defined (that is, we simply check if n codes a program, and if it does, take the program P_n and run it with input n). If this process eventually terminates, then add one to R_1 and output its contents. This is a description of a process to compute g, and so g is partial recursive (but not necessarily total). □

Note that, in the above example, we have used Church's thesis to show that the function g is partial recursive. We have *not* shown that g is computable (= total recursive), and indeed g is definitely not total. Church's thesis can be used to show that a function is partial recursive (= can be simulated by a register machine), but often we need to use some additional mathematical argument(s) to verify when such functions are total.

1.6. **Recursively enumerable sets and diagonalisation.**

We have looked at *recursive* sets; those X for which the characteristic function $\mathcal{X}_X$ is total recursive. But what if we only had a partial recursive function f which, if defined, matches $\mathcal{X}_X$, but whose domain of definition merely contains X (but not necessarily $\mathbb{N} \setminus X$)? That is, f can tell us if $n \in X$, but doesn't always say when $n \notin X$.

Now that we know that the partial recursive functions are precisely the partial computable functions, we will write $f(n) \uparrow$ to mean '$f(n)$ is undefined', and $f(n) \downarrow$ to mean '$f(n)$ is defined', matching the notion of when a partial computable function doesn't/does halt.

We now need the notion of *diagonalising an algorithm.* Suppose we have a partial recursive function $f : \mathbb{N} \to \mathbb{N}$. Suppose also that we would like to know if $f(7)$ is defined. Then we simply take a register machine with program P_n which computes f, and start running $P_n(7)$. That is, we put 7 in register R_1, then apply the instructions of P_n step-by-step. If we reach the halting state S_0 in some finite number of steps, then we stop and can conclude that $f(7)$ is defined. In actual fact, we have just described an algorithm which takes as input a register machine P_m for any code m and, if $P_m(7) \downarrow$, will halt and confirm this (but will not halt if $P_m(7) \uparrow$) Thus, by Church's thesis, there is some register machine with program Q which simulates this. That is,

$$Q(m) = \begin{cases} 1 & \text{if } m \text{ codes a program and } P_m(7) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

Now suppose we take the same partial function f, and we want to know if either $f(7)$ or $f(9)$ is defined. We could do as before, and take P_n with input 7 and run the register machine. If $P_n(7) \downarrow$ then this process will halt and tell us 'yes'. But what if $P_n(7) \uparrow$ but $P_n(9) \downarrow$? Then we would need to 'wait until infinity' for $P_n(7)$ to finish, before moving on to computing the steps of $P_n(9)$. What we need to do is to *diagonalise the algorithms.* That is, do one step of $P_n(7)$, then one step of $P_n(9)$, then another step of $P_n(7)$, then another step of $P_n(9)$, and so on. If either of these eventually halt (and say it is $P_n(9)$ after 1460 steps), the our algorithm will terminate after $2 * 1460$ steps with the answer 'yes', which is what we want! It helps to write $P_n^t(k)$ to mean 'the register machine P_n, on input k, after t computational steps' (this would be completely described by the integer $h(1, k, t)$ from the proof of Theorem 1.16).

Here is a picture of the order of the computational steps in the process we described:

$$\begin{tikzcd}
P_n^1(7) \arrow[r] & P_n^1(9) \arrow[dl] \\
P_n^2(7) \arrow[r] & P_n^2(9) \arrow[dl] \\
P_n^3(7) \arrow[r] & P_n^3(9)
\end{tikzcd}$$

Now suppose we wanted to know 'does f halt on *any* input?' Again, we can diagonalise. But this time we need an infinite diagonal process, which explores infinitely many computational processes at once. To do this, we first need to do one step of $P_n(1)$, then one step of $P_n(2)$ followed by one more step of

$P_n(1)$, then one step of $P_n(3)$ followed by one more step of $P_n(2)$ followed by one more step of $P_n(1)$, and so on. This is a much larger diagonal process, but can still be simulated by one register machine as we have given a complete verbal description of the algorithm. Here is a diagram of the first 10 steps of this process. To make it clearer, we have written $[t]$ next to the t^{th} step of the process.

$$\begin{tikzcd}
{[1]\ P_n^1(1)} & {[2]\ P_n^1(2)} \arrow[dl] & {[4]\ P_n^1(3)} \arrow[dl] & {[7]\ P_n^1(4)\ldots} \arrow[dl] \\
{[3]\ P_n^2(1)} & {[5]\ P_n^2(2)} \arrow[dl] & {[8]\ P_n^2(3)\ldots} \arrow[dl] & \vdots \\
{[6]\ P_n^3(1)} & {[9]\ P_n^3(2)\ldots} \arrow[dl] & \vdots & \\
{[10]\ P_n^4(1)\ldots} & \vdots & & \\
\vdots & & &
\end{tikzcd}$$

In general, when we have several register machines with an input each, and we want to run them all at the same time until one of them halts, we can do so via diagonalisation. Moreover, this is an algorithmic process (we have given a full description of the algorithm, above). So by Church's thesis there is a register machine to do this computation, and we can construct such a machine from the description given.

Definition 1.25 (Recursively enumerable sets)**.**
A set $E \subseteq \mathbb{N}$ is said to be *recursively enumerable*[10] (abbreviated to *r.e.*) if the function ϕ_E defined by

$$\phi_E(n) := \begin{cases} 0 & \text{if } n \in E \\ \uparrow & \text{otherwise} \end{cases}$$

is partial recursive.

So ϕ_E will always tell us if $n \in E$, but won't say anything about when $n \notin E$. We will now show why recursively enumerable sets are named so; it because we can *enumerate* them in a recursive manner (that is, start a recursive process which eventually outputs each element in the set).

Theorem 1.26 (Equivalent definitions of recursively enumerable sets)**.**
For a set $E \subseteq \mathbb{N}$, the following are equivalent:
a) $E = \{f_{n,k}(m_1, \ldots, m_k) \mid (m_1, \ldots, m_k) \in \mathbb{N}^k\}$ for some fixed $k \geq 1$ and some fixed n. That is, E is the range of some partial recursive function on some number of variables.
b) $E = \{m \in \mathbb{N} \mid f_{n,1}(m) \downarrow\}$ for some fixed n. That is, E is the domain of

[10]Computability theorists sometimes call these *computably enumerable*, abbreviated to *c.e.*

definition of some partial recursive function on 1 *variable.*
c) The function ϕ_E *defined by*

$$\phi_E(n) := \begin{cases} 0 & \text{if } n \in E \\ \uparrow & \text{otherwise} \end{cases}$$

is partial recursive. That is, E is recursively enumerable.
d) The function ψ_E *defined by*

$$\psi_E(n) := \begin{cases} n & \text{if } n \in E \\ \uparrow & \text{otherwise} \end{cases}$$

is partial recursive.

Proof.
$(b) \Rightarrow (c)$: Given a program P_n for computing the function $f_{n,1}$ with domain E, we modify it by inserting an instruction which is 'triggered' before the halting state; this empties register R_1 and then proceeds to the halting state. Explicitly, we add a state S_{n+1} (where $S_0, \ldots, S_n$ are the existing states). Then, for each instruction of the form $(j, +, 0)$ or $(j, -, k, 0)$ or $(j, -, 0, l)$, we replace it with $(j, +, n+1)$ (resp. $(j, -, k, n+1)$, $(j, -, n+1, l)$). Then we add the instruction $(1, -, n+1, 0)$ for state S_{n+1}.
$(c) \Rightarrow (d)$: Given a program P for computing ϕ_E, let r be the upper register index of P. Now insert a new state/instruction pair which copies R_1 to R_{r+1} at the beginning of the computation, and another state/instruction pair which adds R_{r+1} to R_1 just before the program reaches the halting state.
$(d) \Rightarrow (a)$: This is immediate.
$(a) \Rightarrow (b)$: Given the program P_n for computing the function $f_{n,k} : \mathbb{N}^k \to \mathbb{N}$ with range E, we describe the following algorithm Q. For each $t \in \mathbb{N}$, we start a diagonal process which starts computing $f_{n,k}$ for all of its inputs (recall that we have an ordered listing of all the elements of $\mathbb{N}^k$). Each time $f_{n,k}(m_1, \ldots, m_k)$ halts in this diagonal process, compare the output to t; if we eventually find one such output is equal to t, then Q terminates on t and outputs 1 (if we never find such an output, then Q is undefined on t). As we have given a full description of the algorithm for Q, then by Church's thesis we can find a register machine which computes Q, and thus a partial computable function whose domain is E. $\square$

Since the definition of r.e. is one of the conditions above (c), then all the above conditions are equivalent to being r.e. We will make use of these conditions interchangeably in later proofs, depending on which is more convenient to work with at the time.

1.7. **Properties of recursively enumerable sets.**

There is actually a stronger form of Theorem 1.26 (a) when considering non-empty r.e. sets:

Theorem 1.27. *Let $E \subseteq \mathbb{N}$ be non-empty. Then E is r.e. if and only if it is the range of some total recursive function on some number of variables.*

Proof. By Theorem 1.26 (a), the range of a total recursive function will be r.e. and non-empty. So we need only show that, if E is r.e. and non-empty, then it is the range of some total recursive function. If E is finite, then it can be written $E = \{e_1, \ldots, e_k\}$ for some $k \geq 1$. So we can define a total recursive function $f : \mathbb{N} \to \mathbb{N}$ whose range is E by:

$$f(n) := \begin{cases} e_n & \text{if } n \leq k \\ e_k & \text{if } n > k \end{cases}$$

If E is infinite, then it is the domain of some partial recursive function $g : \mathbb{N} \to \mathbb{N}$. So we start a diagonal process which starts computing $g(n)$ for each $n \in \mathbb{N}$. Then we define a new function f as follows: we assign $f(1)$ to be the first such n for which the diagonal process gives that $g(n)$ halts, $f(2)$ to be the second such n for which the diagonal process gives that $g(n)$ halts, and so on. By Church's thesis, we have defined a total recursive function f on 1 variable whose range is precisely E.

Note that when we say 'first such n' when defining f, we mean 'when running the diagonal process, the n for which the diagonal process give the first conclusive halting of g'. For example, we might have that $g(5)$ and $g(8)$ both halt, but in the diagonal process we see $g(8)$ halt long before we see $g(5)$ halt; in this case we would have $f(a) = 8$ and $f(b) = 5$ for some pair $a < b$. $\square$

Given condition (b) of Theorem 1.26, we see that each register machine program P_n corresponds to a recursively enumerable set, and in particular this set is the domain of definition of $f_{n,1}$. For simplicity, we will write the domain of definition of $f_{n,1}$ from hereon as

$$W_n := \{x \in \mathbb{N} \mid f_{n,1}(x) \downarrow\}$$

We call W_n the n^{th} *recursively enumerable set.* Of course, this definition is only valid when n actually encodes a register machine program P_n.

It turns out that recursive sets and recursively enumerable sets are closely related:

Theorem 1.28. *Let $E \subseteq \mathbb{N}$. Then E is recursive if and only if both E and $\mathbb{N} \setminus E$ are recursively enumerable.*

Proof. If E is recursive then its characteristic function $\mathcal{X}_E$ is a total recursive function. Thus the functions ϕ_E and $\phi_{\mathbb{N}\setminus E}$ from Definition 1.25 are partial recursive. This can be seen by defining an algorithm which takes an integer n and computes $\mathcal{X}_E(n)$; if this is 1 then the algorithm outputs 0, and if this is 0 then the algorithm enters a non-terminating loop. By Church's thesis, we have given a description of ϕ_E, which means that it is partial recursive. A similar argument works for $\phi_{\mathbb{N}\setminus E}$.

Conversely, if both E and $\mathbb{N} \setminus E$ are recursively enumerable, then ϕ_E and $\phi_{\mathbb{N}\setminus E}$ are partial recursive. So, for each $n \in \mathbb{N}$, start a diagonal process which computes $\phi_E(n)$ and $\phi_{\mathbb{N}\setminus E}(n)$; if (and only if) $\phi_E(n)$ halts, then $\mathcal{X}_E(n) = 1$, and if (and only if) $\phi_{\mathbb{N}\setminus E}(n)$ halts then $\mathcal{X}_E(n) = 0$. Moreover, precisely one of these will halt for each n. As we have given a full description of an algorithm computing $\mathcal{X}_E$, then by Church's thesis it is a total recursive function, and hence E is recursive. $\square$

There are many interesting examples of r.e. sets.

Definition 1.29 (Diophantine sets)**.**
A set $X \subseteq \mathbb{N}^k$ is *Diophantine* if there is an integer polynomial P on $k + l$ variables such that

$$X = \{(n_1, \ldots, n_k) \in \mathbb{N}^k \mid (\exists (m_1, \ldots, m_l) \in \mathbb{N}^l)(P(n_1, \ldots, n_k, m_1, \ldots, m_l) = 0)\}$$

Lemma 1.30. *Every Diophantine set is r.e.*

Proof. Let $X \subseteq \mathbb{N}^k$ be diophantine, with associated integer polynomial P on $k+l$ variables. Then, given $(n_1, \dots, n_k) \in \mathbb{N}^k$, we can start a diagonal process which starts computing $P(n_1, \dots, n_k, m_1, \dots, m_l)$ for every $(m_1, \dots, m_l) \in \mathbb{N}^l$ (recall that we have an ordered listing all elements of $\mathbb{N}^l$). If any of these parallel processes ever terminate, then we say that $(n_1, \dots, n_k) \in X$. This is a complete description of an algorithm which halts iff $(n_1, \dots, n_k) \in X$, and thus by Church's thesis we have that X is r.e. □

Surprisingly, the converse statement to the above theorem is also true. It is a deep result, posed as one of Hilbert's famous problems from 1900 (his 10^{th} problem) and solved by Matijasevič, but is beyond the scope of this course. We state it here for completeness.

Theorem 1.31 (Matijasevič's theorem)**.** *Every r.e. set is Diophantine.*

Recursively enumerable sets satisfy some useful closure properties.

Lemma 1.32. *Let $E \subseteq \mathbb{N}$ be an r.e. set. Then the set $C \subset E$ given by*

$$C(E) := \{e \in E \mid e \text{ is a code for a register machine } \}$$

is also r.e. Moreover, given an index n such that $W_n = E$, we can construct an index m such that $W_m = C(E)$.

Proof. Recall that the function $f : \mathbb{N} \to \mathbb{N}$ given by

$$f(n) := \begin{cases} 1 & \text{if } n \text{ codes a program} \\ 0 & \text{otherwise} \end{cases}$$

is recursive. So, take an enumeration for E (the domain of some partial recursive function g). Begin a diagonal process which starts computing $g(n)$ for each $n \in \mathbb{N}$. For each n on which $g(n)$ halts, test if n is a code using f. If so, output 0; if not, enter some non-terminating loop. As this is a complete description of an algorithm which computes ϕ_C, we can apply Church's thesis to conclude that ϕ_C is partial recursive, and thus that C is r.e. Moreover, a further application of Church's thesis allows us to compute an index m for a register machine P_m which computes ϕ_C. That is, $\phi_C = f_{m,1}$, and so $C = W_m$. □

Theorem 1.33 (Unions and intersections of r.e. sets)**.**
1. *Let $I \subseteq \mathbb{N}$ be a (possibly infinite) r.e. set of integers, and $I' \subset I$ those which are codes for register machines. Then the union of r.e. sets*

$$\bigcup_{n \in I'} W_n$$

is again r.e.
2. *Let $J \subseteq \mathbb{N}$ be a finite set of integers, and $J' \subset J$ those which are codes for register machines. Then the intersection of r.e. sets*

$$\bigcap_{n \in J'} W_n$$

is again r.e. This does not *extend to an infinite intersection, even if the index set recursive.*

In both case 1 and 2, we can construct an index for the union/intersection of these r.e. sets.

Proof.

1. For each $x \in \mathbb{N}$, begin a diagonal process which starts computing $f_{n,1}(x)$ for each $n \in I'$, and look to see if any of these halt. This process will terminate iff x lies in at least one W_n. As this is a description of an algorithm to enumerate $\bigcup_{n\in I'} W_n$, we have that it is thus r.e. by Church's thesis, and moreover this explicit description allows us to construct an index for the r.e. set $\bigcup_{n\in I'} W_n$.

2. For each $x \in \mathbb{N}$, begin a diagonal process which starts computing $f_{n,1}(x)$ for each $n \in J'$, and look to see if all of these halt. This process will terminate iff x lies in all the W_n $(n \in J')$. As this is a description of an algorithm to enumerate $\bigcap_{n\in I'} W_n$, we thus have that it is r.e. by Church's thesis, and moreover this explicit description allows us to construct an index for the r.e. set $\bigcap_{n\in J'} W_n$. $\square$

1.8. **Universality and undecidability.**

We saw, in Lemma 1.23, a function g that was constructed to 'contradict' every partial computable function on 1 variable at least once. We will now invert this idea, to construct a function u which simulates every partial computable function, simultaneously.

Theorem 1.34 (Universal partial recursive function)**.**
There exists a partial recursive function $u : \mathbb{N}^3 \to \mathbb{N}$ *such that*

$$u(n,k,m) = \begin{cases} r & \text{if: } n \text{ codes a program} \\ & \quad \text{and } m \text{ codes a } k\text{-tuple of integers } ((m)_1,\ldots,(m)_k) \\ & \quad \text{and } f_{n,k}((m)_1,\ldots,(m)_k) \text{ is defined and equals } r. \\ \uparrow & \text{otherwise} \end{cases}$$

Such a partial recursive function is said to be universal*, as it is capable of simulating any program for a register machine, and thus simulating any partial recursive function.*

Proof. We describe an algorithm to compute u; by Church's thesis this will suffice. On input (n,k,m), we first check if n is a code; if not, enter some non-terminating loop. If so, decode m as a k-tuple $((m)_1,\ldots,(m)_k)$ if possible (i.e., if $m > 0$ and the largest prime dividing m is at most p_k); if this is unsuccessful, enter a non-terminating loop. Otherwise, simulate the computation of program P_n with input registers $((m)_1,\ldots,(m)_k,0,0,\ldots)$. If $P_n((m)_1,\ldots,(m)_k)$ eventually halts, then we output the contents of register R_1. $\square$

The implications of the existence of a universal partial recursive function (equivalently, a universal computing device) are what underpin the last 80 years of human technological advancements. To make this clear: a *universal partial recursive function* is often referred to as a *programmable computer*. With this, it is possible to construct *one* physical computation device, and then on it have the ability to simulate *all* possible computer programs, without any need to modify the hardware[11].

We now show that not every recursively enumerable set is recursive.

Theorem 1.35 (Undecidability and the halting set)**.**
The set

$$\mathbb{K} := \{n \in \mathbb{N} \mid f_{n,1}(n)\downarrow\}$$

[11]If you are still unconvinced of the importance of universal computation, imagine a world where every update of your favourite operating system (Windows, macOS, Ubuntu, Android, etc.) ships with a screwdriver.

is known as the halting set*. It is recursively enumerable, but not recursive.*

Proof. Given any $n \in \mathbb{N}$, we start computing $f_{n,1}(n)$; this halts iff $n \in \mathbb{K}$. As this is a description of a partial algorithm, then by Church's thesis we see that $\mathbb{K}$ is r.e.

Now, suppose that $\mathbb{N} \setminus \mathbb{K}$ were r.e.; we proceed by contradiction. For if it were, then there would exist some N such that $W_N = \mathbb{N} \setminus \mathbb{K}$ (that is, $\mathbb{N} \setminus \mathbb{K}$ is the domain of $f_{N,1}$). So now we ask whether N is in $\mathbb{K}$ or its complement. But we see that

$$\begin{aligned} N \in \mathbb{K} &\Leftrightarrow f_{N,1}(N)\downarrow \\ &\Leftrightarrow N \in W_N \\ &\Leftrightarrow N \in \mathbb{N} \setminus \mathbb{K} \\ &\Leftrightarrow N \notin \mathbb{K} \end{aligned}$$

which is a contradiction. □

What this is telling us is the following: Given a partial recursive function $f_{n,1}$, and an input t, we cannot compute *in advance* whether or not $f_{n,1}(t)$ is defined (that is, whether or not the computation $f_{n,1}(t)$ halts). If it is defined, we can verify this computationally. But if it is not defined, then there are cases when we can never 'be sure' that this is indeed the case. So even though $f_{n,1}$ completely defines W_n (existentially), we can't always compute membership in W_n algorithmically.

It is important to note that universality implies undecidability. That is, just using the statement 'there is a universal register machine', it is possible to prove the statement 'there is an r.e. set which is not recursive'. One might lament the fact that we encounter undecidability in the study of computation, but it is this very fact which allows us to build universal (also known as *programmable*) computers; if we did not have undecidability, then we could not have universality[12].

1.9. Reductions.

We now look at another way to show that certain sets are not recursive (or even r.e.), and that is via *reductions*. Intuitively, we look for ways to conclude, in a computational manner, membership in one set X from membership in another set Y. Thus, if we know that we can't decide membership in X, then it means we can't decide membership in Y.

Definition 1.36 (Many-one reductions)**.**
Given two sets $A, B \subseteq \mathbb{N}$, a *many-one reduction* of A to B is a total recursive function $f : \mathbb{N} \to \mathbb{N}$ such that, for all $n \in \mathbb{N}$, we have

$$n \in A \Leftrightarrow f(n) \in B$$

If there is a many-one reduction of A to B, then we say that A *many-one reduces to* B, or A *is many-one reducible to* B, and we write this as $A \leq_m B$.

