Automata and Formal Languages

November 20, 2019

1 Register Machines and Computability

Books: PTJ (Chapter 4)

NOTE: HERE $\mathbb{N} = \{0, 1, 2, \ldots\}$

A ***register machine (RM)*** consists of:

1. A sequence of ***registers*** $R_1, R_2, R_3, \ldots$ where at discrete time steps $t = 0, 1, 2, \ldots$ have $R_i(t) \in \mathbb{N}$, In fact, we only have finitely many registers, and regard $R_i \equiv 0$ for all $i \geq I$.
2. A finite ***program*** consisting of a fixed number of ***states*** S_0 (HALT), S_1 (START), $S_2, \ldots, S_n$. Each state comes with a fixed instruction performed when in state S_i. When the computer reaches HALT, we get the output from R_1. Otherwise, for $1 \leq i \leq n$ we have 2 types of ***commands***:
   (a) Increment R_j, then move to state S_k. We write this $S_i : (j, +, k)$.
   (b) If $R_j \neq 0$ then decrement R_j, then move to state S_k. Otherwise move to state S_l. We write this $S_i : (j, -, k, l)$.

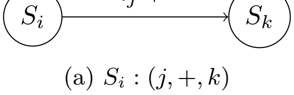
A ***sequence of instructions*** for a RM is the ordered list of the instructions for the program. An ***input*** for a RM is, for some $k \geq 1$, a finite k-tuple $(n_1, \ldots, n_k) \in \mathbb{N}^k$ which are the initial values of $R_1, \ldots, R_k$. The other registers are set to 0.

A ***program diagram*** for a RM is a directed graph with vertices being the states of the machine and the labelled arrows denote the instructions: $S_i : (j, +, k)$

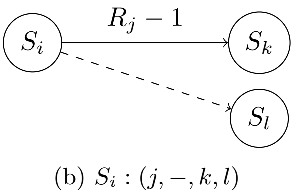
We can then use these to describe programs:

For any $k > 0$ a program P ***halts*** on input $(m_1, m_2, \ldots, m_k) \in \mathbb{N}^k$ if it ever reaches state S_0, written $P(m_1, \ldots, m_k) \downarrow$
The ***halting set*** $\Omega(P)$ is a set of inputs on which P halts.

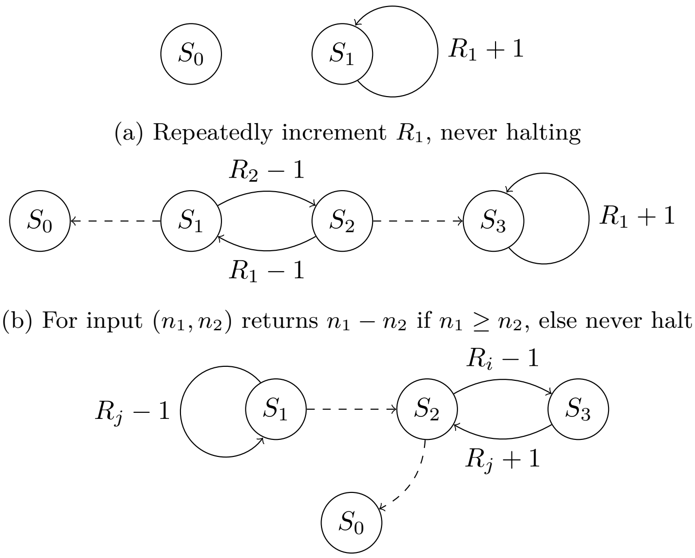
$$\Omega(P) = \cup_{k>0}\{(m_1, \ldots, m_k) : P(m_1, \ldots, m_k) \downarrow\}$$