So in order to compute membership of n in A, we evaluate the function $f(n)$ and then 'ask B one question: Is $f(n)$ in B or not?' If so, $n \in A$, if not, $n \notin A$; in either case, we *cannot* do any computation after this question. The name 'many-one reduction' comes from the fact that membership in A of several elements $n_1, n_2, \ldots$ can reduce to testing if one element lies in B (that

[12]Undecidability is not only interesting, but also inherently useful.

is, we might have $f(n_1) = f(n_2) = \ldots$). This differs from the notion of *Turing reductions* (which is beyond the scope of this course), where we are allowed to ask B whether several elements lie inside or outside it, and we can carry out computational steps in between these questions.

Lemma 1.37. *$A \leq_m B \Leftrightarrow \mathbb{N} \setminus A \leq_m \mathbb{N} \setminus B$.*

Proof. This follows immediately from the definition of many-one reductions. □

Many-one reductions help us identify sets which are/aren't recursive or r.e.

Lemma 1.38.
a) If $A \leq_m B$, and B is r.e., then so is A.
b) If $A \leq_m B$, and B is recursive, then so is A.

Proof.
a) Let $A \leq_m B$ via the total recursive function f. As B is r.e., it is the domain of some partial recursive function g (that is, $x \in B \Leftrightarrow g(x)\downarrow$). So A is thus the domain of $g \circ f$, which is partial recursive. Hence A is also r.e.

b) Let $A \leq_m B$ via the total recursive function f. Then the same map f is a many-one reduction of $\mathbb{N} \setminus A$ to $\mathbb{N} \setminus B$ (as $x \notin A \Leftrightarrow f(x) \notin B$). Thus, by a), A and $\mathbb{N} \setminus A$ are both r.e. (as B and $\mathbb{N} \setminus B$ are both r.e.). So A is recursive. □

We now introduce a useful idea, which shows that holding some of the variables of a partial recursive function fixed gives us another partial recursive function (which we can construct a register machine for). This is often referred to as *currying*, named after Haskell Curry. The theorem itself is called the 's-m-n theorem', named after the notation used in the original proof by Kleene[13].

Theorem 1.39 (The s-m-n theorem)**.**
For all $m, n > 0$, a partial function $h : \mathbb{N}^{m+n} \to \mathbb{N}$ is partial recursive if and only if there is a total recursive function $g : \mathbb{N}^m \to \mathbb{N}$ such that, for all $(e_1, \ldots, e_m, x_1, \ldots, x_n) \in \mathbb{N}^{m+n}$, we have that

$$h(e_1, \ldots, e_m, x_1, \ldots, x_n) = f_{g(e_1,\ldots,e_m),n}(x_1, \ldots, x_n)$$

Here '=' is interpreted to include 'one side is defined iff the other side is'.

Proof. Suppose h satisfies the hypotheses of the theorem; we show that it is partial recursive. Given input $(e_1, \ldots, e_m, x_1, ..., x_n)$, we first compute the total recursive function $g(e_1, \ldots, e_m) = M$, and then start the computation of $f_{M,n}(x_1, \ldots, x_n)$ via the register machine with program P_M (if M is not a code then we simply say that h is undefined for this input). If the computation of $f_{M,n}(x_1, \ldots, x_n)$ ever halts, then we take the output as the value of $h(e_1, \ldots, e_m, x_1, \ldots, x_n)$. Given that we have completely described an algorithm to partially compute h, then by Church's thesis we have that h is partial recursive.

Now, suppose that h is partial recursive. For each $(e_1, \ldots, e_m) \in \mathbb{N}^m$, we describe a function $k_{(e_1,\ldots,e_m)} : \mathbb{N}^n \to \mathbb{N}$ as follows: given input $(x_1, \ldots, x_n)$, start the computation of $h(e_1, \ldots, e_m, x_1, \ldots, x_n)$, and if this halts, take the output as $k_{(e_1,\ldots,e_m)}(x_1, \ldots, x_n)$. Thus we have a complete description of an algorithm which partially computes $k_{(e_1,\ldots,e_m)}$, thus by Church's thesis we can

[13]And not for any deeper or more insightful reason.

construct, from $(e_1, \dots, e_m)$, a code (call it $g(e_1, \dots, e_m)$) for a register machine $P_{g(e_1,\dots,e_m)}$ which partially computes the function $k_{(e_1,\dots,e_m)}$. That is,

$$k_{(e_1,\dots,e_m)} = f_{g(e_1,\dots,e_m),n} : \mathbb{N}^n \to \mathbb{N}$$

But this is a total algorithm which describes how to construct the (total) function $g : \mathbb{N}^m \to \mathbb{N}$, and so by another application of Church's thesis we see that g is total recursive. As $h(e_1, \dots, e_m, x_1, \dots, x_n) = f_{g(e_1,\dots,e_m),n}(x_1, \dots, x_n)$ by definition, we have that h satisfies the required conditions. □

We use this to show that the halting set $\mathbb{K}$ is the strongest r.e. set under many-one reductions, in the following sense.

Theorem 1.40. *A set $X \subseteq \mathbb{N}$ is r.e. if and only if $X \leq_m \mathbb{K}$.*

Proof. From Lemma 1.38, we see that if $X \leq_m \mathbb{K}$ then X must be r.e. (as $\mathbb{K}$ is). Now, suppose that X is r.e. Define the partial function $f : \mathbb{N}^2 \to \mathbb{N}$ via

$$f(e,n) := \begin{cases} 1 & \text{if } e \in X \\ \uparrow & \text{otherwise} \end{cases}$$

Then f is partial recursive; given an input (e,n) we begin computing $\phi_X(e)$, and this will halt iff $e \in X$. When it does, output 1 for $f(e,n)$. Thus, by Church's thesis, f is partial recursive. So by Theorem 1.39, there is a total recursive function $g : \mathbb{N} \to \mathbb{N}$ with $f(e,n) = f_{g(e),1}(n)$ for all $(e,n) \in \mathbb{N}^2$. So now we see that

$$\begin{aligned} e \in X &\Leftrightarrow f(e, g(e)) \downarrow \\ &\Leftrightarrow f_{g(e),1}(g(e)) \downarrow \\ &\Leftrightarrow g(e) \in \mathbb{K} \end{aligned}$$

So $e \in X \Leftrightarrow g(e) \in \mathbb{K}$, where g is a total recursive function. Thus $X \leq_m \mathbb{K}$. □

The idea here is that, with absolute knowledge of $\mathbb{K}$, we have absolute knowledge of each r.e. set, in a computable way.

We can also apply the s-m-n theorem to prove that every computable function has a 'fixed point'. This is known as the recursion theorem.

Theorem 1.41 (The recursion theorem)**.**
For each total recursive function $h : \mathbb{N} \to \mathbb{N}$, there is some $n \in \mathbb{N}$ with $f_{n,1} = f_{h(n),1}$ as functions.

Proof. Consider the function $g : \mathbb{N}^2 \to \mathbb{N}$ given by

$$g(x,y) := u(\ h(u(x, 1, 3^x)),\ 1,\ 3^y\ )$$

where u is the universal partial recursive function from Theorem 1.34. By the s-m-n theorem, we can 'curry' this, and find a total recursive function on one variable (say $f_{m,1}$) for which

$$g(x,y) = f_{f_{m,1}(x),1}(y) \quad \forall x, y \in \mathbb{N}$$

Let $n = f_{m,1}(m)$; this will be our fixed point. Then for all y with $f_{n,1}(y)\downarrow$ we have

$$\begin{aligned} f_{n,1}(y) &= f_{f_{m,1}(m),1}(y) \\ &= g(m,y) \\ &= u(\ h(u(m,1,3^m)),\ 1,\ 3^y\ ) \\ &= u(\ h(f_{m,1}(m))),\ 1,\ 3^y\ ) \quad \text{by definition of } u \\ &= u(\ h(n),\ 1,\ 3^y\ ) \\ &= f_{h(n),1}(y) \end{aligned}$$

The above reasoning also shows that $f_{n,1}(y)\uparrow \Rightarrow f_{h(n),1}(y)\uparrow$.
Thus $f_{n,1} = f_{h(n),1}$ as functions. □

1.10. **Rice's theorem.**

One would, of course, like to compute things *about* r.e. sets. It would be useful, for example, to be able to determine (in a computable way) whether or not W_n is all of $\mathbb{N}$, as this would tell us precisely when $f_{n,1}$ is total. We will soon see that this is not possible; moreover, there is *no* (non-trivial) property of r.e. sets that we can compute!

Definition 1.42. A *property* of r.e. sets is a map

$$\rho : \{X \subseteq \mathbb{N} \mid X \text{ is r.e.}\} \to \{0,1\}$$

where $0,1$ represent 'false' and 'true' respectively.

For example, the property of 'being empty' is represented by the map

$$\rho(X) := \begin{cases} 1 & \text{if } X = \emptyset \\ 0 & \text{if } X \neq \emptyset \end{cases}$$

In order to compute whether an r.e. set has a particular property or not, we need a finite way to describe this r.e. set. We can take a code n for the register machine P_n which describes the characteristic function of the r.e. set, but note that it is the *set* which does or doesn't have the property, independent of which register machine we pick to describe it (and there may be many). So really, we are computing $\rho(W_n)$ (and actually, we can view this as computing $\rho(n)$). So which properties can we compute?

Example 1.43. *The property 'being non-empty' is r.e. but not recursive. That is, the set* $I = \{n \in \mathbb{N} \mid \text{ n codes a program and } W_n \neq \emptyset\}$ *is r.e., but not recursive.*

Proof. Take n and compute if it is a code for a program. If so, start a diagonal process to begin computing $f_{n,1}(1), f_{n,1}(2), \ldots$. One of these will terminate iff W_n is non-empty, and so this index set is r.e. by Church's thesis.
On the other hand, take an integer n, and define a partial function g via

$$g(n,x) := \begin{cases} 1 & \text{if } n \in \mathbb{K} \\ \uparrow & \text{otherwise} \end{cases}$$

So by Church's thesis g is partial recursive, and by Theorem 1.39 there is a recursive function $h : \mathbb{N} \to \mathbb{N}$ such that $g(n,x) = f_{h(n),1}(x)\ \forall (n,x) \in \mathbb{N}^2$. If $n \in \mathbb{K}$ then $W_{h(n)} = \mathbb{N} \neq \emptyset$. If $n \notin \mathbb{K}$ then $W_{h(n)} = \emptyset$. Thus $n \in \mathbb{N} \setminus \mathbb{K} \Leftrightarrow W_{h(n)} = \emptyset \Leftrightarrow h(n) \in \mathbb{N} \setminus I$, and so we have a many-one reduction from a non-r.e. set $\mathbb{N} \setminus \mathbb{K}$ to the set $\mathbb{N} \setminus I$, and so the latter is not r.e. Hence I is not recursive. □

Definition 1.44. A property of r.e. sets ρ is said to be *nontrivial* if there exist two r.e. sets A, B such that $\rho(A) = 0$ and $\rho(B) = 1$. That is, not all sets have (or do not have) the property described.

It turns out that the only properties of r.e. sets we can algorithmically recognise are the trivial ones[14].

Theorem 1.45 (Rice's theorem)**.**
Let C be a non-trivial class of r.e. sets, and I the set of indices which code programs and give r.e. sets in C. That is,

$$I = \{n \in \mathbb{N} \mid n \text{ encodes a register machine and } W_n \in C\}$$

If $\emptyset \notin C$ then $\mathbb{K} \leq_m I$; if $\emptyset \in C$ then $\mathbb{K} \leq_m (\mathbb{N} \setminus I)$.

Proof. There are two cases to consider here.
Case 1: $\emptyset \notin C$. In this case, fix any r.e. set $\emptyset \neq A \in C$. Now define the following partial recursive function $g : \mathbb{N}^2 \to \mathbb{N}$ by

$$g(n,x) = \begin{cases} 1 & \text{if } n \in \mathbb{K} \text{ and } x \in A \\ \uparrow & \text{otherwise} \end{cases}$$

This is a description of how to compute if g halts on a given input, and so by Church's thesis g is partial recursive. So by Theorem 1.39 there is a total recursive function $h : \mathbb{N} \to \mathbb{N}$ such that $g(n,x) = f_{h(n),1}(x)\ \forall (n,x) \in \mathbb{N}^2$. Notice that $n \in \mathbb{K} \Rightarrow W_{h(n)} = A \Rightarrow W_{h(n)} \in C$, and $n \notin \mathbb{K} \Rightarrow W_{h(n)} = \emptyset \Rightarrow W_{h(n)} \notin C$. Thus $n \in \mathbb{K} \Leftrightarrow h(n) \in I$, and so we have a many-one reduction $\mathbb{K} \leq_m I$.
Case 2: $\emptyset \in C$ (analogous to the first case). In this case, fix any r.e. set $\emptyset \neq A \notin C$. Now define the following partial recursive function $g : \mathbb{N}^2 \to \mathbb{N}$ by

$$g(n,x) = \begin{cases} 1 & \text{if } n \in \mathbb{K} \text{ and } x \in A \\ \uparrow & \text{otherwise} \end{cases}$$

This is a description of how to compute if g halts on a given input, and so by Church's thesis g is partial recursive. So by Theorem 1.39 there is a total recursive function $h : \mathbb{N} \to \mathbb{N}$ such that $g(n,x) = f_{h(n),1}(x)\ \forall (n,x) \in \mathbb{N}^2$. Notice that $n \in \mathbb{K} \Rightarrow W_{h(n)} = A \Rightarrow W_{h(n)} \notin C$, and $n \notin \mathbb{K} \Rightarrow W_{h(n)} = \emptyset \Rightarrow W_{h(n)} \in C$. Thus $n \in \mathbb{K} \Leftrightarrow h(n) \in \mathbb{N} \setminus I$, and so we have a many-one reduction $\mathbb{K} \leq_m \mathbb{N} \setminus I$. ☐

Corollary 1.46. *Every non-trivial property of r.e. sets is undecidable (i.e., nonrecursive). That is, if ρ is a non-trivial property of r.e. sets, then the set*

$$\{n \in \mathbb{N} \mid n \text{ encodes a register machine and } \rho(W_n) = 1\}$$

is not recursive.

Thus, if you are given an r.e. set W_n and asked some non-trivial question about it (i.e., Is it finite? Is it empty? Does it contain more than 55 elements? Does it contain 9 but not 6? Is it recursive? Is it co-finite? Are all its elements even?), then you have no way of answering in an algorithmic manner. You may be able to answer the question for *some* particular cases of n, but not for all cases.

[14]Hopefully by now this does not come as a surprise to you.

2. REGULAR LANGUAGES AND FINITE-STATE AUTOMATA

We have seen the most general types of computing machines. Now we turn our attention to some more restrictive machines, which are less powerful but easier to work with.

2.1. Deterministic finite-state automata.

Definition 2.1 (Languages)**.**
Let $X = \{x_1, \ldots, x_n\}$ be a finite set. Then we define X^* to be the set of all finite strings of elements of X (including the empty string ϵ). We often refer to elements of X^* as *words*. A *language* over X is any subset of X^*.

Sometimes we can describe languages in a nice way. For example, take $X = \{0, 1\}$. Then the following are all languages over X:

(1) All words that start with 0.
(2) All words that contain the same number of 0's and 1's.
(3) All words which, for some fixed n, are the binary expansion for an integer which lies in W_n.

As we can see, some languages (like (3) above) will thus be undecidable! We can also take a different finite set, such as $A = \{a, b, c, \ldots, x, y, z\}$, and take our language to be all strings which give English words. We now give another way to describe some languages.

Definition 2.2 (Deterministic finite-state automata)**.**
A *deterministic finite-state automaton* (DFA) is a structure $D = (Q, \Sigma, \delta, q_0, F)$ consisting of the following:

(1) A finite set of *states* Q.
(2) A finite *input alphabet* Σ.
(3) A *transition function* $\delta : Q \times \Sigma \to Q$ which is total.
(4) A designated *start state* $q_0 \in Q$.
(5) A finite set of *accept states* $F \subseteq Q$.

The *input* of a DFA is any finite string $w = \sigma_1 \ldots \sigma_k \in \Sigma^*$. The DFA takes w, reads the first symbol σ_1 whilst 'in' the start state q_0, and then evaluates the transition function $\delta(q_0, \sigma_1) = p_1$ to 'move to' a new state. The DFA then reads the next symbol σ_2 of w, and evaluates $\delta(p_1, \sigma_2)$, and moves to the next state. This continues for the entire string w.

If at the end of this process the DFA is in one of the accept states F, then we say that w is *accepted* by A. Otherwise, we say that w is *rejected*.

The above description is not very intuitive. There are more convenient ways to describe DFA's, such as *transition diagrams* and *transition tables*.

Definition 2.3 (Transition diagrams)**.**
A *transition diagram* for a DFA $D = (Q, \Sigma, \delta, q_0, F)$ is a directed graph Γ_D with the following properties:

(1) The vertex set of Γ_D is precisely the set of states Q, labelled as such.
(2) For each $(q, \sigma) \in Q \times \Sigma$, we add a directed edge from q to $\delta(q, \sigma)$, and label this with σ.
(3) We add one additional directed edge from 'nowhere' to the vertex q_0, and label this 'start'.
(4) For clarity, we draw each vertex as the state q with a circle drawn around it; if $q \in F$ then we instead draw two circles around q.

Notice that every vertex will have precisely $|\Sigma|$ edges leading out of it; one for each $\sigma \in \Sigma$. Hence the name *deterministic*: our next move is completely determined. Now, to process an input $w = \sigma_1 \dots \sigma_k \in \Sigma^*$, we place a small movable marker at q_0, then read the first symbol σ_1 of w and move the marker along the edge out of q_0 labelled by σ_1 and to the adjacent vertex $\delta(q_0, \sigma_1)$. Then read the next symbol σ_2 of w and repeat the process. Upon reaching the end of the string w, note on which state the marker has reached. This completely describes the given DFA.

Transition diagrams are very intuitive, and give a clear 'picture' of what that DFA does. However, they take up a lot of space on a page. There is a more compact way to describe a DFA.

Definition 2.4 (Transition tables)**.**
A *transition table* for a DFA $D = (Q, \Sigma, \delta, q_0, F)$ is a table T_D with:

(1) Labels down the left of the table; one for each state in Q.
(2) Labels across the top of the table; one for each symbol in Σ.
(3) Entries in the middle of the table; position (q, a) is given value $\delta(q, a)$.
(4) For clarity, we place a star $*q$ next to the states down the left of the table which correspond to accepting states, and we place an arrow $\to q_0$ next to the state on the left of the table which is the start state.

As you can see, from this table we can read off the transition function δ, as well as the states Q, the accept states F, the start state q_0, and the alphabet Σ. This completely describes the given DFA.

2.2. **Regular languages.**

Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, we often want to know 'where does w end up when input into D?' We can define a function to do this.

Definition 2.5 (Extended transition function)**.**
Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We define the *extended transition function of* D, $\hat{\delta} : Q \times \Sigma^* \to Q$, inductively via:

$$\begin{aligned} \hat{\delta}(q, \epsilon) &:= q \text{ for } q \in Q \\ \hat{\delta}(q, \sigma) &:= \delta(q, \sigma) \text{ for } q \in Q,\ \sigma \in \Sigma \\ \hat{\delta}(q, \sigma_1 \dots \sigma_k) &:= \delta(\hat{\delta}(q, \sigma_1 \dots \sigma_{k-1}), \sigma_k) \text{ for } q \in Q,\ \sigma_1, \dots, \sigma_k \in \Sigma \end{aligned}$$

In particular, for any $w \in \Sigma^*$, $\hat{\delta}(q_0, w)$ tells us the state that we end up at when we input w into D.

Lemma 2.6. *Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Then, for all $1 \le l \le k$, all $\sigma_1 \dots \sigma_k \in \Sigma^*$, and all $q \in Q$, we have that*

$$\hat{\delta}(q, \sigma_1 \dots \sigma_k) = \hat{\delta}\big(\hat{\delta}(q, \sigma_1 \dots \sigma_l), \sigma_{l+1} \dots \sigma_k\big)$$

Proof. We proceed by induction. Clearly the statement is true for $k = 1$; assume it is true for all $k < m$. Then we have

$$\begin{aligned} \hat{\delta}(q, \sigma_1 \dots \sigma_m) &= \delta(\hat{\delta}(q, \sigma_1 \dots \sigma_{m-1}), \sigma_m) \quad \text{(definition of } \hat{\delta}\text{)} \\ &= \delta(\hat{\delta}(\hat{\delta}(q, \sigma_1 \dots \sigma_l), \sigma_{l+1} \dots \sigma_{m-1}), \sigma_m) \quad \text{(induction)} \\ &= \hat{\delta}(\hat{\delta}(q, \sigma_1 \dots \sigma_l), \sigma_{l+1} \dots \sigma_m) \quad \text{(definition of } \hat{\delta}\text{)} \end{aligned}$$

□

Definition 2.7 (Regular languages)**.**
Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We define the *language of* D, $\mathcal{L}(D)$, to be the words which are accepted by D. That is,

$$\mathcal{L}(D) := \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

This is the language of words over Σ which are taken to an accepting state in D. We say a language L is *regular* if $L = \mathcal{L}(D)$ for some DFA D.

2.3. Nondeterminism.

We now define a (seemingly) more general form of finite state automaton: one which can 'explore' several possibilities simultaneously. The only difference between these and the DFA's that we saw before is that these new automata have a non-deterministic transition function δ. We adopt the notation $\mathcal{P}(X)$ for the *power set* of a set X.

Definition 2.8 (Nondeterministic finite-state automata)**.**
A *nondeterministic finite-state automaton* (NFA) is a structure $N = (Q, \Sigma, \delta, q_0, F)$ consisting of the following:

(1) A finite set of *states* Q.
(2) A finite *input alphabet* Σ.
(3) A *transition function* $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ which is total.
(4) A designated *start state* $q_0 \in Q$.
(5) A finite set of *accept states* $F \subseteq Q$.

The *input* of an NFA is any finite string $w = \sigma_1 \ldots \sigma_k \in \Sigma^*$. The NFA takes w, reads the first symbol σ_1 whilst 'in' the start state q_0, and then evaluates the transition function $\delta(q_0, \sigma_1) = \{p_1, \ldots, p_m\}$ to simultaneously 'move to' all the new states. The NFA then reads the next symbol σ_2 of w, and evaluates $\delta(p, \sigma_2)$ for all $p \in \delta(q_0, \sigma_1)$, and simultaneously moves to all these states $\bigcup_{p \in \delta(q_0, \sigma_1)} \delta(p, \sigma_2)$. This continues for the entire word w.

If at the end of this process the NFA is in a configuration that contains at least one of the accept states F, then we say that w is *accepted* by N. Otherwise, we say that w is *rejected.*

Be aware that the transition function δ might give the empty set on certain inputs. That is, we might have $\delta(q, \sigma) = \emptyset$ for some $\sigma \in \Sigma$. This is fine.

The utility of having a nondeterministic transition function is that we can 'explore' many possibilities for an input word at once.

Definition 2.9 (Transition diagrams and transition tables for NFA's)**.**
We define the *transition diagram* Γ_N and *transition table* T_N for the NFA N in essentially the same way that we define them for a DFA. The only differences are:

(1) For the transition diagram of an NFA, we might have several directed edges out of the same state with the same label, or even none at all.
(2) For the transition table of an NFA, the entries in the interior of our table will be sets of states (including possibly the empty set).

NFA's are best understood via their transition diagrams. To see how an NFA N processes an input $w = \sigma_1 \ldots \sigma_k \in \Sigma^*$, we place a small marker at q_0, then read the first symbol σ_1 of w. Now we take more markers and move them along all the edges out of q_0 labelled by σ_1 and to the adjacent vertex set $\delta(q_0, \sigma_1)$, and then we remove the original marker. Then read the next symbol σ_2 of w

and repeat the process (It helps to have two colours of markers: red and blue. For each iteration, we take the new markers to be the opposite colour to the old markers, and then remove the old markers). Upon reaching the end of the word w, note on which states the markers are on; if any of these are accept states, the N accepts w, otherwise it is rejected. This completely describes the given NFA.

Be aware that, in the above description, we might reach state q, read symbol σ, and have that there is no edge out of q labelled σ (that is, $\delta_N(q, \sigma) = \emptyset$). This is fine. In this case, the marker in question merely 'drops off'.

Definition 2.10 (Extended transition function for NFA's)**.**
Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA. We define the *extended transition function of* N, $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$, inductively via:

$$\begin{aligned} \hat{\delta}(q, \epsilon) &:= \{q\} \text{ for } q \in Q \\ \hat{\delta}(q, \sigma) &:= \delta(q, \sigma) \text{ for } q \in Q,\ \sigma \in \Sigma \\ \hat{\delta}(q, \sigma_1 \dots \sigma_k) &:= \bigcup_{p \in \hat{\delta}(q, \sigma_1 \dots \sigma_{k-1})} \delta(p, \sigma_k) \end{aligned}$$

In particular, for any $w \in \Sigma^*$, $\hat{\delta}(q_0, w)$ gives us the set of states we end up at when we input w into N.

With this, we can say what the language of an NFA is.

Definition 2.11 (Language of an NFA)**.**
Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA. We define the *language of* N, $\mathcal{L}(N)$, to be the words which are accepted by N. That is,

$$\mathcal{L}(N) := \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

This is the language of words over Σ for which $\hat{\delta}(q_0, w)$ contains at least one accepting state of N.

2.4. **Equivalence of DFA's and NFA's.**

It is straightforward to show that any regular language is the language accepted by some NFA. What is less obvious is that the converse is true: every language accepted by an NFA is also accepted by some DFA. To do this, we employ what is known as the *subset construction* which takes an NFA N and produces a DFA D on the same alphabet such that $\mathcal{L}(N) = \mathcal{L}(D)$.

Definition 2.12 (The subset construction)**.**
Let $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ be an NFA. We define the following construction of a DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$, called the *subset construction*, as follows:

(1) $Q_D := \mathcal{P}(Q_N)$; the power set of Q_N.
(2) $F_F := \{S \subseteq Q_N \mid S \cap F \neq \emptyset\}$; the set of subsets of Q_N which intersect the accepting states F_N of N.
(3) For each $S \subseteq Q_N$ and each $a \in \Sigma$, we define

$$\delta_D(S, \sigma) := \bigcup_{p \in S} \delta_N(p, \sigma)$$

which is the set of states reached from the states in S by going along an edge labelled σ in Γ_N.

Observe that the start state of D constructed above is $\{q_0\}$; that is, the *set* containing q_0. This is because the set of states is $\mathcal{P}(Q_N)$

It is usually easiest to give the subset construction as a transition table, as it often becomes complicated when trying to draw the transition diagram.

Theorem 2.13 (Extended transition function in the subset construction)**.**
Let $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ be the DFA constructed from the NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ via the subset construction (Definition 2.12). Then, for any $w \in \Sigma^*$, we have

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

Proof. We proceed by induction on $|w|$. Clearly, if $|w| = 0$ (that is, $w = \epsilon$), then $\hat{\delta}_D(\{q_0\}, \epsilon) = \{q_0\}$, and so $\hat{\delta}_N(q_0, \epsilon) = \{q_0\}$.

Now suppose that $\hat{\delta}_D(\{q_0\}, v) = \hat{\delta}_N(q_0, v)$ for all v with $|v| \leq n$. Let w be a word of length $n+1$. Then $w = x\sigma$, where $\sigma \in \Sigma$ is the final symbol of w. By induction, as $x = n$, we have that $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$. Call this set $\{p_1, \dots, p_k\} \subseteq Q_N$. By definition of $\hat{\delta}$ for NFA's, we have

$$\hat{\delta}_N(q_0, x\sigma) = \bigcup_{q \in \hat{\delta}_N(q_0, x)} \delta_N(q, \sigma) = \bigcup_{i=1}^{k} \delta_N(p_i, \sigma)$$

Now, the subset construction gives that

$$\delta_D(\{p_1, \dots, p_k\}, \sigma) = \bigcup_{i=1}^{k} \delta_N(p_i, \sigma)$$

So we have that

$$\begin{aligned}
\hat{\delta}_D(\{q_0\}, w) &= \hat{\delta}_D(\{q_0\}, x\sigma) \\
&= \delta_D(\hat{\delta}_D(\{q_0\}, x), \sigma) \\
&= \delta_D(\{p_1, \dots, p_k\}, \sigma) \\
&= \bigcup_{i=1}^{k} \delta_N(p_i, \sigma)
\end{aligned}$$

Thus we have that $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$, and the induction is complete. □

We can now use this to say something about the accepted languages of these automata.

Theorem 2.14 (Equivalence of language in the subset construction)**.**
Let $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ be the DFA constructed from the NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ via the subset construction. Then $\mathcal{L}(D) = \mathcal{L}(N)$.

Proof. From the previous theorem, we see that, for any $w \in \Sigma^*$,

$$\begin{aligned}
D \text{ accepts } w &\Leftrightarrow \hat{\delta}_D(\{q_0\}, w) \text{ contains a state in } F_N \\
&\Leftrightarrow \hat{\delta}_N(q_0, w) \text{ contains a state in } F_N \\
&\Leftrightarrow N \text{ accepts } w
\end{aligned}$$

Thus $\mathcal{L}(D) = \mathcal{L}(N)$, as both D and N have the same alphabet Σ. □

Theorem 2.15. *A language L is accepted by some DFA if and only if it is accepted by some NFA.*

Proof. We showed in Theorem 2.14 that, from any NFA, we can construct a DFA on the same alphabet which accepts the same language.

So now suppose we have that L is accepted by some DFA $D = (Q, \Sigma, \delta_D, q_0, F)$. Take a transition diagram for D. Then this also defines a transition diagram for an NFA N with the same states, same alphabet, and same accepting states. We need only remark that the transition function δ_N for N will be given by

$$\delta_N(q, \sigma) := \{\delta_D(q, \sigma)\}$$

That is, if $\delta_D(q, \sigma) = p$, then $\delta_N(q, \sigma) = \{p\}$. We prove by induction on $|w|$ that, if $\hat{\delta}_D(q_0, w) = p$, then $\hat{\delta}_N(q_0, w) = \{p\}$:
Basis: Let $|w| = 1$, so $w = \sigma \in \Sigma$. We have defined $\delta_N(q_0, \sigma) = \{\delta_D(q_0, \sigma)\}$, so we're done.
Induction: Suppose, for all v with $|v| \leq n$, we have $\hat{\delta}_N(q_0, v) = \{\hat{\delta}_D(q_0, v)\}$. Take w with $|w| = n + 1$, then $w = x\sigma$ for some $x \in \Sigma^*, \sigma \in \Sigma$. Now we have that:

$$\begin{aligned}
\hat{\delta}_N(q_0, x\sigma) &= \bigcup_{p \in \hat{\delta}_N(q_0, x)} \delta_N(p, \sigma) \\
&= \bigcup_{p \in \{\hat{\delta}_D(q_0, x)\}} \delta_N(p, \sigma) \\
&= \delta_N(\hat{\delta}_D(q_0, x), \sigma) \\
&= \{\delta_D(\hat{\delta}_D(q_0, x), \sigma)\} \\
&= \{\hat{\delta}_D(q_0, x\sigma)\}
\end{aligned}$$

Thus D and $N = (Q, \Sigma, \delta_N, q_0, F)$ accept the same words, so $\mathcal{L}(D) = \mathcal{L}(N)$. □

So we see that, whenever we want to show that a language L is regular, it suffices to produce a DFA *or* an NFA which accepts L.

2.5. ϵ-transitions on NFA's.

An NFA allows us to 'explore' many paths through a transition diagram simultaneously. However, we are constrained to not change the states that we are 'in' until we read the next letter of the input word. By modifying things slightly and introducing *ϵ-transitions*, we give ourselves an extra degree of flexibility.

Definition 2.16 (ϵ-NFA)**.**
An ϵ-NFA is very similar to an NFA, in that it consists of:

(1) A finite set of *states* Q.
(2) A finite *input alphabet* Σ.
(3) A *transition function* $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ which is total (that is, we now have transitions on the empty word ϵ).
(4) A designated *start state* $q_0 \in Q$.
(5) A finite set of *accept states* $F \subseteq Q$.

We write this as $E = (Q, \Sigma, \delta, q_0, F)$.

Definition 2.17 (Transition diagrams and transition tables for ϵ-NFA's)**.**
We define the transition diagram Γ_E and transition table T_E for the ϵ-NFA E in essentially the same way that we define them for an NFA. The only differences are:

(1) For the transition diagram of an ϵ-NFA, we might have directed edges out of a state labelled with ϵ.
(2) For the transition table of an ϵ-NFA, ϵ is now one of the symbols at the top of the table.

So basically, a ϵ-NFA looks just like an NFA, but with ϵ acting as an extra 'symbol'. It still takes as input words $w \in \Sigma^*$, but processes them in a slightly different way to an NFA. To describe this, we first need the notion of ϵ-closure.

Definition 2.18 (ϵ-closure)**.**
Let $E = (Q, \Sigma, \delta, q_0, F)$ be an ϵ-NFA, and $q \in Q$. We define the *ϵ-closure of q*, eclose(q), to be the set of all states that can be reached from q by sequences of transitions of the form $\delta_E(p, \epsilon)$ (such transitions are called *ϵ-transitions*). That is, we inductively define sets of states $S_i(q)$ by

$$\begin{aligned} S_0(q) &:= \{q\} \\ S_1(q) &:= \{q\} \cup \delta(q, \epsilon) \\ S_{i+1}(q) &:= S_i(q) \cup \Big(\bigcup_{r \in S_i(q)} \delta(r, \epsilon) \Big) \end{aligned}$$

When this series stabilises (that is, when we find an n with $S_{n+1}(q) = S_n(q)$; and it will stabilise as there are only finitely many states in E), then we set

$$\operatorname{eclose}(q) := \bigcup_{i=0}^{\infty} S_i(q) = S_n(q)$$

If $S \subseteq Q$ is an arbitrary set of states, then we define

$$\operatorname{eclose}(S) := \bigcup_{s \in S} \operatorname{eclose}(s)$$

We say that $R \subseteq Q$ is *ϵ-closed* if eclose(R) $= R$.

Lemma 2.19 (ϵ-closure is a closure property)**.**
Let $S \subseteq Q$. Then eclose(S) *is ϵ-closed.*

Proof. We need to show that eclose(eclose(S)) = eclose(S) for any $S \subseteq Q$. So take any $r \in$ eclose(eclose(S)). Then $r \in$ eclose(t) for some $t \in$ eclose(S), and moreover $t \in$ eclose(s) for some $s \in S$. So:
1. We can reach t from s by following a sequence of ϵ-transitions (that is, transitions of the form $\delta_E(p, \epsilon)$).
2. We can reach r from t by following a sequence of ϵ-transitions.

Thus we can reach r from s by ϵ-transitions, and thus $r \in$ eclose(s) $\subseteq$ eclose(S). Our choice of r was arbitrary, so eclose(eclose(S)) $\subseteq$ eclose(S).

Finally, it is clear that eclose(S) $\subseteq$ eclose(eclose(S)), as $s \in$ eclose(s) for any state s.

Thus we have that eclose(eclose(S)) = eclose(S). □

The idea of including ϵ as a symbol to be processed by the transition function is that we can 'explore' all transitions labelled by ϵ without reading the next symbol in the input word. We now define the extended transition function of an ϵ-NFA.

Definition 2.20 (Extended transition fuction for ϵ-NFA)**.**
Let $E = (Q, \Sigma, \delta, q_0, F)$ be an ϵ-NFA. We define the *extended transition function of* E, $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$, inductively. Firstly, for each $q \in Q$, we set

$$\hat{\delta}(q, \epsilon) := \text{eclose}(q)$$

Now, suppose $w = x\sigma \in \Sigma^*$ for some $\sigma \in \Sigma$. Then we do the following:

(1) Let $\{p_1 \ldots, p_k\}$ be $\hat{\delta}(q, x)$.
(2) Let $\{r_1, \ldots, r_m\} = \bigcup_{i=1}^{k} \delta(p_i, \sigma)$.
(3) Define $\hat{\delta}(q, x\sigma) := \bigcup_{j=1}^{m} \text{eclose}(r_j)$.

In particular, the inductive definition looks like this:

$$\hat{\delta}(q, x\sigma) := \bigcup_{r \in \bigcup_{p \in \hat{\delta}(q,x)} \delta(p,\sigma)} \text{eclose}(r)$$

So for any $w \in \Sigma^*$, $\hat{\delta}(q_0, w)$ gives us the set of states we end up at when we input w into E. We say that E *accepts* w if $\hat{\delta}(q_0, w)$ contains at least one state from F.

The above definition is somewhat confusing, so here is a description of what happens when we input a word w into an ϵ-NFA $E = (Q, \Sigma, \delta, q_0, F)$ (and it is best to picture a transition diagram Γ_E for E when doing this):

E first computes eclose(q_0) and places a red marker on all the states in eclose(q_0); that is, places a red marker on all states which can be reached from q_0 by following edges labelled ϵ. Then E takes w, reads the first symbol σ_1 whilst 'in' the set of states eclose(q_0), and then evaluates the transition functions to get a set of states $\bigcup_{q \in \text{eclose}(q_0)} \delta(q, \sigma_1)$; this is done on the transition diagram by following all edges labelled σ_1 out of red-marked states and placing a blue marker on all the new states. E then computes the ϵ-closure of all these states, and simultaneously 'moves to' this closure; so we place extra blue markers on all states which can be reached from the current blue-marked states by following edges labelled ϵ. Now remove all red markers. E then reads the next symbol σ_2 of w, evaluates $\delta(q, \sigma_2)$ for all q in the states that it is currently in, and then takes the ϵ-closure of these new states (this is the same process as before, interchanging the roles of red and blue markers). Once E has read all the symbols in w, we are left with a transition diagram Γ_E with several markers (all of the same colour) on states; if any of these markers lie on a state in F, then E accepts w.

Definition 2.21 (Language of an ϵ-NFA)**.**
Let $E = (Q, \Sigma, \delta, q_0, F)$ be an ϵ-NFA. We define the *language of* E, $\mathcal{L}(E)$, to be the words which are accepted by E. That is,

$$\mathcal{L}(E) := \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

This is the language of words over Σ for which $\hat{\delta}(q_0, w)$ contains at least one accepting state of E.

If we had a transition diagram for an ϵ-NFA E containing no ϵ's (that is, $\delta_E(q, \epsilon) = \emptyset$ for all $q \in Q$), then what we have is an NFA. Thus, all definitions and results on ϵ-NFA's from Section 2.5 carry to NFA's. Thus Section 2.3 is somewhat redundant; we included it only to develop the intuition in a more natural way.

It would seem as though ϵ-NFA's are much more general than NFA's and DFA's. However, we will again see that we do not get any new languages by considering ϵ-NFA's over NFA's or DFA's.

2.6. **Equivalence of DFA's and ϵ-NFA's.**

We show that regular languages are precisely those accepted by an ϵ-NFA. To do this, we describe a process of converting an ϵ-NFA into a DFA, very similar to the subset construction.

Definition 2.22 (The subset construction with ϵ-transitions)**.**
Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ be an ϵ-NFA. We define the following construction of a DFA $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$, called the *subset construction with ϵ-transitions*, to have:

(1) *States* $Q_D := \mathcal{P}(Q_E)$; the power set of Q_E.
(2) *Start state* $q_D :=$ eclose(q_0); the set of states in the ϵ-closure of q_0.
(3) *Accept states* $F_D := \{S \subseteq Q_E \mid S \cap F \neq \emptyset\}$; the set of subsets of Q_E which intersect the accepting states F_E of E.
(4) *Transition function*: for each $S \subseteq Q_E$ and each $\sigma \in \Sigma$, we define $\delta_D(S, \sigma)$ as follows:
a) Let $S = \{p_1, \dots, p_k\}$
b) Let $\bigcup_{i=1}^{k} \delta_E(p_i, \sigma) = \{r_1, \dots, r_m\}$
c) Define $\delta_D(S, \sigma) := \bigcup_{j=1}^{m} \text{eclose}(r_j)$
That is,
$$\delta_D(S,\sigma) := \text{eclose}\,\Big(\bigcup_{p\in S} \delta_E(p,\sigma)\Big)$$
which is the set of states reached from the states in S by going along an edge labelled σ followed by some number of edges labelled ϵ.

Observe that the start state of D constructed above is the ϵ-closure of q_0, rather than the set containing just q_0. Also observe that, if S is ϵ-closed, then so is $\delta_D(S, \sigma)$ (as a subset of Q_E) for every $\sigma \in \Sigma$. Thus, given that our start state of D, eclose(q_0), is ϵ-closed, then we can only ever reach other ϵ-closed sets via the transition function δ_D.

Definition 2.23 (Accessible states of an automaton)**.**
Let A be any finite-state automaton (DFA, NFA, ϵ-NFA). The *accessible* states of A are those which are reachable from the start state by a finite number of applications of the transition function δ_A; that is, states q for which $q = \hat{\delta}_A(q_0, w)$ (or $q \in \hat{\delta}_A(q_0, w)$; whichever is relevant) for some word w. The rest of the states are said to be *inaccessible*.

Thus we see that the accessible states of D in Definition 2.22 must be ϵ-closed, as the start state is.

Theorem 2.24 (Extended transition function in the subset construction with ϵ-transitions)**.**
Let $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ be the DFA constructed from the ϵ-NFA $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ via the subset construction with ϵ-transitions (Definition 2.22*). Then, for any $w \in \Sigma^*$, we have*
$$\hat{\delta}_D(q_D, w) = \hat{\delta}_E(q_0, w)$$

Proof. We proceed by induction on $|w|$. If $|w| = 0$ (that is, $w = \epsilon$), then we have that $\hat{\delta}_E(q_0, \epsilon) = \text{eclose}(q_0)$. But for a DFA, we know that $\hat{\delta}_D(p, \epsilon) = p$ for any state p. Thus $\hat{\delta}_D(q_D, \epsilon) = q_D = \text{eclose}(q_0)$, and so $\hat{\delta}_E(q_0, \epsilon) = \hat{\delta}_D(q_D, \epsilon)$.

Now suppose that $\hat{\delta}_D(q_D, v) = \hat{\delta}_E(q_0, v)$ for all v with $|v| \leq n$. Let w be a word of length $n+1$. Then $w = x\sigma$, where $\sigma \in \Sigma$ is the final symbol of w. By induction, as $|x| = n$, we have that $\hat{\delta}_D(q_D, x) = \hat{\delta}_E(q_0, x)$. Call this set $\{p_1, \ldots, p_k\} \subseteq Q_N$. By definition of $\hat{\delta}$ for ϵ-NFA's (Definition 2.20), we compute $\hat{\delta}_E(q_0, x\sigma)$ by

(1) Let $\{r_1, \ldots, r_m\} = \bigcup_{i=1}^{k} \delta_E(p_i, \sigma)$.
(2) Then $\hat{\delta}_E(q_0, x\sigma) = \bigcup_{j=1}^{m} \text{eclose}(r_j)$.