(a) $S_i : (j, +, k)$



(b) $S_i : (j, -, k, l)$



(a) Repeatedly increment R_1, never halting

(b) For input (n_1, n_2) returns $n_1 - n_2$ if $n_1 \geq n_2$, else never halt

(c) Transfer R_i to R_j, emptying R_i

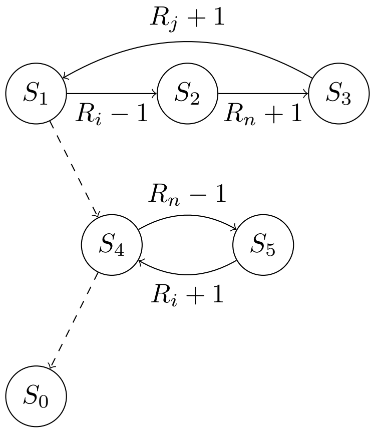
If P does not halt, we write $P(m_1, \ldots, m_k) \uparrow$.

For each program P, the ***upper register index*** $\text{Upper}(P)$ is the largest index of a register appearing in the instructions for P. So if $i > \text{Upper}(P)$ then R_i never changes.

A ***partial function*** $f : \mathbb{N}^k \to \mathbb{N}$ is one where the domain of f is a subset of $\mathbb{N}^k$, and undefined otherwise. If f is defined everywhere then we call it a ***total function***. This lets us define these programs as functions - we say f is ***partial computable*** by a program P such that $\forall (m_1, \ldots, m_k) \in \text{dom}(f)$ have $P(m_1, \ldots, m_k) \downarrow$ with $f(m_1, \ldots, m_k) = R_1$ on halting, and $\forall (m_1, \ldots, m_k) \notin \text{dom}(f)$ we have $P(m_1, \ldots, m_k) \uparrow$. Hence any program P and $k > 0$ gives a partial function $f : \mathbb{N}^k \to \mathbb{N}$.

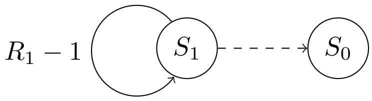
Lemma 1.1. *We can add R_i to R_j leaving R_i unchanged.*

Proof.



Thus by setting $(i = 2, j = 1)$ we see that $(n_1, n_2) \mapsto n_1 + n_2$ is total computable. □

We have already seen that the function $n \mapsto 0$ is also computable. This can be done with the machine:



Corollary 1.2. *There exists a routine which can copy R_i to R_j leaving R_i unchanged.*

Proof. First empty R_j, then use **1.1** to add R_i to R_j. □

We can use these as subroutines to join with other programs P. Use registers R_n s.t. $n >$ Upper(P) and largest input register. Then replace the halt state of P with the start state of the subroutine. In fact we have already done this - if you look carefully at the adding machine, you can see that the middle section is the same as the machine in (c) of the examples - this is the part where we replace the value in R_i from its temporary location in R_n.

Partial Recursive Functions

Partial computable functions have good closure properties.

Theorem 1.3.

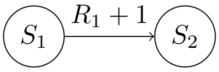
1. *For $i \leq k$, the* **projection function** $(n_1, \ldots, n_k) \mapsto n_i$ *is computable.*
2. *The zero function $n \mapsto 0$ and* **successor function** $n \mapsto n + 1$ *are computable*
3. *(Composition) If $f : \mathbb{N}^k \to \mathbb{N}$ and $g_1, \ldots, g_k : \mathbb{N}^l \to \mathbb{N}$ are all partial computable then so is the composition function $h(n_1, \ldots, n_l) = f(g_1(n_1, \ldots, n_l), \ldots, g_k(n_1, \ldots, n_l))$ where defined. If $f, g_1, \ldots, g_k$ are total functions, so is h.*
4. *(Recursion) If f on k variables and g on $k+2$ variables are partial computable, then so is the partial function $h : \mathbb{N}^{k+1} \to \mathbb{N}$ defined inductively as:*

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

   Moreover, f, g total $\implies$ h total.
5. *(Minimisation) If f on $k+1$ variables is partial computable then so is the partial function $g : \mathbb{N}^k \to \mathbb{N}$ defined by $g(n_1, \ldots, n_k) = n$ if $f(n_1, \ldots, n_k, n) = 0$ and $f(n_1, \ldots, n_k, m) > 0$ for all $m < n$, and is undefined if no zero is ever found. Note that f total $\not\Rightarrow$ g total.*

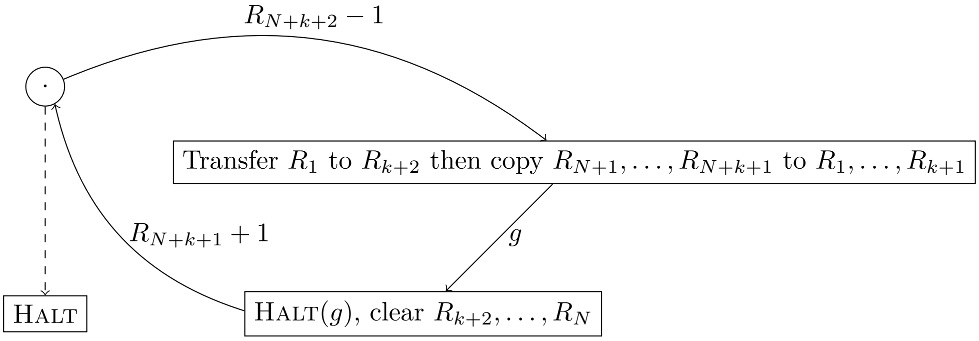
Proof.

1. We can use the program Transfer R_i to R_1, Halt.
2. Zero function has already been seen. For successor function, use:



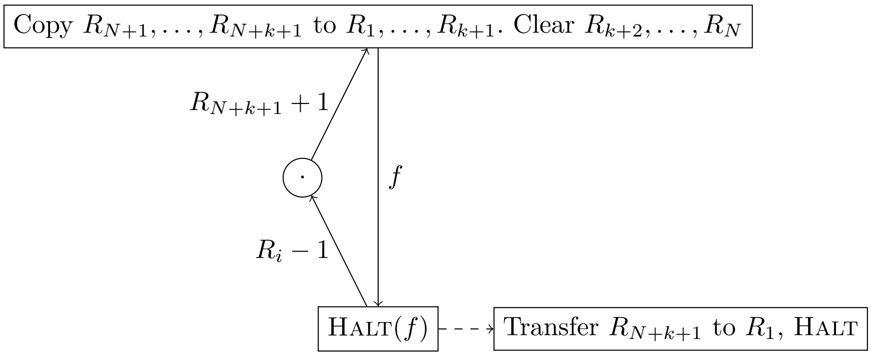
3. First transfer $R_1, \dots, R_l$ to $R_{N+1}, \dots, R_{N+l}$ where N is large enough to not be needed in other subroutines. Then for each $1 \leq i \leq k$ in turn, copy $R_{N+1}, \dots, R_{N+l}$ to $R_{k+1}, \dots, R_{k+l}$, perform g_i but with all registers shifted up by k and then transfer answer from R_{k+1} to R_i, then clear $R_{k+2}, \dots, R_N$. Finally, apply f.

4. Copy $R_1, \dots, R_k$ to $R_{N+1}, \dots, R_{N+k}$, transfer R_{k+1} to R_{N+k+2} ("counts down"), then do f. Then:



$R_{N+k+2} - 1$
Transfer R_1 to R_{k+2} then copy $R_{N+1}, \dots, R_{N+k+1}$ to $R_1, \dots, R_{k+1}$
g
$R_{N+k+1} + 1$
HALT
HALT(g), clear $R_{k+2}, \dots, R_N$

5. Copy $R_1, \dots, R_k$ to $R_{N+1}, \dots, R_{N+k}$. Then



Copy $R_{N+1}, \dots, R_{N+k+1}$ to $R_1, \dots, R_{k+1}$. Clear $R_{k+2}, \dots, R_N$
$R_{N+k+1} + 1$
f
$R_i - 1$
HALT(f)
Transfer R_{N+k+1} to R_1, HALT

□

The class of ***partial recursive functions*** is the smallest class of partial functions from $\mathbb{N}^k$ to $\mathbb{N}$ over all $k \geq 1$ closed under the operations **1.3** (1) to (5). That is, f can be constructed from basic functions and applications of (3), (4), (5) a finite number of times.