Now, we know that $\hat{\delta}_D(q_D, x\sigma) = \delta_D(\hat{\delta}_D(q_D, x), \sigma) = \delta_D(\{p_1, \ldots, p_k\}, \sigma)$. So by definition of the subset construction with ϵ-transitions (Definition 2.22), we compute $\hat{\delta}_D(q_D, x\sigma)$ as follows:

(1) Let $\{r_1, \ldots, r_m\} = \bigcup_{i=1}^{k} \delta(p_i, \sigma)$.
(2) Then $\hat{\delta}_D(q_D, x\sigma) = \bigcup_{j=1}^{m} \text{eclose}(r_j)$.

But this is exactly the same set as $\hat{\delta}_E(q_0, x\sigma)$. Thus we have that $\hat{\delta}_D(q_D, w) = \hat{\delta}_E(q_0, w)$, and the induction is complete. □

We can now use this to say something about the accepted languages of these automata.

Theorem 2.25 (Equivalence of language in the subset construction with ϵ-transitions)**.**
Let $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$ be the DFA constructed from the ϵ-NFA $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ via the subset construction with ϵ-transitions. Then $\mathcal{L}(D) = \mathcal{L}(E)$.

Proof. From Theorem 2.24, we see that, for any $w \in \Sigma^*$,

$$\begin{aligned} D \text{ accepts } w &\Leftrightarrow \hat{\delta}_D(q_D, w) \text{ contains a state in } F_E \\ &\Leftrightarrow \hat{\delta}_E(q_0, w) \text{ contains a state in } F_E \\ &\Leftrightarrow E \text{ accepts } w \end{aligned}$$

Thus $\mathcal{L}(D) = \mathcal{L}(E)$, as both D and E have the same alphabet Σ. □

Theorem 2.26 (Equivalence of DFA's and ϵ-NFA's)**.**
A language L is accepted by some DFA if and only if it is accepted by some ϵ-NFA.

Proof. We showed in Theorem 2.25 that, from any ϵ-NFA, we can construct a DFA on the same alphabet which accepts the same language.

So now suppose we have that L is accepted by some DFA $D = (Q, \Sigma, \delta_D, q_0, F)$. Take a transition diagram for D. Then this also defines a transition diagram for an ϵ-NFA E with the same states, same alphabet, and same accepting states, provided we define $\delta_E(q, \epsilon) := \emptyset$ for all $q \in Q$. We need only remark that the transition function δ_E for E will be given by

$$\delta_E(q, \sigma) = \{\delta_D(q, \sigma)\} \ \forall q \in Q, \sigma \in \Sigma$$

That is, if $\delta_D(q, \sigma) = p$, then $\delta_E(q, \sigma) = \{p\}$. Thus the transitions of D and E are the same. Moreover, there are no transitions out of any state on ϵ. Thus E is genuinely an ϵ-NFA. We prove by induction on $|w|$ that, if $\hat{\delta}_D(q_0, w) = p$,

then $\hat{\delta}_E(q_0, w) = \{p\}$:
Basis: Let $|w| = 1$, so $w = \sigma \in \Sigma$. We have defined $\delta_E(q_0, \sigma) = \{\delta_D(q_0, \sigma)\}$, so we're done.
Induction: Suppose, for all v with $|v| \leq n$, we have $\hat{\delta}_E(q_0, v) = \{\hat{\delta}_D(q_0, v)\}$. Take w with $|w| = n+1$, then $w = x\sigma$ for some $x \in \Sigma^*, \sigma \in \Sigma$. Now we have that:

$$\begin{aligned}
\hat{\delta}_E(q_0, x\sigma) &= \bigcup_{p \in \hat{\delta}_E(q_0, x)} \delta_E(p, \sigma) \\
&= \bigcup_{p \in \{\hat{\delta}_D(q_0, x)\}} \delta_E(p, \sigma) \\
&= \delta_E(\hat{\delta}_D(q_0, x), \sigma) \\
&= \{\delta_D(\hat{\delta}_D(q_0, x), \sigma)\} \\
&= \{\hat{\delta}_D(q_0, x\sigma)\}
\end{aligned}$$

Thus D and $E = (Q, \Sigma, \delta_E, \{q_0\}, F)$ accept the same words, so $\mathcal{L}(D) = \mathcal{L}(E)$. □

Again, as all NFA's are ϵ-NFA's, we see that the subset construction in Section 2.4 has now been made redundant, as we need only do things for ϵ-NFA's.

So now we see that, whenever we want to show that a language L is regular, it suffices to produce a DFA *or* an NFA *or* an ϵ-NFA which accepts L. For clarity, we will usually use D to denote a DFA, N to denote an NFA, and E to denote an ϵ-NFA (though we may, on occasions, us other letters also).

2.7. **Regular expressions.**

In the previous sections we saw various 'mechanical' ways of defining languages, all of which define the same set of languages. We now give an algebraic way to define languages, called *regular expressions*. Though seemingly different to our mechanical definitions of DFA's, NFA's, and ϵ-NFA's, we will show that regular expressions define precisely the set of regular languages, hence the name. We first need some algebraic operations on languages.

Definition 2.27 (Operations on languages)**.**

(1) The *union* of two languages L and M, denoted $L \cup M$, is the set of all words which lie in either L or M.
(2) The *concatenation* of two languages L and M, denoted LM, is the set of all words which are the concatenation of one word in L followed by one word in M. We adopt the notation $L^0 := \{\epsilon\}$, $L^1 := L$, $L^{n+1} := L^n L$ (the concatenation of $n+1$ copies of L).
(3) The *closure* of a language L, denoted L^*, is the set of all words formed by taking a finite number of words in L (possibly with repetition), and concatenating them. In particular, this is given by

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

Observe that $\emptyset^* = \{\epsilon\}$, and $\{\epsilon\}^* = \{\epsilon\}$; these are the only two languages whose closure is not infinite.

We will now describe the algebra of regular expressions. If R is a regular expression, then we write $\mathcal{L}(R)$ for the language defined by R.

Definition 2.28 (Regular expressions)**.**
A *regular expression* is any expression built from the following basis set with the following inductive rules:
Basis:

(1) The constants ϵ and $\emptyset$ are regular expressions, denoting the languages $\mathcal{L}(\epsilon) := \{\epsilon\}$ and $\mathcal{L}(\emptyset) := \emptyset$.
(2) If a is a symbol, then $\mathbf{a}$ is a regular expression denoting the language $\mathcal{L}(\mathbf{a}) := \{a\}$. We will always use boldface to denote the expression corresponding to the symbol.
(3) A variable, written as a capital letter such as M, denotes a variable representing any language.

Induction:

(4) If E, F are regular expressions, then $E + F$ is a regular expression denoting the union of their languages; $\mathcal{L}(E + F) := \mathcal{L}(E) \cup \mathcal{L}(F)$.
(5) If E, F are regular expressions, then EF is a regular expression denoting the concatenation of their languages; $\mathcal{L}(EF) := \mathcal{L}(E)\,\mathcal{L}(F)$.
(6) If E is a regular expression, then E^* is a regular expression denoting the closure of its language; $\mathcal{L}(E^*) := \mathcal{L}(E)^*$.
(7) If E is a regular expression, then (E) is a regular expression denoting the same language; $\mathcal{L}((E)) := \mathcal{L}(E)$. This is used to remove ambiguity when writing expressions such as $(E+F)^*$, which is different to $E+F^*$.

Definition 2.29 (Order of precedence)**.** The order of precedence for regular expressions is:

(1) Parentheses ( )
(2) Closure *
(3) Concatenation
(4) Union +

For example, the expression $\mathbf{01}^* + \mathbf{1}$ should be read $(\mathbf{0}(\mathbf{1}^*)) + \mathbf{1}$.

2.8. **Equivalence of DFA's and regular expressions.**

We can now show that regular expressions give us precisely the set of regular languages. First, we show how to construct an ϵ-NFA from a regular expression.

Theorem 2.30 (Constructing an ϵ-NFA from a regular expression)**.**
For each regular expression R there is an associated ϵ-NFA E such that $\mathcal{L}(E) = \mathcal{L}(R)$.

Proof. We will construct an ϵ-NFA E with the following properties:

(1) Exactly one accepting state.
(2) No arcs into the initial state.
(3) No arcs out of the accepting state.

We build up such an ϵ-NFA inductively, in the same way that we defined regular expressions inductively.
Basis:
We construct an ϵ-NFA for each of the basis regular expressions. We do this for ϵ in Figure 1, $\emptyset$ in Figure 2, and $\mathbf{a}$ (for a some symbol) in Figure 3. There is no need for us to label the states, but observe that each ϵ-NFA in Figures 1–3 satisfy each of the three properties listed above.

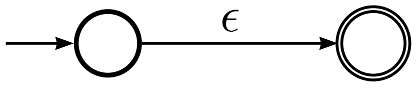

Figure 1: An ϵ-NFA which accepts the language $\{\epsilon\}$.

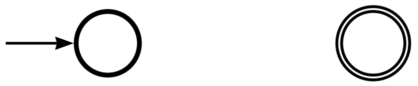

Figure 2: An ϵ-NFA which accepts the language $\emptyset$.

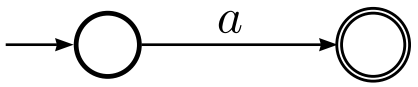

Figure 3: An ϵ-NFA which accepts the language $\{a\}$.

Induction:
We assume that we are given regular expressions R and S with corresponding ϵ-NFA's which satisfy properties $(1), (2), (3)$ above, and now we show how to construct ϵ-NFA's with the same language as $R + S$, RS and R^* respectively. We demonstrate in Figure 4 how we will represent the ϵ-NFA for a regular expression R (which satisfies properties $(1), (2), (3)$ above) within a larger ϵ-NFA. As each ϵ-NFA for our basis regular expressions has precisely one start and one accept state, we take these as the two 'end points' of the ϵ-NFA, and use these to join them into larger ϵ-NFA's. All the ϵ-NFA's that we will now construct also have precisely one start and one accept state, so this 'joining' process will always work.

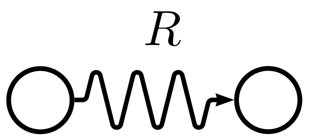

Figure 4: Representing the ϵ-NFA for the regular expression R within a larger ϵ-NFA.

An ϵ-NFA for $R + S$:
The ϵ-NFA E in Figure 5 gives precisely the same language as $R + S$. First, take a word $w \in \mathcal{L}(R) \cup \mathcal{L}(S)$. If we start at the start state of E, then we can simultaneously ϵ-transit to the beginning of the ϵ-NFA for R, and the beginning of the ϵ-NFA for S. As w lies in the language of either R or S, then after reading all of w we will eventually reach the accept state of that respective ϵ-NFA (or both). Then we can ϵ-transit to the accept state of E.
Conversely, suppose w is accepted by E. Then, starting from the start state of E, we ϵ-transit to the start state of the ϵ-NFA's of R and S, simultaneously. Then we will reach the accept state of one of the ϵ-NFA's of R and S, and in doing so we must read all of w. There are no more symbols of w to read, and so we take the final ϵ-transition to the accept state of E, and so w is accepted by E.
Thus the language of E is $\mathcal{L}(R) \cup \mathcal{L}(S)$.

An ϵ-NFA for RS:
The ϵ-NFA E in Figure 6 gives precisely the same language as RS. First, take a word $w \in \mathcal{L}(R)\,\mathcal{L}(S)$, so $w = uv$ for some $u \in \mathcal{L}(R)$, $v \in \mathcal{L}(S)$. If we start at the start state of E, then we can ϵ-transit to the start state of the ϵ-NFA for R. After reading all of u we will eventually reach the accept state of the ϵ-NFA for R. Then we can ϵ-transit to the start state of the ϵ-NFA for S, with v still to read. But after reading all of v we will eventually reach the accept state of the ϵ-NFA for S, which is the accept state of E.
Conversely, suppose w is accepted by E. Then, starting from the start state of N, we ϵ-transit to the start state of the ϵ-NFA for R. Then, to reach the accept state of this, we first have that w has some prefix $u \in \mathcal{L}(R)$ which we read and get to the accept state of R. Then we ϵ-transit to the start state of the ϵ-NFA for S. But to reach the accept state of the ϵ-NFA for S (that is, the accept state of N), we must have that the entire remaining suffix v of w lies in $\mathcal{L}(S)$. That is, $w = uv$ for some $u \in \mathcal{L}(R)$, $v \in \mathcal{L}(S)$.
Thus the language of E is $\mathcal{L}(R)\,\mathcal{L}(S)$.

An ϵ-NFA for R^*:
The ϵ-NFA N in Figure 7 gives precisely the same language as R^*. First, take a word $w \in \mathcal{L}(R^*)$. If $w = \epsilon$ then we can ϵ-transit directly to the accept state of E. If $w \neq \epsilon$, then we have that $w = u_1 \dots u_m$ for some $m > 0$ and some collection of non-empty $u_i \in R$. So start at the start state of E, then ϵ-transit to start state of the ϵ-NFA for R. After reading all of u_1 we will eventually reach the accept state of the ϵ-NFA for R. Then we can ϵ-transit back to the start state of the ϵ-NFA for R, with $u_2 \dots u_m$ still to read. Repeat this process a total of m times; then we're left at the accept state of the ϵ-NFA for R with nothing left to read, and thus can ϵ-transit to the accept state of E.
Conversely, suppose w is accepted by E. Then, starting from the start state of E, we either ϵ-transit to the accept state of E and are accepted (thus giving $w = \epsilon \in \mathcal{L}(R^*)$), or we ϵ-transit to the start state of the ϵ-NFA for R. Then, to reach the accept state of the ϵ-NFA for R, we must have that w has some prefix $u_1 \in \mathcal{L}(R)$ (so $w = u_1 v$). So we read such a prefix, and then reach the accept state of the ϵ-NFA for R with the remaining suffix left to read. If this suffix v is empty we are done (we ϵ-transit to the accept state of E and are accepted). Otherwise, the only other option is to ϵ-transit back to the start state of the

ϵ-NFA for R, with v still to read. We keep repeating this process, and the only way for w to be accepted by E is if, after m such cycles, we have reached the accept state of the ϵ-NFA for R and have nothing left to read (so that we can ϵ-transit to the accept state of E), and thus $w = u_1 \dots u_m$ for $u_i \in \mathcal{L}(R)$, and so $w \in \mathcal{L}(R^*)$.
Thus the language of E is $\mathcal{L}(R^*)$.

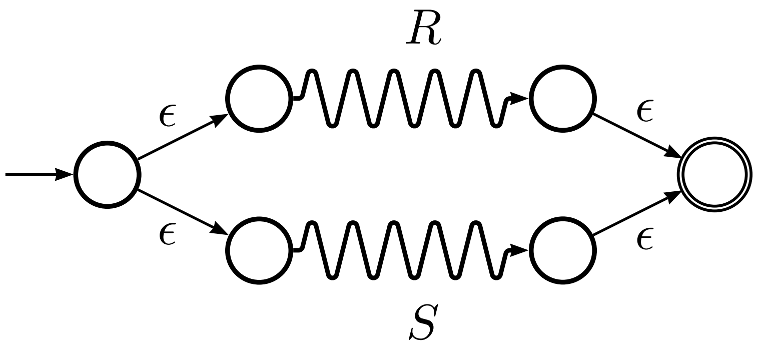

Figure 5: An ϵ-NFA which accepts the language $\mathcal{L}(R) + \mathcal{L}(S)$.

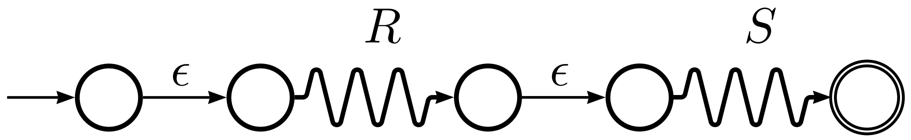

Figure 6: An ϵ-NFA which accepts the language $\mathcal{L}(R)\,\mathcal{L}(S)$.

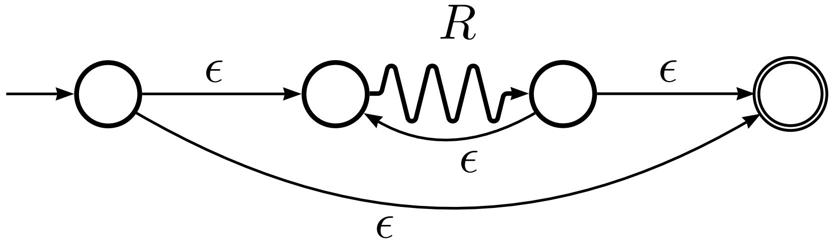

Figure 7: An ϵ-NFA which accepts the language $\mathcal{L}(R)^*$.

All the ϵ-NFA's we constructed in Figures 5–7 satisfy properties $(1), (2), (3)$ (assuming the ϵ-NFA's for R and S also satisfy these). Thus our induction is complete; we can build up an ϵ-NFA for any regular expression. $\square$

We now show how to construct a regular expression from a DFA.

Theorem 2.31 (Constructing a regular expression from a DFA)**.**
For each DFA D there is an associated regular expression R such that $\mathcal{L}(D) = \mathcal{L}(R)$.

Proof. For each DFA $D = (Q, \Sigma, \delta, q_0, F)$, we construct a regular expression R as follows:
First we number the states of D by $\{1, \ldots, n\}$ (this re-naming will be helpful later). Now we will inductively define regular expressions $R_{ij}^{(k)}$ $(1 \leq i, j \leq n, 0 \leq k \leq n)$ whose languages are the words w which begin at state i and end at state j without passing through any intermediate state whose number is more than k (think of w as giving a path in a transition diagram for D). Note that the beginning and end points are not intermediate, so we may have i and/or j being greater than k.

We first consider the case $k = 0$. Note that every state is numbered greater than 0, so $R_{ij}^{(0)}$ is only paths from i to j with no intermediate states. That is, direct edges from i to j. If $i \neq j$, then this will be single edges from i to j. If $i = j$, then this is the path of length 0 (that is, ϵ), as well as all loops from i to itself. So let $\{a_1, \ldots, a_l\}$ be all the symbols labelling arcs from i to j (that is, all the symbols $a \in \Sigma$ for which $\delta(i, a) = j$). Then we define

$$\text{For } i \neq j,\ R_{ij}^{(0)} := \begin{cases} \mathbf{a_1} + \ldots + \mathbf{a_l} & \text{if } l > 0 \\ \emptyset & \text{if } l = 0 \end{cases}$$

$$\text{For } i = j,\ R_{ii}^{(0)} := \begin{cases} \mathbf{a_1} + \ldots + \mathbf{a_l} + \epsilon & \text{if } l > 0 \\ \epsilon & \text{if } l = 0 \end{cases}$$

We have added the ϵ in the case $i = j$ to cover the situation where we have the path of length 0 from i to itself. Thus we have a regular expression for $R_{ij}^{(0)}$ which gives us the language we desire.

Now we must inductively define the expression $R_{ij}^{(k)}$, assuming we have defined $R_{ij}^{(t)}$ for all $t < k$ and all i, j. So, suppose we have a path from state i to state j that does not pass through any intermediate state higher than k. This falls in to one of two cases:
Case 1. The path does not pass through state k at all, in which case the path gives a word which lies in the language of $R_{ij}^{(k-1)}$.
Case 2. The path passes through state k (as an intermediate state) some number of times. In this case, we can break up the path into three pieces:
Piece 1. A (nonempty) path from state i to state k that does not pass through state k in any intermediate step. This will lie in the language of $R_{ik}^{(k-1)}$.
Piece 2. A (possibly empty) path from state k to state k which does not pass through state k in any intermediate step, followed by another such path, and so on (finitely many times). These sub-pieces will lie in the language of $R_{kk}^{(k-1)}$, and so the entire piece will be concatenations of these, and thus lie in the language of $\big(R_{kk}^{(k-1)}\big)^*$.
Piece 3. A (nonempty) path from state k to state j that does not pass through state k in any intermediate step. This will lie in the language of $R_{kj}^{(k-1)}$.
So in case 2, our word lies in the concatenation of the languages from piece 1, then piece 2, then piece 3. That is, it lies in the language of $R_{ik}^{(k-1)}\big(R_{kk}^{(k-1)}\big)^* R_{kj}^{(k-1)}$.
Adding the regular expressions from the two cases (that is, taking the union of the two languages), we have the regular expression $R_{ij}^{(k-1)} + R_{ik}^{(k-1)}\big(R_{kk}^{(k-1)}\big)^* R_{kj}^{(k-1)}$

whose language contains all words w which begin at state i and end at state j without passing through any intermediate state whose number is more than k. But clearly this defines all such words, as its language is no more than these words. Thus we define the regular expression

$$R_{ij}^{(k)} := R_{ij}^{(k-1)} + R_{ik}^{(k-1)}\big(R_{kk}^{(k-1)}\big)^* R_{kj}^{(k-1)}$$

for the language of all words representing paths from state i to state j which do not pass through any intermediate state higher than k.

Now, given that we only have n states in total, we see that the language of $R_{ij}^{(n)}$ will be all words representing paths which start at state i and end at state j. Now assume that state 1 is the start state (our initial numbering was arbitrary, so we could have easily defined it that way). Take the sum of all expressions of the form $R_{1j}^{(n)}$ for j an accepting state, and the associated language will be the language of D. That is, we have just proved that

$$\mathcal{L}(D) = \mathcal{L}\left(R_{1j_1}^{(n)} + \ldots + R_{1j_s}^{(n)}\right) \text{ where } F = \{j_1, \ldots, j_s\}$$

□

So now we see that, whenever we want to show that a language L is regular, it suffices to produce a DFA *or* an NFA *or* an ϵ-NFA *or* a regular expression whose associated language is L. For clarity, we will usually use D to denote a DFA, N to denote an NFA, E to denote an ϵ-NFA, and R to denote a regular expression (though we may, on occasions, us other letters).

Having all these equivalent definitions at our disposal may seem confusing at first. But ultimately it is quite helpful, as there are various languages which are very easily shown to be regular with one definition, but not the others. Also, keep in mind that it is not always obvious which definition will be the easiest to use to show that a language is regular.

2.9. **Closure properties of regular languages.**

We will now use the various definitions of regular languages to show some closure properties.

Theorem 2.32 (Closure under union)**.**
Let L, M be regular languages over Σ, Γ respectively. Then $L \cup M$ is a regular language over $\Sigma \cup \Gamma$.

Proof. Let R_L, R_M be regular expressions for L, M respectively. Then $R_L + R_M$ is a regular expression, and $\mathcal{L}\left(R_L + R_M\right) = \mathcal{L}(R_L) \cup \mathcal{L}(R_M) = L \cup M$. □

Theorem 2.33 (Closure under complementation)**.**
Let L be a regular language over Σ. Then the complement of L, $\overline{L} := \Sigma^* \setminus L$, is a regular language over Σ.

Proof. Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA such that $L = \mathcal{L}(D)$. Now define a new DFA $\overline{D} := (Q, \Sigma, \delta, q_0, Q \setminus F)$. Notice that, for any $w \in \Sigma^*$, we have

$$\begin{aligned} w \in \overline{L} &\Leftrightarrow w \in \Sigma^* \setminus L \\ &\Leftrightarrow w \notin L \\ &\Leftrightarrow \hat{\delta}(q_0, w) \notin F \\ &\Leftrightarrow \hat{\delta}(q_0, w) \in Q \setminus F \\ &\Leftrightarrow w \in \mathcal{L}(\overline{D}) \end{aligned}$$

□

Observe that the above proof only works because D and $\overline{D}$ are deterministic (if not, then we could not conclude that $\hat{\delta}(q_0, w) \notin F \Leftrightarrow \hat{\delta}(q_0, w) \in Q \setminus F$). Thus the above argument would not work (as written) if we were using an NFA to describe L. This is an example of where one description of regular languages is more convenient than another.

Theorem 2.34 (Closure under concatenation)**.**
Let L, M be regular languages over Σ, Γ respectively. Then LM is a regular language over $\Sigma \cup \Gamma$.

Proof. Let R_L, R_M be regular expressions for L, M respectively. Then $R_L R_M$ is a regular expression, and $\mathcal{L}\left(R_L + R_M\right) = \mathcal{L}(R_L)\,\mathcal{L}(R_M) = LM$. □

Theorem 2.35 (Closure under closure operator)**.**
Let L be a regular language over Σ. Then L^* is a regular language over Σ.

Proof. Let R be a regular expression for L. Then $(R)^*$ is a regular expression, and $\mathcal{L}\left((R)^*\right) = \mathcal{L}(R)^* = L^*$. □

Theorem 2.36 (Closure under intersection)**.**
Let L, M be regular languages over Σ, Γ respectively. Then $L \cap M$ is a regular language over $\Sigma \cap \Gamma$.

Proof. First observe that $\overline{L} = \Sigma^* \setminus L$ and $\overline{M} = \Gamma^* \setminus M$ are both regular over Σ, Γ respectively (by Theorem 2.33). So $\overline{L} \cup \overline{M}$ is regular over $(\Sigma \cup \Gamma)^*$ (by Theorem 2.32). Thus $\overline{\overline{L} \cup \overline{M}}$ is also regular over $(\Sigma \cup \Gamma)^*$ (by Theorem 2.33). But by DeMorgan's laws, we have that $L \cap M = \overline{\overline{L} \cup \overline{M}}$ (taking the final complement in $(\Sigma \cup \Gamma)^*$), so $L \cap M$ is regular over $\Sigma \cup \Gamma$. But clearly $L \cap M \subseteq \Sigma^* \cap \Gamma^* = (\Sigma \cap \Gamma)^*$, so $L \cap M$ is a regular language over $\Sigma \cap \Gamma$. □

Definition 2.37. Let $\sigma_1 \ldots \sigma_m \in \Sigma^*$ be some word. We define its *reverse* by

$$(\sigma_1 \ldots \sigma_m)^R := \sigma_m \ldots \sigma_1$$

(that is, the word $\sigma_1 \ldots \sigma_m$ written in reverse). If L is a language, we define the *reverse* of L, L^R, to be

$$L^R := \{v^R \mid v \in L\}$$

(that is, the language consisting of the reverses of all the words in L).

Theorem 2.38. *Let L be a regular language. Then L^R is also regular.*

Proof. Take a DFA $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$ for which $\mathcal{L}(D) = L$. From this we construct a 'reverse ϵ-NFA' E from D with

(1) *States* $Q_E := Q_D \cup \{z\}$ (same as D, with one extra state z added).
(2) *Alphabet* Σ (same as D).
(3) *Start state* z.
(4) *Accept states* $F_E := \{q_0\}$; the start state of D.
(5) *Transition function*: $\delta_E : Q_E \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q_E)$ given by

$$\delta_E(q, \epsilon) := \begin{cases} F_D & \text{if } q = z \\ \emptyset & \text{if } q \neq z \end{cases}$$

$$\delta_E(q, \sigma) := \{p \in Q_D \mid \delta_D(p, \sigma) = q\}$$

That is, we reverse all existing transitions in D, then add a new state z, and finally add ϵ-transitions from z to all states in F. Thus $w \in \mathcal{L}(D) \Leftrightarrow w^R \in \mathcal{L}(E)$, so L^R is regular. □

2.10. **The pumping lemma and non-regular languages.**

Now that we have a long list of ways to show that languages are regular, it is time to develop some more tools, this time to show that certain languages are not regular.

Suppose we have a DFA $D = (Q, \Sigma, \delta, q_0, F)$, and a word w which is accepted by D. Suppose moreover that $|w| > |Q|$, that is, w has more symbols than D has states. Consider the transition diagram Γ_D of D. By the pigeonhole principle, we must have that w defines a path in Γ_D which visits the same state twice. So we can break up w into 3 subwords $w = xyz$, with y defining a non-empty path from some state q back to itself. So y gives us a 'loop' in the transition diagram. Now consider what would happen if we were to remove y from w, to have the word xz. Or if we were to 'do y again', to have the word $xyyz$. Would these new words be accepted by D?

Theorem 2.39 (The pumping lemma for regular languages)**.**
Let L be a regular language. Then there exists a constant n (depending on L) such that for every word $w \in L$ with $|w| \geq n$ we can break up w into 3 *words $w = xyz$ such that:*

(1) $y \neq \epsilon$.
(2) $|xy| \leq n$.
(3) *For all $k \geq 0$, we have that the word xy^kz is also in L.*

This theorem is named the *pumping lemma* because for each word w of sufficient length we can find a subword y which we can *pump*; that is, we can repeat y as many times as we like.

Proof. As L is regular, we have that $L = \mathcal{L}(D)$ for some DFA $D = (Q, \Sigma, \delta, q_0, F)$. Suppose that D has n states ($|Q| = n$). Now, take any accepted word $w \in L$ with $|w| \geq n$, say $w = \sigma_1 \dots \sigma_m$; $m \geq n$, where each $\sigma_j \in \Sigma$. Let p_i be the state that D is in after reading the subword $\sigma_1 \dots \sigma_i$ ($1 \leq i \leq m$). That is, $p_i := \hat{\delta}(q_0, \sigma_1 \dots \sigma_i)$. Define $p_0 := \hat{\delta}(q_0, \epsilon) = q_0$.

By the pigeonhole principle, the $n+1$ p_i's $\{p_0, \dots, p_n\}$ must have some repeated state, as there are only n different states of D. So we can find two integers $0 \leq l < r \leq n$ with $p_l = p_r$. So we now break up w as $w = xyz$, where

$$\begin{aligned} x &= \sigma_1 \dots \sigma_l \quad (\text{or } \epsilon \text{ if } l = 0) \\ y &= \sigma_{l+1} \dots \sigma_r \quad (\text{and so } |xy| \leq n \text{ and } y \neq \epsilon) \\ z &= \sigma_{r+1} \dots \sigma_m \quad (\text{or } \epsilon \text{ if } r = n) \end{aligned}$$

That is, x traces a path from p_0 to p_l, y traces a loop from p_l back to itself (as $p_l = p_r$), and z takes us to some accepting state q_t as w is an accepted word. Note that x and/or z are permitted to be empty, but by definition we have that y is non-empty (as $l < r$). Now consider what happens when we input xy^kz into D:
If $k = 0$, then we go from q_0 (which is p_0) to p_l on a path traced by x. We then go from p_r (which is p_l) to the accepting state q_t (which is p_m) on a path traced by z. Thus xz traces a path from q_0 to the accept state q_t (the same accept state that w ends at), and so $xz \in L$.

If $k \geq 1$, then we go from q_0 (which is p_0) to p_l on a path traced by x. We then loop from p_l back to itself k times on a path traced by y (recalling that $p_l = p_r$). We then go from p_r to the accepting state q_t (which is p_m) on a path traced by z. Thus xy^kz traces a path from q_0 to the accept state q_t (the same accept state that w ends at), and so $xy^kz \in L$. □

Here is a typical way of using the pumping lemma to show that a language is not regular:

Example 2.40. *The language $L = \{0^n1^n \mid n \geq 1\}$, of all words consisting of some number of 0's followed by the same number of 1's, is not regular.*

Proof. Suppose L were regular; we proceed by contradiction using the pumping lemma. If L were regular, then we would have some constant N satisfying the hypotheses of the pumping lemma. So consider the word $w = 0^N1^N$. Then $w \in L$. Moreover, by the pumping lemma, we can break up $w = xyz$ such that:
1. $y \neq \epsilon$.
2. $|xy| \leq N$.
3. For all $k \geq 0$, we have that the word xy^kz is also in L.

As $|xy| \leq N$, we must have that $xy = 0^m$ for some $m \leq N$ (as the first N symbols in w are all 0's), and moreover that $y = 0^l$ for some $0 < l \leq m \leq N$. Thus, by the pumping lemma, we must have that $xz \in L$. But $xz = 0^{N-l}1^N$, which is not in L as $N - l \neq N$. □

We could have also argued that, by the pumping lemma, we must have that $xy^2z \in L$. But $xy^2z = 0^{N+l}1^L$, which is again not in L.

The pumping lemma will be the usual tool we will use to show certain languages are not regular. The standard technique for this is:

(1) Take a language L and assume that it is regular.
(2) Suppose there is an N which satisfies the hypothesis of the pumping lemma.
(3) Choose a word w in L and suppose it has a decomposition $w = xyz$ as per the pumping lemma.
(4) For any such decomposition of w as above, show that there is a suitable power $k \geq 0$ of y such that $xy^kz \notin L$.

2.11. **Equivalence relations and minimisation of DFA's.**

We now describe a way to find a *minimal* version of any given DFA D; that is, a DFA D' which accepts the same language but has the smallest possible number of states. The underlying idea is that we take our original DFA D and group together states which are *equivalent.*

Definition 2.41 (State equivalence in DFA's)**.**
Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We call two states $p, q \in Q$ *equivalent* or *indistinguishable* if, for all $w \in \Sigma^*$, we have that $\hat{\delta}(p, w) \in F$ if and only if $\hat{\delta}(q, w) \in F$. We write this as $p \sim q$. That is,

$$p \sim q \text{ if and only if } \forall w \in \Sigma^*(\hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F)$$

If two states p, q are not equivalent, then we say they are *distinguishable*, and we say that they are *distinguished by w* if one of $\hat{\delta}(p, w)$, $\hat{\delta}(q, w)$ is accepting but the other is not.

We do not require $\hat{\delta}(p, w) = \hat{\delta}(q, w)$ for all words w in order for p, q to be equivalent, just that they are always either both accepting or both non-accepting.

Lemma 2.42.
The relation $\sim$ on states given in Definition 2.41 is an equivalence relation. In other words, it is:

1. Reflexive*: $p \sim p$ $\forall p \in Q$*
2. Symmetric*: if $p \sim q$ then $q \sim p$*
3. Transitive*: if $p \sim q$ and $q \sim r$, then $p \sim r$.*

Thus we can define the equivalence class *of a state p, written $[p]$, as all the states equivalent to p. That is,*

$$[p] := \{q \in Q \mid q \sim p\}$$

This gives a partition of Q into disjoint equivalence classes.

Proof. 1 and 2 are immediate.
To prove 3: Let w be a word on the alphabet of the DFA. If $p \sim q$ then $\hat{\delta}(p, w), \hat{\delta}(q, w)$ are either both accepting or both non-accepting. Similarly, if $q \sim r$ then $\hat{\delta}(q, w), \hat{\delta}(r, w)$ are either both accepting or both non-accepting. Thus $\hat{\delta}(p, w), \hat{\delta}(r, w)$ are either both accepting or both non-accepting, and so $p \sim r$. □

Definition 2.43 (The table-filling algorithm for DFA's)**.**
We describe the following algorithm, called the *table-filling algorithm*, for determining which states of a DFA $D = (Q, \Sigma, \delta, q_0, F)$ are distinguishable.

We begin by drawing a table T, with the rows and columns indexed by elements of Q. The point is to mark, in entry with coordinates (p, q), whether p and q are distinguishable or not. As this is a reflexive property, we need only consider the lower-left triangle of the table. We start with all entries being 'empty', and mark later entries by the following inductive process.
Basis:
We place a mark 'x' in every entry labelled by a pair of states (p, q) with one of p, q accepting and the other non-accepting. These states are distinguishable; ϵ will distinguish them.
Inductive step:
Take the table T at the current point in the algorithm. Take any pair of states p, q where entry (p, q) is unmarked. If there is some symbol σ with $\delta(p, \sigma) = r$ and $\delta(q, \sigma) = s$ with (r, s) already marked in T (which corresponds to r, s being distinguished states in D), then we know that p and q are distinguished states. This is because r, s are distinguished (say by w), and thus precisely one of $\hat{\delta}(r, w), \hat{\delta}(s, w)$ is accepting. But $\hat{\delta}(p, \sigma w) = \hat{\delta}(r, w)$, and $\hat{\delta}(q, \sigma w) = \hat{\delta}(s, w)$, and thus σw distinguishes p and q. So we place a new mark 'x' at entry (p, q). Now repeat the inductive step.
Conclusion:
If we have filled the table T sufficiently so that, for every pair of states p, q where entry (p, q) is unmarked, there is no symbol σ with $\delta(p, \sigma) = r$ and $\delta(q, \sigma) = s$ with (r, s) already marked in T, then our algorithm halts.

Theorem 2.44 (Proof of the table-filling algorithm)**.**
Let D be a DFA. Then two states p, q correspond to a marked entry in the table-filling algorithm if and only if they are distinguished states in D.

Proof. Let $D = (Q, \Sigma, \delta, q_0, F)$, and T the table at the end of the table-filling algorithm. Clearly, if entry (p, q) is marked in T, then p and q are distinguished states in D. This is because through the algorithm we can inductively construct a distinguishing word for p and q.

We call a pair of states p, q a *bad pair* if (p, q) is unmarked in T but p, q are distinguished states in D. We will proceed by contradiction to show there are no bad pairs. So, assume a bad pair exists. Now, over all possible bad pairs, take one such pair p, q with the shortest possible distinguishing element $w = \sigma_1 \dots \sigma_n$ (that is, if p', q' are a bad pair distinguished by word w', then $|w'| \geq |w|$). So precisely one of $\hat{\delta}(p, w), \hat{\delta}(q, w)$ is accepting.

Clearly, w is not ϵ, for if it were then (p, q) would be marked in the basis step of the table-filling algorithm, and thus not be a bad pair. So $n \geq 1$.

Now consider the states $r = \delta(p, \sigma_1)$ and $s = \delta(q, \sigma_1)$. Thus r, s are distinguished by the word $\sigma_2 \dots \sigma_n$ which is of length $n-1$, and so by the minimality of n we have that r, s is not a bad pair. Thus the entry (r, s) is marked in T. But once entry (r, s) is marked, the table-filling algorithm will then eventually mark entry (p, q), as $\delta(p, \sigma_1) = r$ and $\delta(q, \sigma_2) = s$, with entry (r, s) already marked.

Thus there are no bad pairs, and so every pair of states corresponding to an unmarked entry in T are actually indistinguishable in D. □

Part of the reason for studying equivalence of states is to take a DFA D and 'group together' the equivalent state, to construct a new DFA with fewer states but which accepts the same language. We do that now.

Lemma 2.45. *Let p, q be two equivalent states of a DFA $D = (Q, \Sigma, \delta, q_0, F)$, and take any $\sigma \in \Sigma$. Then $\delta(p, \sigma), \delta(q, \sigma)$ are equivalent.*

Proof. Suppose $\delta(p, \sigma) \nsim \delta(q, \sigma)$. Then they are distinguished by some word w. But then p, q would be distinguished by the word σw; a contradiction since $p \sim q$. □

So we see that if we start with any two equivalent states p, q, then the transition function δ must take them to equivalent states for every symbol $\sigma \in \Sigma$.

Lemma 2.46. *Let p, q be two equivalent states of a DFA $D = (Q, \Sigma, \delta, q_0, F)$. Then p is accepting if and only if q is accepting ($p \in F \Leftrightarrow q \in F$).*

Proof. If one of p, q were accepting but the other not, then the word ϵ would distinguish them. □

Definition 2.47 (DFA minimisation).
Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, we define the *minimal DFA for D*, written $D/\sim$, as the DFA with:

(1) *Alphabet* Σ (the same as D).
(2) *States* $Q' := \{[p] \mid p \in Q\}$.
(3) *Transition function* δ' defined by $\delta'([p], \sigma) := [\delta(p, \sigma)]$.
(4) *Start state* $q_0' := [q_0]$.
(5) *Accepting states* $F' := \{[p] \mid p \in F\}$.

By Lemma 2.45, we have that δ' is indeed well-defined. That is, to decide where to send the state $[p]$ when reading symbol σ, we just need to pick out one state in $[p]$ (p itself will suffice), and see which equivalence class $\delta(q, \sigma)$ lies in.

Lemma 2.45 ensures that this always gives us precisely one equivalence class: $[\delta(p, \sigma)]$.

Observe that, by Lemma 2.46, the equivalence relation $\sim$ partitions F into disjoint sets which do not intersect $Q \setminus F$. So the states of $D/\sim$ consist of collections of (equivalent) states which are either all accepting or all non-accepting.

Lemma 2.48. *Let w be a word in a DFA D, q a state in D, and take $D/\sim$ as constructed above. Then*

$$\hat{\delta}'([p], w) = [\hat{\delta}(p, w)]$$

Proof. We induct on $|w|$. Clearly, if $|w| = 0$ (i.e., $w = \epsilon$), then we have $\hat{\delta}'([p], \epsilon) = [p] = [\hat{\delta}(p, \epsilon)]$.
Now suppose $\hat{\delta}'([p], v) = [\hat{\delta}(p, v)]$ for all v with $|v| \leq n$, for some fixed n. Take w with $|w| = n + 1$. Then $w = u\sigma$ for some u with $|u| = n$. So

$$\begin{aligned}
\hat{\delta}'([p], u\sigma) &= \delta'(\hat{\delta}'([p], u), \sigma) \text{ (definition of } \hat{\delta}') \\
&= \delta'([\hat{\delta}(p, u)], \sigma) \text{ (induction, as } |u| = n) \\
&= [\delta(\hat{\delta}(p, u)], \sigma)] \text{ (definition of } \delta') \\
&= [\hat{\delta}(p, u\sigma)] \text{ (definition of } \hat{\delta})
\end{aligned}$$

□

Thus the extended transition function $\hat{\delta}'$ works as a natural extension of the extended transition function $\hat{\delta}$.

Theorem 2.49 (Equivalence of languages of minimal DFA's)**.**
Let D be a DFA, and $D/\sim$ be its minimal DFA. Then $\mathcal{L}(D/\sim) = \mathcal{L}(D)$.