So **1.3** says that partial recursive $\implies$ partial computable.

A partial function is ***primitive recursive*** if we never use **1.3** (5) its construction. Note that primitive recursive $\implies$ total recursive, as (5) was the only construction that breaks the totality of the function. [The converse implication is not true: the Ackermann function.]

Example: + and × are primitive recursive:

+: Let $h(m, 0) = m$, $h(m, n+1) = h(m, n) + 1 = g(m, n, h(m, n))$, where $g(x, y, z) = z + 1$.

×: $H(m, 0) = 0$, $H(m, n+1) = H(m, n) + m = g(m, n, H(m, n))$ for $g(x, y, z) = x + z$.

Example: $(m, n) \mapsto m^n$ is primitive recursive - left as exercise.

We need to be able to "encode" finite sequences of arbitrary length in $\mathbb{N}$. For $n > 0$ and $i \in \mathbb{N}$, write p_i for the $(i+1)^{\text{th}}$ prime (so $p_0 = 2$). Write $(n)_i$ for the largest power of the prime p_i that divides n.

Lemma 1.4. *For each fixed i, the 1 variable function $(\cdot)_i : \mathbb{N} \to \mathbb{N}$ is primitive recursive.*

Proof. First note that, for any finite sequence $(m_0, m_1, \ldots, m_s) \subseteq \mathbb{N}^{s+1}$, the function $f(n) = \begin{cases} m_n & n \leq s \\ 0 & n > s \end{cases}$ is primitive recursive.

By induction on s and recursion from **1.3** *(4)*, for $k = 0$ if c constant and $g : \mathbb{N}^2 \to \mathbb{N}$ is primitive recursive, then so is $h(0) = c, h(n+1) = g(n, h(n))$.

Thus, given $f : \mathbb{N} \to \mathbb{N}$ primitive recursive, let $g(n,m) := f(n)$, which is primitive recursive. So $h(0) = c, h(n+1) = f(n)$ is primitive recursive, and we can repeat this process.

This includes for each fixed k:

1. The step function $\text{Step}_k(n) = \begin{cases} 1 & 0 \leq n \leq k-1 \\ 0 & \text{otherwise} \end{cases}$
2. The delta function $\delta_k(n) = \begin{cases} 1 & n = k \\ 0 & n \neq k \end{cases}$ Let $\epsilon(n) = \delta_0(\delta_0(n)) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \end{cases}$ - this is also primitive recursive.
3. The slope function $\text{Slope}_k(n) = \begin{cases} n+1 & 0 \leq n \leq k-2 \\ 0 & \text{otherwise} \end{cases}$
4. The remainder function $\text{Rem}_k(n) = n \mod k$ use recursion in the form $g(n,m) := f(m)$, so $h(0) = 0$, $h(n+1) = f(h(n))$ primitive recursive if f is. Here, $\text{Rem}_k(n+1) = \text{Slope}_k(\text{Rem}_k(n))$

*5. $\text{Floor}_k(n) = \lfloor \frac{n}{k} \rfloor$

*6. $\text{Divide}_k(n) = \begin{cases} n/k & n \equiv 0 \mod k \\ 0 & \text{otherwise} \end{cases}$

*7. Division by powers $\text{Power}_k(n,m) = \begin{cases} n/k^m & n \equiv 0 \mod k^m \\ 0 & \text{otherwise} \end{cases}$

*8. $\text{Maxpower}_k(n) = \begin{cases} 0 & n = 0 \\ \text{largest power of k dividing n} & n \neq 0 \end{cases}$

Proofs of *ed function are on example sheet 1.

Now define by recursion $h(n,0) = 0$ and $h(n,m+1) = h(n,m) + \epsilon(\text{Power}_k(n,m+1))$.
$\epsilon(\text{Power}_k(n,j)) = \begin{cases} 1 & k^j \text{ divides } n > 0 \\ 0 & \text{otherwise} \end{cases}$, so is 0 if $j \geq n$

So $h(n,n) = \sum_{i=1}^{n} \epsilon(\text{Power}_k(n,1)) = \text{Maxpower}_k(n)$, so $h(n,n)$ is primitive recursive. □

Computable = Recursive

We have seen already that partial recursive $\implies$ partial computable.

Theorem 1.5. *Every partial computable function $f : \mathbb{N}^k \to \mathbb{N}$ is partial recursive.*

Proof. From a program P for f, define $g : \mathbb{N}^{k+2} \to \mathbb{N}$, "what actually goes on in P", to be the function:

$$g(n_1, \dots, n_k, 0, t) \text{ is the state of } P \text{ after time } t \text{ with input } (n_1, \dots, n_k)$$

So $t = 0$ gives 1 and if halt at t_0 then gives 0 for all $t \geq t_0$, and:

$$(n_1, \dots, n_k, i, t) \text{ is the contents of } R_i \text{ at time } t$$

So have N (assume $> k$) such that $g(\cdots, i, \cdot) = 0 \forall i > N$. Note that g is a total function.

Suppose that g is recursive and define $q(n_1, \dots, n_k) = \min\{t : g(n_1, \dots, n_k, 0, t) = 0\}$. Then q is partial recursive, and so $f(n_1, \dots, n_k) = g(n_1, \dots, n_k, 1, q(n_1, \dots, n_k))$ is partial recursive.

Proof that g is recursive:
Fix $n_1, \dots, n_k$ and t. For each $0 \leq i \leq N$, g gives $(g_0, \dots, g_N) \in \mathbb{N}^{N+1}$, encode as $c(d_0, \dots, d_N) = 2^{d_0} 3^{d_1} \dots p_N^{d_N} \in \mathbb{N}$ is primitive recursive. Also, $(c(d_0, \dots, d_N))_i = d_i$ is primitive recursive. We will define $h : \mathbb{N}^{k+2} \to \mathbb{N}$ via recursion where $h(n_0, n_1, \dots, n_k, t)$ is the coded integer of state and registers of P at time t for input $n_1, \dots, n_k$ and start state n_0 (here $= 1$).

In particular, for $t = 0, h = 2^{n_0} 3^{n_1} \dots p_k^{n_k}$. For recursion for h, we need $s : \mathbb{N} \to \mathbb{N}$, the "transition function", which computes in coded form the changes at each step. □

Algorithms and Recursive Sets

A function $f : \mathbb{N}^k \to \mathbb{N}$ is ***recursive*** or ***computable*** if it is total. If it s not even partial recursive, then it is ***incomputable***.

A subset $X \subseteq \mathbb{N}^k$, (often $X \subseteq \mathbb{N}$) is ***recursive*** or ***computable*** or ***decidable*** if the characteristic function $\chi_X(n) = \begin{cases} 1 & n \in X \\ 0 & n \notin X \end{cases}$ is computable, i.e. if we can program a computer to tell us if a given number is in it or not.

An ***algorithm*** is any process which takes an input in $\mathbb{N}^k$ for some specified k, or a recursive subset $X \subseteq \mathbb{N}^k$, and returns an output in $\mathbb{N}$ which is simulated by a register machine.

A ***total algorithm*** terminates for all elements in X, whilst a ***partial algorithm*** may fail to terminate for some choices of input.

Lemma 1.6. *For each $k \geq 1$, there is some total function $f : \mathbb{N}^k \to \mathbb{N}$ which is incomputable.*

Proof. Each computable program comes from a finite program with $n+1$ states for some n. Since there are only countably many finite programs, but $\mathcal{P}(\mathbb{N})$ is uncountable, hence one of these sets is not computable. Then its indicator function is not computable. □

For given m, the ***shortlex*** ordering on $\mathbb{N}^m$ is $(n_1, \dots, n_m) < (n'_1, \dots, n'_m)$ if $\sum n_i < \sum n'_i$ or $\sum n_i = \sum n_i$ and there is some j with $n_i = n'_i$ for $i < j$, but $n_{j+1} > n'_{j+1}$.