Proof. Observe that D and $D/\sim$ have the same alphabet (call it Σ). So take any word $w \in \Sigma^*$. Then:

$$\begin{aligned}
w \in \mathcal{L}(D/\sim) &\Leftrightarrow \hat{\delta}'(q_0', w) \in F' \text{ (definition of acceptance)} \\
&\Leftrightarrow \hat{\delta}'([q_0], w) \in F' \text{ (definition of } q_0') \\
&\Leftrightarrow [\hat{\delta}(q_0, w)] \in F' \text{ (Lemma 2.48)} \\
&\Leftrightarrow \hat{\delta}(q_0, w) \in F \text{ (Lemma 2.46)} \\
&\Leftrightarrow w \in \mathcal{L}(D) \text{ (definition of acceptance)}
\end{aligned}$$

□

Lemma 2.50. *Let D be a DFA, and $D' = D/\sim$ be its minimal DFA. Then no two states of $D/\sim$ are equivalent.*

Proof. Suppose $[p] \sim [q]$ in D'. Then, for every word w, we have that $\hat{\delta}'([p], w)$, $\hat{\delta}'([q], w)$ are either both accepting or both non-accepting in D'. Thus, $[\hat{\delta}(p, w)]$, $[\hat{\delta}(q, w)]$ are either both accepting or both non-accepting in D' (Lemma 2.48). So then $\hat{\delta}(p, w), \hat{\delta}(q, w)$ are either both accepting or both non-accepting in D (Lemma 2.46). So $p \sim q$ in D, and thus $[p] = [q]$ in D'. □

Definition 2.51. Let A, B be DFA's. We say A and B are *equivalent*, written $A \equiv B$, if, up to possible re-labelling of states, they have same alphabet, transition function, set of states, start state, and accept states.

So equivalent DFA's are 'functionally' identical; they only differ on state names (which are only arbitrary labels). Given that we can count how many states there are in a DFA, and moreover that there are only finitely many ways to permute state names, we see that we can algorithmically compute whether two DFA's are equivalent or not.

Corollary 2.52. *Let D be a DFA. Then $(D/\sim)/\sim \equiv D/\sim$ as DFA's.*

Proof. Performing DFA minimisation on a DFA gives a new DFA whose states are a partitioning of the states of the old DFA into disjoint equivalence classes. But no two distinct states in $D/\sim$ are equivalent (Lemma 2.50). So performing DFA minimisation on $D/\sim$ partitions its states into sets of size 1, which gives the exact same set of states (up to re-labelling). Thus the alphabet, transition function, and states remain unchanged (modulo this state re-labelling). □

Theorem 2.53 (Removing inaccessible states from a DFA)**.**
There is an algorithm which takes a DFA $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$ and produces a DFA $A = (Q_A, \Sigma, \delta_A, q_0, F_A)$ with no inaccessible states, for which $\mathcal{L}(A) = \mathcal{L}(D)$.

Proof. Let $n = |Q_D|$. Then form the sequence S_i of subsets of Q_D, $1 \leq i \leq n$, via

$$S_0 := \{q_0\}$$
$$S_{i+1} := \bigcup_{q \in S_i} \big(\bigcup_{\sigma \in \Sigma} \delta_D(q, \sigma) \big)$$

So S_{i+1} is the set of states which can be reached from S_i with one transition step. At most n steps are needed to reach any state, and so S_n will be the set of all accessible states of D. So define $Q_A := Q_D \cap S_n$, $F_A := F_D \cap S_n$, and δ_A as the restriction of δ_D to $Q_A \times \Sigma$. □

We now prove that minimal DFA's uniquely define regular languages, up to re-naming of states.

Theorem 2.54 (Minimality of minimal DFA's)**.**
Let D be a DFA with no inaccessible states, and suppose that A is another DFA on the same alphabet as D and for which $\mathcal{L}(D) = \mathcal{L}(A)$. Then A has at least as many states as $D/\sim$.
Moreover, if A has the same number of states as $D/\sim$, then $A \equiv D/\sim$.

Proof. Suppose that A has fewer states than $D' := D/\sim$; we proceed by contradiction. Take some DFA B on the same alphabet as D with the least number of states for which $\mathcal{L}(B) = \mathcal{L}(D)$. Now form the 'disjoint union' of the DFA's D', B as follows: With $B = (S, \Sigma, \delta, p_0, G)$, and $D' = (Q', \Sigma, \delta', q_0', F')$ respectively, form a new DFA $U := (Q' \sqcup \overline{S}, \Sigma, \rho, q_0', \overline{G} \sqcup F')$, where the transition function ρ is defined via δ and δ'. (We need to mark the states of B by an overline, to ensure that they are disjoint from those of D', so that $Q' \sqcup \overline{S}$ are all unique.) Picture this as taking the transition diagrams for D', B and drawing them next to each other to get a new transition diagram (we take q_0' as our start state, but it doesn't really matter).

Now run the table-filling algorithm on this disjoint union U, and observe that the start states of D' and B are equivalent as $\mathcal{L}(D') = \mathcal{L}(D) = \mathcal{L}(B)$ (Theorem 2.49). Observe that if p, q are equivalent states (in any DFA), then by Lemma

2.45 their successors $\delta(p,\sigma), \delta(q,\sigma)$ are also equivalent for any symbol σ. By induction, $\hat{\delta}(p,w), \hat{\delta}(q,w)$ are also equivalent for any word w.

Since D has no inaccessible states, then neither does D'. To see this, take a state $[q]$ in D'; there is some word w with $\hat{\delta}_D(q_0,w) = q$ as D has no inaccessible states, and thus $\hat{\delta}_{D'}([q_0],w) = [\hat{\delta}_D(q_0,w)] = [q]$. Moreover, B has no inaccessible states, or else we could form a new DFA C with the inaccessible states of B removed, for which $\mathcal{L}(C) = \mathcal{L}(B)$ (Theorem 2.53); this would contradict the minimality of B.

So far we have:
- The start states of D' and B are equivalent in their disjoint union DFA U.
- The successors of any pair of equivalent states in U are again equivalent.
- Neither D' nor B have inaccessible states.

Thus every state of D' is equivalent to some state of B. To see this, let p be some state of D'. Then, as D' has no inaccessible states, there is some word $\sigma_1 \dots \sigma_m$ which gives a path from the start state of D' to p. But now the same word gives a path from the start state of B to some state q in B (as B is on the same alphabet as D'), and these states are thus equivalent via Lemma 2.45 (as the start states of D' and B are equivalent, since $\mathcal{L}(D') = \mathcal{L}(B)$).

Now, since B has fewer states than D', then by the pigeonhole principle there must be two different states of D' which are equivalent to the same state of B. Thus these two states of D' are equivalent to each other. But two equivalent states of D' must be the same state (Lemma 2.50); a contradiction. So A must have at least as many states as D'.

If A has the same number of states as D', then again we see that it cannot have any inaccessible states (and neither does D'). Also, neither A nor D' can have any equivalent states, by the first part of the theorem. Thus, in the disjoint union DFA of A and D' (as defined above), each state of A is equivalent to precisely one state of D', and vice-versa. As pairs of equivalent states in a DFA are preserved under the transition function, we have that each state of A 'matches' precisely one state of D' (same symbols transitioning in, from matching states; same symbols transitioning out, to matching states). □

Theorem 2.55 (Testing equivalence of regular languages)**.**
There is an algorithm that, on input of DFA's D_1, D_2, determines whether or not they define the same regular language.

Proof. Let $D_1 = (Q_1, \Sigma_1, \delta_1, q_{1,0}, F_1)$, $D_2 = (Q_2, \Sigma_2, \delta_2, q_{2,0}, F_2)$ respectively. Set $L_i := \mathcal{L}(D_i)$ for $i = 1, 2$. Now replace D_1 with a new DFA A_1 with:
- Alphabet $\Sigma_1 \cup \Sigma_2$.
- States $Q_1 \sqcup \{z_1\}$ (some symbol z_1 disjoint from Q_1).
- Start state $q_{1,0}$.
- Accept states F_1.
- Transition function ρ_1 which extends δ_1 by defining $\rho_1(q,\sigma) := z_1$ for all $q \in Q_1$ and all $\sigma \in \Sigma_2 - \Sigma_1$, and $\rho_1(z_1,\sigma) := z_1$ for all $\sigma \in \Sigma_1 \cup \Sigma_2$.

Replace D_2 with a new DFA A_2 in an analogous manner, interchanging the subscripts 1 and 2. Then remove the inaccessible states of each A_i (Theorem 2.53), and call the resulting DFA B_i. It is then immediate that $\mathcal{L}(B_i) = \mathcal{L}(A_i) = \mathcal{L}(D_i) = L_i$ for $i = 1, 2$.

Now form $B_1/\sim$ and $B_2/\sim$, and compute if they are equivalent. As they are on the same alphabet, then by Theorem 2.54 this occurs if and only if $L_1 = L_2$. □

3. PUSHDOWN AUTOMATA AND CONTEXT-FREE LANGUAGES

Having dealt with finite-state automata, we saw that they were very straightforward to work with, but were highly limited in the languages they could recognise. The reason for this is that they have a bounded *memory*; even though they can recognise arbitrarily long words, in a sense they are only seeing a 'finite number of options'. An example of this is computing the remainder $n \mod m$, where we only need to keep track of the remainder as we read along the input word, and this can only take one of a finite number of values.

We now describe a slightly more general finite-state machine; one with an unbounded 'memory stack', which can compute not only languages, but also sentences with some form of structure. This new form of computation is still weaker than register machines, but can do more than DFA's.

3.1. **Context-free grammars and context-free languages.**

We will start in the reverse order this time, and will define context-free grammars (akin to regular expressions) as a way to generate languages algebraically. Later, we will show that these give the same languages as a more general finite-state machine, known as a pushdown automaton.

Definition 3.1 (Context-free grammar)**.**
We define a *context-free grammar* (CFG) to be a quadruple (N, Σ, P, S), with

(1) A finite set of *nonterminal symbols* N.
(2) A finite set of *terminal symbols* Σ, disjoint from N.
(3) A finite set of *productions* $P \subset N \times (N \cup \Sigma)^*$.
(4) A *start symbol* $S \in N$.

We will often use capital letters $A, B, C, \dots$ for nonterminal symbols, and lower case letters $a, b, c, \dots$ for terminal symbols. Words in $(N \cup \Sigma)^*$ will often be written with Greek letters $\alpha, \beta, \gamma, \dots$.

We will often write productions (A, α) as $A \to \alpha$, to emphasise that there is some sort of substitution occurring here. For convenience, we will collect together productions with the same first (nonterminal) symbol, and use a vertical bar $\mid$ to separate all the words associated to that nonterminal. For example, if we had productions $(A, \alpha_1), (A, \alpha_2), (A, \alpha_3)$, we would write this as

$$A \to \alpha_1 \mid \alpha_2 \mid \alpha_3$$

Example 3.2. *Here is a small example of a CFG:*
$N = \{S\}$
$\Sigma = \{a, b\}$
$P = \{(S, aSb), (S, \epsilon)\}$
In later cases we would write P *as* $P = \{S \to aSb \mid \epsilon\}$*, according to our convention above.*

Later, we will see what this example actually represents.

Definition 3.3 (CFG terminology)**.**
Let $G = (N, \Sigma, P, S)$ be a CFG.

(1) Let $\alpha, \beta \in (N \cup \Sigma)^*$. We say that β is *derivable from* α *in one step* if β can be obtained from α by replacing some nonterminal A occurring in α with $\gamma \in (N \cup \Sigma)^*$, where $(A, \gamma) \in P$. That is, there exist $\alpha_1, \alpha_2 \in (N \cup \Sigma)^*$ and production $A \to \gamma$ such that $\alpha = \alpha_1 A \alpha_2$ and $\beta = \alpha_1 \gamma \alpha_2$

We write this as

$$\alpha \xrightarrow[G]{1} \beta$$

(indicating that, in the CFG G, we require *one* such substitution to go from α to β).

(2) For $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$, we inductively define the following notation:

$\alpha \xrightarrow[G]{0} \alpha$ for any α.

$\alpha \xrightarrow[G]{n+1} \beta$ if there exists γ with $\alpha \xrightarrow[G]{n} \gamma$ and $\gamma \xrightarrow[G]{1} \beta$.

$\alpha \xrightarrow[G]{*} \beta$ if there exists $n \geq 0$ with $\alpha \xrightarrow[G]{n} \beta$.

(3) If $\alpha \xrightarrow[G]{*} \beta$ then we say that β is *derivable* from α.

(4) If β is derivable from α, then a *derivation* (of length n) of β from α is a sequence of n steps from α to β. That is, a sequence $\gamma_1, \ldots, \gamma_{n-1} \in (N \cup \Sigma)^*$ such that

$$\alpha \xrightarrow[G]{1} \gamma_1 \xrightarrow[G]{1} \gamma_2 \xrightarrow[G]{1} \cdots \xrightarrow[G]{1} \gamma_{n-1} \xrightarrow[G]{1} \beta$$

(5) A word $w \in (N \cup \Sigma)^*$ which is derivable from the start symbol S is said to be in *sentential form*.

(6) A word $w \in (N \cup \Sigma)^*$ in sentential form is said to be a *sentence* if it contains only terminal symbols. That is, if $w \in \Sigma^*$.

Definition 3.4 (Language of a CFG)**.**
Let $G = (N, \Sigma, P, S)$ be a CFG. The *language generated by* G, $\mathcal{L}(G)$, is the set of all sentences derivable by G. That is,

$$\mathcal{L}(G) := \{w \in \Sigma^* \mid S \xrightarrow[G]{*} w\}$$

A language L is a *context-free language* (CFL) if $L = \mathcal{L}(G)$ for some CFG G.

By keeping our convention of writing nonterminals in uppercase (with start symbol always given by S), and terminals in lowercase, we can fully describe a CFL from the *productions* P of a CFG describing it. That is, we don't need to specify N, Σ, S explicitly[15]. For example, we can describe the language of the CFG from Example 3.2 simply by writing the productions, which are

$$S \to aSb \mid \epsilon$$

The only terminals that can appear in a CFL are terminals appearing in the productions, so there is no need to explicitly give the finite alphabet Σ.

Example 3.5. *The set $X = \{a^n b^n \mid n \geq 0\}$ is a CFL, generated by the CFG G from Example* 3.2*. That is, X is generated by the grammar*

$$S \to aSb \mid \epsilon$$

To see that $X \subseteq \mathcal{L}(G)$, we induct on n to show that

$$S \xrightarrow[G]{n+1} a^n b^n$$

Conversely, an induction on the length of derivations shows that $\mathcal{L}(G) \subseteq X$.

[15]This bypasses unused nonterminals and terminals, but we get the same language.

Note that this language is not a regular language (Example 2.40), and so we immediately see that CFL's can be non-regular. On example sheet 4 we will see that every regular language is a CFL, and thus there is a strict inclusion (regular language $\Rightarrow$ CFL) of these types of languages.

Example 3.6. *The set $X = \{w \in \{a,b\}^* \mid w = w^R\}$ is a CFL, generated by the CFG G defined by*

$$S \to aSa \mid bSb \mid a \mid b \mid \epsilon$$

The first two productions $S \to aSa \mid bSb$ give the symmetry of X when read from the left and right sides simultaneously. The next two productions $S \to a \mid b$ 'finish off' palindromes of odd length, and the final production $S \to \epsilon$ 'finishes off' palindromes of even length.

Now we look at a more involved example, which is very important in real-world computation[16]. This is the set of balanced parentheses; strings of ['s and]'s which obey certain containment rules.

Definition 3.7 (Balanced parentheses)**.**
Take any string $x \in \{\ [\ ,\ ]\ \}^*$. We define

(1) $L(x) := \#[(x) =$ the number of left parentheses [occurring in x.
(2) $R(x) := \#](x) =$ the number of right parentheses] occurring in x.

We say that x is *balanced* if

(i) $L(x) = R(x)$.
(ii) For all prefixes y of x, we have that $L(y) \geq R(y)$.

Lemma 3.8 (A CFG for balanced parentheses)**.**
Let G be the CFG

$$S \to [S] \mid SS \mid \epsilon$$

Then $\mathcal{L}(G) = \{x \in \{\ [\ ,\ ]\ \}^* \mid x \text{ is balanced}\}$, and so this is a CFL.

A quick application of the pumping lemma for regular languages shows that the language of balanced parentheses is not regular. Later, we will see another version of the pumping lemma, this time for CFL's.

To show that $x \in \mathcal{L}(G) \Rightarrow x$ is balanced is an induction on the length of the shortest possible derivation for x. To show x is balanced $\Rightarrow x \in \mathcal{L}(G)$ is an induction on $|x|$. We give the proof here in full, but it is long and technical.

Proof. 1. $x \in \mathcal{L}(G) \Rightarrow x$ balanced:
We will show that if $\alpha \in (N \cup \Sigma)^*$, and $S \xrightarrow[G]{*} \alpha$, then α satisfies (i) and (ii) from Definition 3.7. We do this by induction on the length of this derivation.
Basis:
If $S \xrightarrow[G]{0} \alpha$, then $\alpha = S$. But S contains no occurrence of [or], and so satisfies (i) and (ii).
Induction:
Suppose our assumption holds for all $k \leq n$, and suppose $S \xrightarrow[G]{n+1} \alpha$. Then we have β such that

$$S \xrightarrow[G]{n} \beta \xrightarrow[G]{1} \alpha$$

By induction, we have that β satisfies (i) and (ii). Now, there are three possible productions that we could apply to β to get α; we show that each preserves (i)

[16]If you ever want your C++ code to compile.

and (ii).
If we applied a production of the form $S \to SS$, or $S \to \epsilon$, then we are automatically done; neither of these productions changes the ordering of parentheses. In particular, we have (for either case) some pair $\beta_1, \beta_2 \in (N \cup \Sigma)^*$ such that

$$\beta = \beta_1 S \beta_2 \quad \text{and} \quad \alpha = \begin{cases} \beta_1\beta_2 & \text{if we applied } S \to \epsilon \\ \beta_1 S \beta_2 & \text{if we applied } S \to SS \end{cases}$$

In either case, α satisfies (i) and (ii), as β does.
So suppose that instead we applied the production $S \to [S]$. Then there exist $\beta_1, \beta_2 \in (N \cup \Sigma)^*$ such that

$$\beta = \beta_1 S \beta_2 \quad \text{and} \quad \alpha = \beta_1 [S] \beta_2$$

Then clearly $L(\alpha) = L(\beta) + 1 = R(\beta) + 1 = R(\alpha)$, as (i) holds in β by the induction hypothesis, and so holds in α also.
Now, to show (ii) holds in α, take any prefix y of α. We consider all the sub cases:

(1) y is a prefix of β_1, in which case y is a prefix of β, so (ii) holds for y by the induction hypothesis.
(2) y is a prefix of $\beta[S$, but not of β_1, in which case:

$$L(y) = L(\beta_1) + 1 \geq R(\beta_1) + 1 > R(\beta_1) = R(y)$$

(3) $y = \beta_1[S]\delta$ where δ is a prefix of β_2, in which case

$$L(y) = L(\beta_1 S \delta) + 1 \geq R(\beta_1 S \delta) + 1 = R(y)$$

In all these subcases, we have $L(y) \geq R(y)$, and so (ii) holds in α. So we're done.
2. x is balanced $\Rightarrow \in \mathcal{L}(G)$:
We do this by induction on $|x|$.
Basis:
If $|x| = 0$, then $x = \epsilon$ (which is balanced) and so can be formed from S by the single production $S \to \epsilon$.
Induction:
We break this up into two cases:
Case a): there exists a proper prefix y of x (i.e., $0 < |y| < |x|$) satisfying (i) and (ii). In this case, we have $x = yz$ for some z, with $0 < |z| < |x|$, and z satisfies (i) and (ii) as well, as

$$L(z) = L(x) - L(y) = R(x) - R(y) = R(z)$$

and for any prefix w of z we have

$$\begin{aligned} L(w) &= L(yw) - L(y) \\ &\geq R(yw) - R(y) \quad \text{since } yw \text{ is a prefix of } x \text{ and } L(y) = R(y) \\ &= R(w) \end{aligned}$$

By induction, we thus have that $y, z \in \mathcal{L}(G)$ (that is, $S \xrightarrow[G]{*} y$ and $S \xrightarrow[G]{n} z$). So we can derive x from S by first applying the derivation $S \to SS$, and then using the above derivations, via

$$S \xrightarrow[G]{1} SS \xrightarrow[G]{*} yS \xrightarrow[G]{*} yz = x$$

Case b): no such prefix y exists.
Thus $x = [z]$ (exercise: check this), for some z satisfying (i) and (ii). z satisfies (i) as we have

$$L(z) = L(x) - 1 = R(x) - 1 = R(z)$$

and z satisfies (ii) since, for every non-empty prefix u of z, we have

$$L(u) - R(u) = L([u) - 1 - R([u) \geq 0$$

(since $L([u) - R([u) \geq 1$ as we are in case b)). By the induction hypothesis, $S \xrightarrow[G]{*} z$. Combining this derivation with the single production $S \to [S]$, we get the following derivation of x:

$$S \xrightarrow[G]{1} [S]S \xrightarrow[G]{*} [z] = x$$

So every balanced word can be derived. □

3.2. **The Chomsky normal form.**

We give a way of converting *any* CFG to one of a particular form, called a Chomsky[17] normal form. All our productions will have a particular format, and this will come in handy for later proofs as we can assume that our CFG is structured in a way that is easy to work with.

Definition 3.9 (Chomsky normal form)**.**
A CFG $G = (N, \Sigma, P, S)$ is said to be in *Chomsky normal form* (CNF) if all productions are of the form

$$A \to BC \quad \text{or} \quad A \to a$$

where $A, B, C \in N$ and $a \in \Sigma$.

Example 3.10. *The following CFG is in Chomsky normal form:*

$$S \to AB \mid AC \mid SS, \quad C \to SB, \quad A \to [\ , \quad B \to\ ]$$

Later, we will see that this CFG gives the CFL of all balanced parentheses from Definition 3.7*.*

Observe that no CFG in CNF can generate the empty word ϵ. We will now show that this is the *only* limitation of this normal form, in the sense that every CFG has at least one corresponding CNF which generates the same language (minus the empty word ϵ).