This gives us a bijection to $\mathbb{N}$, as there are only finitely many k-tuples of naturals with sum less than $N \in \mathbb{N}$.

If a register machine P, then for the i^{th} instruction, let $t_i = \begin{cases} 2^j \cdot 5^k & \text{if it is } (j, +, k) \\ 2^j \cdot 3 \cdot 5^k \cdot 7^l & \text{if it is } (j, -, k, l) \end{cases}$. We can then encode the tuple $(t_1, t_2, \dots, t_n)$ as $m = 2^n \cdot 3^{t_1} \cdot 5^{t_2} \cdot \dots \cdot p_n^{t_{n-1}}$. We denote the program encoded by the number m as P_m, if m is a valid encoding of a program. For these m, we say m ***codes*** a program, and P_m is the ***mth machine***.

The input for a register machine is a k-tuple for varying k, so we define $f_{n,k}$ for the k-variable function computed by the n^{th} machine if P_n exists.

Here is an explicit total function which is not recursive:

Lemma 1.7. *Consider the following function $g : \mathbb{N} \to \mathbb{N}$ given by:*

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

Then g is not recursive.

Proof. If g is recursive then it is computed by some machine. So there exists an N such that $g = f_{N,1}$ is total. But then $f_{N,1}(N) = g(N) = f_{N,1}(N) + 1 \lightning$. □

Church's Thesis

The two key figures in this chapter are Alonzo Church and Alan Turing, doing this work around 1936.

An ***executable process*** is a step-by-step deterministic process with finite description at each step, a finite set of rules, and a finite amount of input and output.

An ***abstract theory of finite computation*** is a theory of computation consisting of these executable processes.

Theorem 1.8 (Church's Thesis)**.**

1. *In any abstract theory of finite computation, $\mathscr{C}$, the $\mathscr{C}$-partial computable function $f : \mathbb{N}^k \to \mathbb{N}$ gives at most the partial recursive functions.*
2. *Any informal description of an executable process starting with input in $\mathbb{N}^k$ and output in $\mathbb{N}$ or never halting is equivalent to a register machine, so we don't need to worry about all the details of the machine.*
3. *There is a total algorithm, that, given the encoding (e.g. shortlex) of a description of an algorithm, returns a code for a register machine that carries out this process.*

This is not so much one theorem as many different independent theorems. It has however been proven that all the following abstract theories of finite computation are equivalent:

- Church's λ-calculus
- Turing machines

- Register machines
- Standard languages
- Quantum/DNA-computers

From now on, we will refer to these three statements as "Church's thesis" or even just "Church".

Lemma 1.9. *Let $h : \mathbb{N} \to \mathbb{N}$ be:*

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

Then h is partial recursive.

Proof. For input n, check if n codes a program - this is total recursive. If so, run the program with input n. If it then halts, add 1 to R_1 and halt, and so by Church h is partial computable = partial recursive. □

Recursively Enumerable Sets

Given a partial recursive function $f : \mathbb{N}^k \to \mathbb{N}$ with domain $X \subseteq \mathbb{N}^k$, suppose we input 7. If $f(7)\downarrow$, then we can run the machine and get the answer within finite time. However, if $f(7)\uparrow$, then we will be waiting forever. By the halting problem, there is no way to know in advance what will happen.

Now suppose we ask: "Does f halt on either 7 or 9?" Then the answer is yes, but if we are unlucky and start with 9, we will never know if we naïvely compute $f(9)$, then $f(7)$.

So instead, we zig-zag: we do one step of each alternately.
Let $P_n^t(x)$ be the t^{th} step of P_n with input x. Then we can do:

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

We can clearly extend this to any set of finite size. We can even do infinite sets, by following a path similar to in the textbook enumeration of $\mathbb{Q}$:

$$\begin{tikzcd}
P_n^1(1) \arrow[r] & P_n^1(2) \arrow[dl] & P_n^1(3) \arrow[d] \\
P_n^2(1) \arrow[d] & P_n^2(2) \arrow[ur] & P_n^2(3) \arrow[dl, dotted] \\
P_n^3(1) \arrow[ur] & {} &
\end{tikzcd}$$

We can even alternate between different machines, and this process can be extended to any countable set. Then by Church, we can write a program that returns 1 for input $x \in \mathbb{N}^k, k < \infty$ if some partial recursive function f halts on input of x.