Definition 3.11 (ϵ- and unit productions)**.**
Let $G = (N, \Sigma, P, S)$ be a CFG.

(1) An *ϵ-production* is a production of the form $A \to \epsilon$, for some $A \in N$.
(2) A *unit production* is a production of the form $A \to B$, for some $A, B \in N$.

ϵ- and unit productions are a hindrance to finding CNF's for CFG's. We show that we can always modify a CFG so as to remove all ϵ- and unit productions, and still end up with exactly the same CFL (minus $\{\epsilon\}$). Recall that, for sets A, B we write the set difference $A - B$ to denote the set $(A \cup B) \setminus B$.

[17]Named after the linguist, logician, philosopher and political activist Noam Chomsky, whom you may have seen on the news (but probably not for his mathematical work).

Theorem 3.12 (Removal of ϵ- and unit productions).
Let $G = (N, \Sigma, P, S)$ be a CFG. Then we can construct a CFG G', with no ϵ- or unit productions, for which $\mathcal{L}(G') = \mathcal{L}(G) - \{\epsilon\}$.

Proof. Let $\hat{P}$ be the smallest set of productions containing P and closed under the following rules:

(1) If $A \to \alpha B\beta$ and $B \to \epsilon$ are in $\hat{P}$, then $A \to \alpha\beta$ is in $\hat{P}$.
(2) $A \to B$ and $B \to \gamma$ are in $\hat{P}$, then $A \to \gamma$ is in $\hat{P}$.

We can construct $\hat{P}$ inductively: let $P_0 = P$ and in each iteration we take P_i and form P_{i+1} by adding all the productions needed to satisfy (1), (2) above for P_i. As P is finite, and as the right hand side of each added production is no longer than the right hand side of an existing production, we get that this series eventually stabilises in finitely many iterations (i.e., $P_n = P_{n+1}$ for some n). When this occurs, there is nothing to add, so we set $\hat{P} := P_n$.

Now define a new CFG $\hat{G} := (N, \Sigma, \hat{P}, S)$. Since $P \subseteq \hat{P}$, every derivation of G is a derivation of $\hat{G}$, and so $\mathcal{L}(G) \subseteq \mathcal{L}(\hat{G})$. But now we can conclude that $\mathcal{L}(G) = \mathcal{L}(\hat{G})$, as every new production in P_{i+1} not in P_i can be simulated by two productions of the (from (1) or (2)) in P_i. Thus we can 'pull back' (though the P_i's) any derivation in $\hat{G}$ to a (possibly longer) derivation in G.

We now show that we can remove all the ϵ- and unit productions from $\hat{P}$, and not change the language of the CFG (apart from removing $\{\epsilon\}$).

Take any $w \in \Sigma^*$ with $w \neq \epsilon$, and consider a minimal length derivation $S \xrightarrow[\hat{G}]{*} w$. Assume that an ϵ-production $B \to \epsilon$ is used in this derivation, and so

$$S \xrightarrow[\hat{G}]{*} \gamma B\delta \xrightarrow[\hat{G}]{1} \gamma\delta \xrightarrow[\hat{G}]{*} w$$

At least one of γ, δ is not ϵ, otherwise w would be. So that particular occurrence of B must have appeared earlier in the derivation as a production $A \to \alpha B\beta$, and so we have that our minimal-length derivation looks like

$$S \xrightarrow[\hat{G}]{m} \eta A\theta \xrightarrow[\hat{G}]{1} \eta\alpha B\beta\theta \xrightarrow[\hat{G}]{n} \gamma B\delta \xrightarrow[\hat{G}]{1} \gamma\delta \xrightarrow[\hat{G}]{k} w$$

for some $m, n, k \geq 0$. But by rule (1), the production $A \to \alpha\beta$ is also in $\hat{P}$, and so we have a strictly shorter derivation

$$S \xrightarrow[\hat{G}]{m} \eta A\theta \xrightarrow[\hat{G}]{1} \eta\alpha\beta\theta \xrightarrow[\hat{G}]{n} \gamma\delta \xrightarrow[\hat{G}]{k} w$$

contradicting the minimality of our original derivation. So we can 'discard' all ϵ-productions from P, and not change $\mathcal{L}(\hat{G})$.

Similarly, now assume that a unit production $A \to B$ is used in this minimal-length derivation of w, say

$$S \xrightarrow[\hat{G}]{*} \alpha A\beta \xrightarrow[\hat{G}]{1} \alpha B\beta \xrightarrow[\hat{G}]{*} w$$

Eventually, that particular occurrence of B will be replaced with some production $B \to \gamma$ (as B is non-terminal). So we have the derivation

$$S \xrightarrow[\hat{G}]{m} \alpha A\beta \xrightarrow[\hat{G}]{1} \alpha B\beta \xrightarrow[\hat{G}]{n} \eta B\theta \xrightarrow[\hat{G}]{1} \eta\gamma\theta \xrightarrow[\hat{G}]{k} w$$

for some $m, n, k \geq 0$. But by rule (2), the production $A \to \gamma$ is also in $\hat{P}$, and so we have a strictly shorter derivation

$$S \xrightarrow[\hat{G}]{m} \alpha A \beta \xrightarrow[\hat{G}]{1} \alpha\gamma\beta \xrightarrow[\hat{G}]{n} \eta\gamma\theta \xrightarrow[\hat{G}]{k} w$$

contradicting the minimality of our original derivation. So we can 'discard' all unit productions from P, and not change $\mathcal{L}(\hat{G})$.

So, by discarding all ϵ- and unit productions from $\hat{P}$, we end up with a CFG G' with no ϵ- or unit productions, for which $\mathcal{L}(G') = \mathcal{L}(\hat{G}) - \{\epsilon\} = \mathcal{L}(G) - \{\epsilon\}$. $\square$

Lemma 3.13.
Let $G = (N, \Sigma, P, S)$ be a CFG. Then we can construct a CFG $G' = (N', \Sigma, P', S)$ with the same terminals Σ, with $\mathcal{L}(G') = \mathcal{L}(G)$, and in which every production is of the form

$$A \to a \quad \text{or} \quad A \to B_1 \cdots B_k,\ k \geq 1$$

for some $A, B_1 \ldots B_k \in N$ and $a \in \Sigma \cup \{\epsilon\}$.

Proof. For each terminal $a \in \Sigma$, we add to N a new nonterminal A_a (distinct from all the existing nonterminals) and a production $A_a \to a$. Call this new set of nonterminals N'. Now replace all occurrences of a on the right hand side of productions in P with A_a, except productions already of the form $B \to a$. Call the new set of productions P'; then all productions in this set are of one of two forms:

$$A \to a \quad \text{or} \quad A \to B_1 \cdots B_k,\ k \geq 1$$

for some $A, B_1 \ldots B_k \in N'$ and $a \in \Sigma \cup \{\epsilon\}$. Now set $G := (N', \Sigma, P', S)$; we show that $\mathcal{L}(G') = \mathcal{L}(G)$.

To see that $\mathcal{L}(G) \subseteq \mathcal{L}(G')$, observe that a production from P with terminals and non-terminals in the right hand side can be reproduced by first applying the corresponding production in P' with only nonterminals in the right hand side, and then applying productions of the form $A \to a$ to recover the terminals.

To see that $\mathcal{L}(G') \subseteq \mathcal{L}(G)$, observe that derivations of sentences with P' can be simulated by P. $\square$

Theorem 3.14 (Realising Chomsky normal form)**.**
From a CFG $G = (N, \Sigma, P, S)$ we can construct an associated CFG G_{Chom} in CNF such that

$$\mathcal{L}(G_{\text{Chom}}) = \mathcal{L}(G) - \{e\}$$

Proof. Take G and construct the CFG $G' = (N, \Sigma, P', S)$ from Theorem 3.12 with no ϵ- and unit productions for which $\mathcal{L}(G') = \mathcal{L}(G) - \{\epsilon\}$. Now apply the construction of Lemma 3.13 to G', to get a CFG $G'' = (N', \Sigma, P'', S)$ whose productions are all of the form

$$A \to a \quad \text{or} \quad A \to B_1 \cdots B_k,\ k \geq 2$$

for some $A, B_1 \ldots B_k \in N'$ and $a \in \Sigma$. Observe that, since P' has no ϵ- or unit productions, then we have $a \neq \epsilon$ and $k > 1$ (this comes from the construction in Lemma 3.13).

Now, in P'', for any production of the form

$$A \to B_1 \cdots B_k$$

with $k \geq 3$ (and thus with each B_i nonterminal), we introduce a new nonterminal C and replace the original production with the following two productions:

$$A \to B_1 C \quad \text{and} \quad C \to B_2 \cdots B_k$$

Keep re-applying the above step until the right hand side of all productions are of length at most 2. Call the resulting set of nonterminals N'', and the resulting set of productions P'''. Then it is immediate that $G_{\text{Chom}} := (N'', \Sigma, P''', S)$ is in CNF, and moreover by the discussion above we have that

$$\mathcal{L}(G_{\text{Chom}}) = \mathcal{L}(G'') = \mathcal{L}(G') = \mathcal{L}(G) - \{e\}$$

□

Example 3.15. *Take the following CFG from Example* 3.8*:*

$$S \to \ [S] \ \mid \ SS \ \mid \ \epsilon$$

which gives the language of all balanced parentheses [ ]*. We first apply the construction of Theorem* 3.12 *to remove all* ϵ*- and unit productions to get the CFG*

$$S \to \ [S] \ \mid \ SS \ \mid \ [\,]$$

(the only ϵ*- production was* $S \to \epsilon$*, which we replaced with* $S \to [\,]$*. Also, there were no unit productions). This CFG generates the language of all* non-empty *balanced parentheses.*

Now we follow the construction of Theorem 3.14 *to build a CFG in CNF for this language. First, we add nonterminals* A, B *and replace the above productions with*

$$S \to \ ASB \ \mid \ SS \ \mid \ AB, \quad A \to [\ , \quad B \to \ ]$$

Finally, we add a new nonterminal C *and replace* $S \to ASB$ *with* $S \to AC$ *and* $C \to SB$*. So we have the CFG*

$$S \to \ AC \ \mid \ SS \ \mid \ AB, \quad C \to SB, \quad A \to [\ , \quad B \to \ ]$$

which is in CNF and generates the language of all non-empty balanced parentheses.

3.3. Parse trees and the pumping lemma for context-free languages.

We will prove a pumping lemma for CFL's, similar in idea to the pumping lemma for regular languages. Before we do this, we need a few new ideas. The first of these helps us understand *how* a derivation is applied in a CFG to give a string of terminals.

Definition 3.16 (Parse trees)**.**
Let G be a CFG. A *parse tree* (or *derivation tree*) for a word $w \in \mathcal{L}(G)$ is a tree representing all the productions applied to S in a derivation of the word w. That is, a 'downward' tree with *root* (top vertex) S, whose vertices of valence > 1 are nonterminals of G, and whose *leaves* (vertices of valence 1) are all terminals and form w when we 'read from left to right'.

We build this tree from a derivation $S \xrightarrow[G]{*} w$ as follows:

(1) Place S as the root.
(2) If $S \to \alpha_1$ is the first production in the derivation, then we add $|\alpha_1|$ downward branches to S, and label the new leaves from left to right by the letters (terminals or nonterminals) of α_1.
(3) If the next production is $X_2 \to \alpha_2$, then we add $|\alpha_2|$ downward branches to the leaf X_2, and label the new leaves from left to right by the letters (terminals or nonterminals) of α_2.
(4) We keep doing this for each production in the derivation.

(5) At the end of this process, we will have no more nonterminals as leaves in the tree, only terminals.

We adopt the following 'drawing convention': when we go down a level of the tree (applying productions to all nonterminals on level n to bring us to level $n+1$), we may find that some of the vertices on level n were terminals. So as to not lose track of them, we add one downward branch for each such terminal, and duplicate them at level $n+1$ also. Thus, when we finish building the tree (say with m levels), we can read off the derived word by just reading level m from left to right.

The *depth* of a tree is the number of edges of the longest path from the root to a leaf; a tree of depth n will thus have $n+1$ levels.

Observe that, if G is a CFG in CNF, then a parse tree of G will have at most 2^n symbols at level n (the root is at level 0). This is because the number of symbols can at most double in a CNF derivation, and we start with one symbol S at level 0.

Theorem 3.17 (The pumping lemma for CFL's)**.**
Let L be a CFL. Then there exists a constant n (depending on L) such that for every word $z \in L$ with $|z| \geq n$ we can break up z into 5 words $z = uvwxy$ such that:

(1) *$vx \neq \epsilon$.*
(2) *$|vwx| \leq n$.*
(3) *For all $k \geq 0$, we have that the word $uv^k wx^k y$ is also in L.*

Proof. Let G be a CFG for L in CNF (this exists by Theorem 3.14). Take $n = 2^{m+1}$, where m is the number of nonterminals of G. Suppose $z \in L$ and $|z| \geq n$. By what we said above, any parse tree for z in G must be of depth at least $m+1$, as level m has at most 2^m symbols. Let γ be (a) longest possible path starting at the root in such a tree. That path must be of length at least $m+1$, and so contains at least $m+1$ occurrences of nonterminals (only the last vertex in the path can be a terminal). As G has only m nonterminals, then by the pigeonhole principle there is some nonterminal which occurs twice in γ. Take the first repeated nonterminal X in γ, when reading from the bottom of the tree up to the root.

Now break z up into substrings $uvwxy$ such that

(1) w is the string of terminals generated by the lower occurrence of X.
(2) vwx is the string of terminals generated by the upper occurrence of X.

Let T be the subtree rooted at the upper occurrence of X, and let t be the subtree rooted at the lower occurrence of X. Now we can 'pump' in two possible ways:
First way: We can remove the (lower, and thus smaller) subtree t from the original tree, and replace it with a copy of the (upper, and thus larger) subtree T. This gives us a valid parse tree for the word uv^2wx^2y. We can continue doing this several times, each time removing t and replacing it with a copy of T, to get a valid parse tree for uv^iwx^iy for every $i \geq 1$.
Second way: We can remove the (upper, and thus larger) subtree T from the original tree, and replace it with a copy of the (lower, and thus smaller) subtree t. This gives us a valid parse tree for the word uwy.

Observe that $vx \neq \epsilon$; that is, at least one of v, x are non-null, as we have taken two occurrences of X at different levels up the path γ, so one 'side' of the upper occurrence of X leads to terminals. This might be the left side (i.e., v), or the right side (i.e., x).

Also, observe that $|vwx| \leq n$, as we chose the *first* repeated occurrence of a nonterminal reading up the path γ, and so this must happen at height at most $m+1$. Since γ was chosen to be the longest path in the original tree, then the subderivation down from this upper occurrence of X gives us a tree of depth at most $m+1$, and thus has at most $2^{m+1} = n$ terminals. □

We can apply the pumping lemma for CFL's to show that a language is not a CFL, in a similar way to how we apply the pumping lemma for regular languages.

Example 3.18. *The set $A = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not a CFL.*

Proof. Suppose A were a CFL. Take n as in the pumping lemma (Lemma 3.17), and consider the word $z = a^n b^n c^n$. Regardless of how we decompose $z = uvwxy$ with $|vwx| \leq n$, we have that vwx either contains no occurrence of a, or contains no occurrence of c (or no occurrence of either). Thus, when we pump z to $uv^k wx^k y$, there will be at least one letter (a or c) for which the number of occurrences of that letter does not change from $uvwxy$ to $uv^k wx^k y$. However, there will be a different letter for which the number of occurrences of that letter does change from $uvwxy$ to $uv^k wx^k y$. Thus $uv^k wx^k y$ cannot be of the form $a^m b^m c^m$, for any $k \neq 1$. □

3.4. **Nondeterministic pushdown automata.**

In the previous chapter on regular languages, we first defined our languages via certain automata, and then gave an algebraic way to generate them via regular expressions. Here, we have done the reverse: we first defined our languages algebraically, and we now give a 'mechanical' way to generate them. These machines work in a very similar way to ϵ-NFA's, but they have access to a 'stack', where they can store a finite but unbounded amount of extra information about what has happened so far in the computation.

Definition 3.19 (Nondeterministic pushdown automata)**.**
A *nondeterministic pushdown automaton* (NPDA) is a structure
$M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ consisting of the following:

(1) A finite set of *states* Q.
(2) A finite *input alphabet* Σ.
(3) A finite *stack alphabet* Γ.
(4) A *transition function* $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to \mathcal{P}(Q \times \Gamma^*)$ which is total.
(5) A designated *start state* $q_0 \in Q$.
(6) A designated *initial stack symbol* $\perp \in \Gamma$.
(7) A finite set of *accept states* $F \subseteq Q$.

Note that our transition function is nondeterministic, and also allows for ϵ-transitions.

The *input* of an NPDA is any finite string $w = \sigma_1 \ldots \sigma_k \in \Sigma^*$, and a vertical *stack* of symbols which initially contains just $\perp$. The NPDA takes w, reads the first symbol σ_1 whilst 'in' the start state q_0, evaluates the transition function $\delta(q_0, \sigma_1, \perp)$ and then simultaneously/nondeterministically does all of the following: for each $(q, B_1 \cdots B_m) \in \delta(q_0, \sigma_1, \perp)$ the NPDA 'moves to' the new state q, removes (*pops*) the top symbol $\perp$ of the stack and replaces it with (*pushes*)

$B_1 \cdots B_m$, with B_m going on to the stack first and thus B_1 ending up at the top of the stack. The NPDA then reads the next symbol σ_2 of w, and repeats the process. This continues for the entire word w.

Before we define what it means for an NPDA to accept an input, we need a way to keep track of the calculation it is performing[18].

Definition 3.20 (Configurations of NPDA's)**.**
Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ be an NPDA. A *configuration* of M is some $(p, w, \gamma) \in Q \times \Sigma^* \times \Gamma^*$, where

(1) p is the state that the NPDA is currently in.
(2) w is the part of the input word which remains to be read.
(3) γ is the contents of the stack (the leftmost letter being at the top; the rightmost at the bottom).

The *start configuration* on input w is denoted $(q_0, w, \perp)$.
To see how the NPDA moves from one configuration to another, we define the *next configuration relation* $\xrightarrow[M]{1}$:
If $(p, a, A) \in \delta(q, \gamma)$, then for any $y \in \Sigma^*$ and $\beta \in \Gamma$ we write

$$(p, ay, A\beta) \xrightarrow[M]{1} (q, y, \gamma\beta)$$

With this, for any configurations C, D, E we inductively define the following notation:

$$C \xrightarrow[M]{0} D \ \text{ if } C = D.$$

$$C \xrightarrow[M]{n+1} D \ \text{ if there exists } E \text{ with } C \xrightarrow[M]{n} E \text{ and } E \xrightarrow[M]{1} D.$$

$$C \xrightarrow[M]{*} D \ \text{ if there exists } n \geq 0 \text{ with } C \xrightarrow[M]{n} D.$$

We can now define acceptance by an NPDA:

Definition 3.21 (Acceptance by an NPDA)**.**
An NPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ is said to *accept w by final state* if $(q_0, w, \perp) \xrightarrow[M]{*} (q, \epsilon, \gamma)$ for some $q \in F$ and $\gamma \in \Gamma^*$. M is said to *accept w by empty stack* if $(q_0, w, \perp) \xrightarrow[M]{*} (q, \epsilon, \epsilon)$ for some $q \in Q$.
We say M *accepts by final state* to mean that the language of M is given by all words w which M accepts by final state. That is,

$$M \text{ accepts by final state } \Rightarrow \mathcal{L}(M) := \{w \in \Sigma^* \mid M \text{ accepts } w \text{ by final state}\}$$

We say M *accepts by empty stack* to mean that the language of M is given by all words w which M accepts by empty stack. That is,

$$M \text{ accepts by empty stack } \Rightarrow \mathcal{L}(M) := \{w \in \Sigma^* \mid M \text{ accepts } w \text{ by empty stack}\}$$

We make the following two remarks:

(1) Because of ϵ-transitions, it is possible for an NPDA to enter an infinite loop and never finish reading the input word; it can just keep modifying the stack forever, without reading any further symbols of the input word.

[18]As we have to worry about the state, the remainder of the tape, and the contents of the stack.

(2) If the stack ever becomes empty *before* the *entire* input word is read, then the machine becomes stuck, as there is no transition function to apply.

3.5. **Equivalence of acceptance by final state or empty stack.**

It turns out that NPDA's which accept by final state have exactly the same computational power as those which accept by empty stack.

Definition 3.22 (Constructing an NPDA where acceptance by final state and empty stack coincide)**.**
Given an arbitrary NPDA M which accepts by either final state or empty stack, we can construct an NPDA M' with a single accept state for which acceptance by final state or empty stack coincide. The construction depends *slightly* on whether M itself accepts by final state or empty stack; we do the two constructions together, and point out the places where they differ.
Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ be our NPDA that accepts by final state or empty stack. Take two new symbols u, t not in Q, and $\perp\!\!\!\perp$ a new stack symbol not in Γ. Now define

$$G := \begin{cases} Q & \text{if } M \text{ accepts by empty stack.} \\ F & \text{if } M \text{ accepts by final state.} \end{cases}$$

$$\Delta := \begin{cases} \{\perp\!\!\!\perp\} & \text{if } M \text{ accepts by empty stack.} \\ \Gamma \cup \{\perp\!\!\!\perp\} & \text{if } M \text{ accepts by final state.} \end{cases}$$

Now define the NPDA M' by

$$M' = (Q \cup \{u, t\}, \Sigma, \Gamma \cup \{\perp\!\!\!\perp\}, \delta', u, \perp\!\!\!\perp, \{t\})$$

where we define δ' as an extension of δ by adding the following:

$$\delta'(u, \epsilon, \perp\!\!\!\perp) := \{(q_0, \perp \perp\!\!\!\perp)\} \tag{1}$$

$$\delta'(q, \epsilon, A) := \delta(q, \epsilon, A) \cup \{(t, A)\},\ \forall q \in G,\ \forall A \in \Delta \tag{2}$$

$$\delta'(t, \epsilon, A) := \{(t, \epsilon)\},\ \forall A \in \Gamma \cup \{\perp\!\!\!\perp\} \tag{3}$$

and $\delta' := \delta$ for all other inputs.

So our new automaton M' has a new start state u, a new initial stack symbol $\perp\!\!\!\perp$, and a new single final state t. It computes as follows:

(1) In its first computational step, it pushes the old initial stack symbol $\perp$ on top of $\perp\!\!\!\perp$ (via an ϵ-transition), then enters the old start state q_0.
(2) It then runs precisely like M, as it has all the transitions of M, and has $\perp$ on top of its stack.
(3) At some point it might enter state t; its accept state. If it does, then it proceeds to empty its entire stack (via ϵ-transitions).
(4) The *only* way M' can empty its stack is if it enters state t; no other state allows it to pop $\perp\!\!\!\perp$.
(5) Between reaching state t and emptying its stack, M' does not read any more of its input word.

Thus M' accepts by empty stack iff it accepts by final state.

Theorem 3.23 (Accepting by final state and empty stack)**.**
Take an NPDA M, and construct the NPDA M' as per Definition 3.22*. Then* $\mathcal{L}(M') = \mathcal{L}(M)$.

Proof. We first show $\mathcal{L}(M) \subseteq \mathcal{L}(M')$:
If M accepts by empty stack, and accepts w, then $(q_0, w, \perp) \xrightarrow[M]{n} (q, \epsilon, \epsilon)$ for some n. But then we have that

$$(u, w, \perp\!\!\perp) \xrightarrow[M']{1} (q_0, w, \perp \perp\!\!\perp) \xrightarrow[M']{n} (q, \epsilon, \perp\!\!\perp) \xrightarrow[M']{1} (t, \epsilon, \perp\!\!\perp) \xrightarrow[M']{1} (t, \epsilon, \epsilon)$$

and so M' accepts w.
If, instead, M accepts by final state, and accepts w, then $(q_0, w, \perp) \xrightarrow[M]{n} (q, \epsilon, \gamma)$ for some n, some $q \in F$, and some $\gamma \in \Gamma^*$. But then we have that

$$(u, w, \perp\!\!\perp) \xrightarrow[M']{1} (q_0, w, \perp \perp\!\!\perp) \xrightarrow[M']{n} (q, \epsilon, \gamma\perp\!\!\perp) \xrightarrow[M']{1} (t, \epsilon, \gamma\perp\!\!\perp) \xrightarrow[M']{*} (t, \epsilon, \epsilon)$$

and so M' accepts w.
Thus, in both cases, M' accepts[19] w if M does. So $\mathcal{L}(M) \subseteq \mathcal{L}(M')$.
We now show $\mathcal{L}(M') \subseteq \mathcal{L}(M)$:
Suppose M' accepts w by either mode (finite state or empty stack). Then we have that

$$(u, w, \perp\!\!\perp) \xrightarrow[M']{1} (q_0, w, \perp \perp\!\!\perp) \xrightarrow[M']{n} (q, y, \gamma\perp\!\!\perp) \xrightarrow[M']{1} (t, y, \gamma\perp\!\!\perp) \xrightarrow[M']{*} (t, \epsilon, \epsilon)$$

for some $q \in G$, $\gamma \in \Gamma^*$. But $y = \epsilon$, since M' can't read any input symbols once it enters state t (the only transitions involving t are ϵ-transitions; (3) from Definition 3.22). So by the way M is simulated by M', we have

$$(q_0, w, \perp) \xrightarrow[M]{n} (q, \epsilon, \gamma)$$

Now consider the definitions of G and of Δ, and the transitions of the form (2) from Definition 3.22 which describe what the first move into state t can be. If we try and analyse how the transition $(q, \epsilon, \gamma\perp\!\!\perp) \xrightarrow[M']{n} (t, \epsilon, \gamma\perp\!\!\perp)$ could come about, then we observe the following:

(1) If M accepts by empty stack, then we must have $\gamma = \epsilon$.
(2) If M accepts by final state, then we must have $q \in F$.

Either way, the transition $(q_0, w, \perp) \xrightarrow[M]{n} (q, \epsilon, \gamma)$ gives that M accepts w. Thus $\mathcal{L}(M') \subseteq \mathcal{L}(M)$. □

3.6. **Equivalence of CFL's and NPDA's.**

We can now prove that NPDA's accept precisely the set of CFL's. We do this in two parts. We first show that, from an NPDA, we can construct a CFG with the same accepted language. Then we show that the reverse is also possible: from a CFG we can construct an NPDA with the same accepted language. In both constructions, we see that the object we construct (NPDA or CFG) mimics the operation of the object we started with (CFG or NPDA) in some controlled way.

Definition 3.24 (Constructing an NPDA from a CFG)**.**
Given a CFG $G = (N, \Sigma, P, S)$, we construct from it an NPDA which accepts by empty stack, as follows:
First, we use Lemma 3.13 to re-write G so that all productions are of the form

$$A \to cB_1 \dots B_k$$

[19]By either finite state or empty stack, as these are equivalent for M'.

where $c \in \Sigma \cup \{\epsilon\}$ and $k \geq 0$. By Lemma 3.13, this new CFG accepts exactly the same language, and so we will discard our original CFG and call this new one by the same name ($G = (N, \Sigma, P, S)$).
Now, from G, we construct an NPDA $M = (\{q\}, \Sigma, N, \delta, q, S, \emptyset)$, where

(1) q is the sole state of M.
(2) Σ (the terminals of G) is the input alphabet of M.
(3) N (the nonterminals of G) is the stack alphabet of M.
(4) q is the start state of M.
(5) S (the start symbol of G) is the initial stack symbol of M.
(6) $\emptyset$ is the set of accept states of M (irrelevant, as M accepts by empty stack).
(7) δ, the transition function of M, is defined as follows: for each production $A \to cB_1 \dots B_k$ in P, we include $(q, B_1 \dots B_k)$ in the set $\delta(q, c, A)$.

Before we prove various facts about this construction, we need to introduce a particular type of derivation, known as a leftmost derivation.

Definition 3.25 (Leftmost derivation).
Let G be a CFG. A derivation $\beta \xrightarrow[G]{*} \gamma$ is said to be a *leftmost derivation* if each production in the derivation is applied to the leftmost nonterminal in the sentential form.

It is immediate that, if a word w lies in $\mathcal{L}(G)$ for some CFG G, then we can always derive w with a leftmost derivation $S \xrightarrow[G]{*} w$, by swapping the order of some of the productions. To see this, draw a parse tree for the derivation, then re-order the applications of productions; we still have the same parse tree, and thus another derivation $S \xrightarrow[G]{*} w$. This idea works because we are dealing with context-free grammars; ones in which we replace *one* nonterminal with some other string, and thus the *context* of this nonterminal (that is, the other symbols around it) does not matter.

The operation of the NPDA M constructed from the CFG G in Definition 3.24 is closely related to that of G. We will see that *leftmost* derivations of G from S to a sentence w of terminals correspond to an accepting computation of M on input w. More strongly: the sequence of sentential forms in the leftmost derivation of w corresponds to the sequence of configurations of M on input w. Thus we see that the machine M, and the CFG G, operate in the same way.

Lemma 3.26 (Operation of the NPDA constructed from a CFG).
Let G be a CFG, and M the NPDA constructed from it in Definition 3.24. *Then, for any $y, z \in \Sigma^*$, any $\gamma \in N^*$, and any $A \in N$, we have that*

$$A \xrightarrow[G]{n} z\gamma \text{ via a leftmost derivation } \Leftrightarrow (q, zy, A) \xrightarrow[M]{n} (q, y, \gamma)$$

Proof. We prove this by induction on n.
Basis: If $n = 0$ then

$$\begin{aligned} A \xrightarrow[G]{0} z\gamma &\Leftrightarrow A = z\gamma \\ &\Leftrightarrow z = \epsilon \text{ and } \gamma = A \\ &\Leftrightarrow (q, zy, A) = (q, y, \gamma) \\ &\Leftrightarrow (q, zy, A) \xrightarrow[M]{0} (q, y, \gamma) \end{aligned}$$

Induction (assuming the statement holds for all $k \leq n$):
We break this up into the forward ($\Rightarrow$) implication and the reverse ($\Leftarrow$) implication.
($\Rightarrow$):
Suppose $A \xrightarrow[G]{n+1} z\gamma$ via leftmost derivation. Suppose $B \to c\beta$ was the last production applied in this leftmost derivation, where $c \in \Sigma \cup \{\epsilon\}$ and $\beta \in N^*$. Then

$$A \xrightarrow[G]{n} uB\alpha \xrightarrow[G]{1} uc\beta\alpha = z\gamma$$

where $z = uc$ and $\gamma = \beta\alpha$. By induction, as $A \xrightarrow[G]{n} uB\alpha$, we have that

$$(q, ucy, A) \xrightarrow[M]{n} (q, cy, B\alpha)$$

But by the definition of δ for M, we have that $(q, \beta) \in \delta(q, c, B)$, as $B \to c\beta$ is a production of G. So we get

$$(q, cy, B\alpha) \xrightarrow[M]{1} (q, y, \beta\alpha)$$

Combining these, we see that

$$(q, zy, A) = (q, ucy, A) \xrightarrow[M]{n} (q, cy, B\alpha) \xrightarrow[M]{1} (q, y, \beta\alpha) = (q, y, \gamma)$$

and thus

$$(q, zy, A) \xrightarrow[M]{n+1} (q, y, \gamma)$$

($\Leftarrow$):
Suppose $(q, zy, A) \xrightarrow[M]{n+1} (q, y, \gamma)$. Suppose that $(q, c, B) \mapsto (q, \beta)$ was the last transition taken, where $(q, \beta) \in \delta(q, c, B)$. Then $z = uc$ for some $u \in \Sigma^*$, $\gamma = \beta\alpha$ for some $\alpha \in \Gamma^*$, and

$$(q, ucy, A) \xrightarrow[M]{n} (q, cy, B\alpha) \xrightarrow[M]{1} (q, y, \beta\alpha)$$

By induction, as $(q, ucy, A) \xrightarrow[M]{n} (q, cy, B\alpha)$, we have that $A \xrightarrow[G]{n} uB\alpha$ via a leftmost derivation in G. Moreover, by construction of M, we have that $B \to c\beta$ is a production of G (as $(q, \beta) \in \delta(q, c, B)$). But now we can apply this production to the sentential form $uB\alpha$ to get

$$A \xrightarrow[G]{n} uB\alpha \xrightarrow[G]{1} uc\beta\alpha = z\gamma$$

via leftmost derivation. □

Theorem 3.27 (Language of the NPDA constructed from a CFG)**.**
Let G be a CFG, and M the NPDA constructed from it in Definition 3.24*. Then* $\mathcal{L}(G) = \mathcal{L}(M)$.

Proof. Take any word $w \in \Sigma^*$. The we see that

$$\begin{aligned} w \in \mathcal{L}(G) &\Leftrightarrow S \xrightarrow[G]{*} w \text{ by a leftmost derivation} \\ &\Leftrightarrow (q, w, S) \xrightarrow[M]{*} (q, \epsilon, \epsilon) \quad (\text{ Lemma 3.26 }) \\ &\Leftrightarrow w \in \mathcal{L}(M) \quad (\text{ as M accepts by empty stack }) \end{aligned}$$

□

We now show that the reverse process is also possible. In fact, we will practically *invert* the construction.

Lemma 3.28 (Constructing a CFG from an NPDA with one state)**.**
Let $M = (\{q\}, \Sigma, \Gamma, \delta, q, \bot, \emptyset)$ be an NPDA with one state which accepts by empty stack. Define the CFG $G = (\Gamma, \Sigma, P, \bot)$, where P contains the production $A \to cB_1 \dots B_k$ for every case where $(q, B_1 \dots B_k) \in \delta(q, c, A)$, with $c \in \Sigma \cup \{\epsilon\}$. Then $\mathcal{L}(G) = \mathcal{L}(M)$.

Proof. This is the exact same argument used in Lemma 3.26 and Theorem 3.27, as all reasoning used was bi-directional ($\Leftrightarrow$). □

Of course, there is no immediate reason to assume that every NPDA is equivalent to one that accepts by empty stack and has only one accept state. We give a construction of this here, and then prove that our new NPDA accepts the same language as our old one.

Definition 3.29 (NPDA which accepts by empty stack, with 1 accept state)**.**
Take any NPDA K, and use Definition 3.22 and Theorem 3.23 to convert it to an NPDA $M = (Q, \Sigma, \Gamma, \delta, s, \bot, \{t\})$ which accepts by final state and by empty stack equivalently, has one final state t, and satisfies $\mathcal{L}(M) = \mathcal{L}(K)$.

Now, we define the set

$$\Gamma' := Q \times \Gamma \times Q$$

This is our new stack alphabet, and we will use this to 'simulate' the action of M on the stack of our new NPDA. We write elements of Γ' as $\langle p\ A\ q\rangle$, where $p, q \in Q$ and $A \in \Gamma$. We now construct our new NPDA M' to be

$$M' := (\{*\}, \Sigma, \Gamma', \delta', *, \langle s\ \ \bot\ \ t\rangle, \emptyset)$$

with one state $*$, where M' accepts by empty stack.

We define the transition function δ' of M' as follows: for each transition $(q_0, B_1, \dots, B_k) \in \delta(p, c, A)$ (where $c \in \Sigma \cup \{\epsilon\}$) we, for all possible choices $\{q_1, \dots, q_k\} \subseteq Q$, include $(*, \langle q_0\ B_1\ q_1\rangle\langle q_1\ B_2\ q_2\rangle \cdots \langle q_{k-1}\ B_k\ q_k\rangle)$ in the set $\delta'(*, c, \langle p\ A\ q_k\rangle)$.

Observe that, for $k = 0$, this reduces to the following: if $(q_0, \epsilon) \in \delta(p, c, A)$, then we include $(*, \epsilon)$ in $\delta'(*, c, \langle p\ A\ q_0\rangle)$.

The point of this construction is that the new machine M' will be able to scan a word w starting with only $\langle p\ A\ q\rangle$ on its stack and end up with an empty stack iff M can start scanning w in state p with only A on its stack and end up in state q with an empty stack.

The idea here is that M' simulates M, guessing nondeterministically which states M will be in at certain future points in the computation, saving those guesses on the stack, and then verifying later that those guesses were correct.

We now prove that these two NPDA's operate in an analogous manner.

Lemma 3.30.
Let M' be the NPDA constructed from M in Definition 3.29. *Then*

$$(p, w, B_1 \cdots B_k) \xrightarrow[M]{n} (q, \epsilon, \epsilon)$$

iff there exist $q_0, \dots, q_k$ such that $p = q_0, q = q_k$, and

$$(*, w, \langle q_0\ B_1\ q_1\rangle\langle q_1\ B_2\ q_2\rangle \cdots \langle q_{k-1}\ B_k\ q_k\rangle) \xrightarrow[M']{n} (*, \epsilon, \epsilon)$$

In particular, we have that

$$(p, w, B) \xrightarrow[M]{n} (q, \epsilon, \epsilon) \Leftrightarrow (*, w, \langle p\ B\ q\rangle) \xrightarrow[M']{n} (*, \epsilon, \epsilon)$$

Proof. We show this by induction on n. The base case $n = 0$ is trivial, as both sides are then equivalent to the assertion that $p = q$, $w = \epsilon$, and $k = 0$. So now suppose the assertion is true for all $l \leq n$.

Firstly, assume we have that $(p, w, B_1 \cdots B_k) \xrightarrow[M]{n+1} (q, \epsilon, \epsilon)$. Let $(p, c, B_1) \mapsto (r, C_1 \cdots C_m)$ be the first transition applied, where $c \in \Sigma \cup \{\epsilon\}$ and $m \geq 0$. Then we have that $w = cy$ and

$$\begin{aligned}(p, w, B_1 \cdots B_k) &\xrightarrow[M]{1} (r, y, C_1 \cdots C_m B_2 \cdots B_k)\\ &\xrightarrow[M]{n} (q, \epsilon, \epsilon)\end{aligned}$$

By induction, we have that there exist $r_0, \ldots, r_{m-1}, q_1, \ldots, q_k$ such that $r = r_0$, $q = q_k$, and

$$(*, y, \langle r_0\ C_1\ r_1\rangle \cdots \langle r_{m-1}\ C_m\ q_1\rangle\langle q_1\ B_2\ q_2\rangle \cdots \langle q_{k-1}\ B_k\ q_k\rangle) \xrightarrow[M']{n} (*, \epsilon, \epsilon)$$

Now, by construction of M', we have that

$$(*, \langle r_0\ C_1\ r_1\rangle \cdots \langle r_{m-1}\ C_m\ q_1\rangle) \in \delta'(*, c, \langle p\ B_1\ q_1\rangle)$$

Combining these, we get

$$\begin{aligned}&(*, w, \langle p\ B_1\ q_1\rangle\langle q_1\ B_2\ q_2\rangle \cdots \langle q_{k-1}\ B_k\ q_k\rangle)\\ &\quad\xrightarrow[M']{1} (*, y, \langle r_0\ C_1\ r_1\rangle \cdots \langle r_{m-1}\ C_m\ q_1\rangle\langle q_1\ B_2\ q_2\rangle \cdots \langle q_{k-1}\ B_k\ q_k\rangle)\\ &\quad\xrightarrow[M']{n} (*, \epsilon, \epsilon)\end{aligned}$$

Conversely, suppose we have

$$(*, w, \langle q_0\ B_1\ q_1\rangle\langle q_1\ B_2\ q_2\rangle \cdots \langle q_{k-1}\ B_k\ q_k\rangle) \xrightarrow[M']{n+1} (*, \epsilon, \epsilon)$$

So let

$$(*, c, \langle q_0\ B_1\ q_1\rangle) \mapsto (*, \langle r_0\ C_1\ r_1\rangle \cdots \langle r_{m-1}\ C_m\ q_1\rangle)$$

be the first transition applied, where $c \in \Sigma \cup \{\epsilon\}$ and $m \geq 0$. Then $w = cy$ and we have that

$$\begin{aligned}&(*, w, \langle q_0\ B_1\ q_1\rangle\langle q_1\ B_2\ q_2\rangle \cdots \langle q_{k-1}\ B_k\ q_k\rangle)\\ &\quad\xrightarrow[M']{1} (*, y, \langle r_0\ C_1\ r_1\rangle \cdots \langle r_{m-1}\ C_m\ q_1\rangle\langle q_1\ B_2\ q_2\rangle \cdots \langle q_{k-1}\ B_k\ q_k\rangle)\\ &\quad\xrightarrow[M']{n} (*, \epsilon, \epsilon)\end{aligned}$$

By induction, we have that

$$(r_0, y, C_1 \cdots C_m B_2 \cdots B_k) \xrightarrow[M]{n} (q_k, \epsilon, \epsilon)$$

Also, by construction of M', we have that $(r_0, C_1 \cdots C_m) \in \delta(q_0, c, B_1)$

Combining these, we see that

$$\begin{aligned}(q_0, w, B_1 \cdots B_k) &\xrightarrow[M]{1} (r_0, y, C_1 \cdots C_m B_2 \cdots B_k)\\ &\xrightarrow[M]{n} (q_k, \epsilon, \epsilon)\end{aligned}$$

□

Theorem 3.31. *Let M' be the NPDA constructed from M in Definition* 3.29*. Then $\mathcal{L}(M') = \mathcal{L}(M)$*

Proof. Take $w \in \Sigma^*$. Then

$$\begin{aligned} w \in \mathcal{L}(M') &\Leftrightarrow (*, w, \langle s \;\; \perp \;\; t\rangle) \xrightarrow[M']{*} (*, \epsilon, \epsilon) \\ &\Leftrightarrow (s, w, \perp) \xrightarrow[M]{*} (t, \epsilon, \epsilon) \quad (\text{ Lemma 3.30 }) \\ &\Leftrightarrow w \in \mathcal{L}(M) \end{aligned}$$

□

Corollary 3.32. *A language L is a CFL iff $L = \mathcal{L}(M)$ for some NPDA M.*

We can use this to give a (mechanical) proof that every regular language is a CFL.

Theorem 3.33. *Let L be a regular language. Then L is a CFL.*

Proof. Take an ϵ-NFA E with $\mathcal{L}(E) = L$. Then we can re-interpret E as an NPDA which accepts by final state, where we just need to introduce one dummy stack alphabet symbol $\perp$, and have every transition mimic one from E but where we pop $\perp$ from the stack and then push it straight back on again. □

There are more direct ways of proving that every regular language is a CFL, but we have shown that CFL's are a true 'generalisation' by showing that we have a more general machine.