We say a set $E \subseteq \mathbb{N}^k$ is ***recursively enumerable*** if the function

$$\phi_E(n) := \begin{cases} 1 & n \in E \\ \uparrow & \text{else} \end{cases}$$

The idea behind this definition is that, compared to a recursive set, here we can only say that x is in E, whereas in a recursive set we can say if $x \in E$ or $x \notin E$. A consequence of this is that, by applying the above process for ϕ_E on all of $\mathbb{N}^k$, we will eventually get out all the elements of E, but we will not know when this has happened (indeed, it might take infinitely long to get all the elements, but we will eventually be notified that any given element is in E), so we can "recursively enumerate" E. Conversely, recursive sets are often called ***decidable*** - we can always decide whether or not $x \in E$. We will cement this discussion into a few theorems now:

Theorem 1.10. *The following are equivalent for any subset $E \subseteq \mathbb{N}$:*

1. *E is the range of some partial recursive function on some number of variables, i.e.* $E = \{f_{n,k}(\mathbf{x}) : \mathbf{x} \in \mathbb{N}^k\}$
2. *E is the domain of definition of some partial recursive function on* 1 *variable, i.e.:* $\exists\ n$ *s.t.* $E = \{m \in \mathbb{N} : f_{n,1}(m) \downarrow\}$
3. *E is recursively enumerable*
4. *The function* $\psi_E(n) = \begin{cases} n & n \in E \\ \uparrow & \textit{otherwise} \end{cases}$ *is partial recursive. i.e. we can enumerate the elements of E in a recursive fashion.*

Proof.

- *2.* $\Longrightarrow$ *3.* Given a program P_n for computing $f_{n,1}$, modifying at end so it empties R_1 then adds 1.
- *3.* $\Longrightarrow$ *4.* Similarly, we start by copying R_1 to R_N for large enough N, then at the end empty R_1 and copy R_N to R_1.
- *4.* $\Longrightarrow$ *1.* Immediate as $f_{n,k} = \psi_E$ for some choice of n, $k = 1$.
- *1.* $\Longrightarrow$ *2.* Given a program P_n for computing $f_{n,k} : \mathbb{N}^k \to \mathbb{N}$ with range E, do the following process Q by Church:

  Given an input $x \in \mathbb{N}$ for Q, run a zig-zag procedure having ordered $\mathbb{N}^k$ to compute $f_{n,k}(\cdot)$ for each input $\cdot \in \mathbb{N}^k$. Each time, $f_{n,k}(m_1, \dots, m_k)$ halts, run the total subroutine comparing the output to x wand halt with output 1 if equal. If not, return to the zig-zag process. So Q will return 1 if x is in the range $f_{n,k}$, and if not will run forever, because we would never get an output equalling x.

□

Properties of Recursively Enumerable Sets

In **1.10**(*1.*), can we have a total recursive function?

Theorem 1.11. *If $E \neq \emptyset$ and $E \subseteq \mathbb{N}$, then E is recursively enumerable if and only if E is the range of a total recursive function $f : \mathbb{N} \to \mathbb{N}$.*

Proof. For $E = \{e_0, \dots, e_{k-1}\}$ finite, define $f(n) := \begin{cases} e_n & n < k \\ e_{k-1} & \text{otherwise} \end{cases}$. This is total recursive.

In the case of infinite $E \subseteq \mathbb{N}$ which is the domain of a partial recursive function $g : \mathbb{N} \to \mathbb{N}$, run a zig-zag process to perform the steps of computing $g(0), g(1), \dots$. Define a total function $f : \mathbb{N} \to \mathbb{N}$ as $f(0)$ is the first input number i for which the computation of $g(i)$ halts in this process (not necessarily the smallest), $f(1)$ the second, and so on. Thus the range of f is E and f is recursive by Church. □

So each program P_n corresponds to a recursively enumerable set which is the domain of $f_{n,1}$. Thus let $W_n = \{x \in \mathbb{N} : f_{n,1}(x) \downarrow\}$ if n encodes a program, and the empty set otherwise. We say W_n is the ***n*th *recursively enumerable set***, though the labelling is not unique, i.e. it could be that $W_n = W_m$ for some $n \neq m$.

Every recursive subset is recursively enumerable by definition. However:

Theorem 1.12. *For any $E \subseteq \mathbb{N}$, E is recursive if and only if E and $\mathbb{N} \setminus E$ is recursively enumerable.*

Proof.

- $\underline{\implies}$ As χ_E is total recursive, ϕ_E is partial recursive because when χ_E halts, if the output is 1 then halt, otherwise enter an infinite loop. We can do the same for $\mathbb{N} \setminus E$, to get $\phi_E, \phi_{\mathbb{N} \setminus E}$.
- $\underline{\impliedby}$ As ϕ_E and $\phi_{\mathbb{N} \setminus E}$ are both partial recursive, we can do a zig-zag process between the two. Then, if $\phi_E(x)$ halts, return 1, otherwise if $\phi_{\mathbb{N} \setminus E}(x)$ halts, return 0. Since $E \cup \mathbb{N} \setminus E = \mathbb{N}$, exactly one of these will halt, and so this gives χ_E which is total recursive.

□

Theorem 1.13.

1. *If $I \subseteq \mathbb{N}$ is recursively enumerable, then the union $\cup_{n \in I} W_n$ is recursively enumerable.*
2. *If $J \subseteq \mathbb{N}$ is finite, then $\cap_{n \in J} W_n$ is recursively enumerable.*

Proof.

1. We want a process that, given an input $x \in \mathbb{N}$, halts if $x \in \cup_{n \in I} W_n$, and not if $x \notin \cup_I W_n$.

   By a big zig-zag process, start outputting elements n of I, test if n codes a program P_n, move on if not but if so, start to run P_n on this input x. One of these subprocesses halts if and only if $x \in W_n$ for some $n \in I$, in which case halt the whole process. However, if $x \notin \cup_I W_n$, then the process will run forever.
2. First check all the $j \in J$ code a program, else we have $\emptyset$. Then run these $|J|$ programs in parallel, each with input x. If/when all of these halt, then halt. Then we halt if and only if $x \in \cap_J W_n$.

□

Universality and Undecidability

Turing's paper was titled "On computable numbers, with an application to the Entscheidungsproblem", published in the Proceedings of the London Mathematical Society 42 (1936) pp. 230-265.

Theorem 1.14 (Existence of universal programs)**.** *There exists a partial recursive function $u : \mathbb{N}^3 \to \mathbb{N}$ such that:*

$$u(n,k,m) = \begin{cases} r & n \text{ codes a program and } m \text{ codes a } k\text{-tuple and } f_{n_k}((m)_1, \ldots, (m)_k) = r \\ \uparrow & \text{otherwise} \end{cases}$$

Proof. On input, check if n codes a program. If not, loop forever. If so, decode m and see if it is a k-tuple, else loop forever. If so, then run the program defined by n on input $((m)_1, \ldots, (m)_k)$. If it halts, output r, the contents of R_1, otherwise just leave it running. □

There exist subsets of $\mathbb{N}$ which are recursively enumerable but not recursive. We will give a specific example of one: the ***halting set*** or $\mathbb{H}$.

$$\mathbb{H} \coloneqq \{(n,m) : n \text{ codes a program and } f_{n,1}(m) \downarrow\} \subseteq \mathbb{N}^2$$

We also define:

$$\mathbb{K} \coloneqq \{n : n \text{ codes a program and } f_{n,1}(n) \downarrow\} \subseteq \mathbb{N}$$

Observe $\mathbb{K} \times \mathbb{K} \subseteq \mathbb{H}$.

Theorem 1.15 (Undecidability of the Halting Problem)**.** *The set $\mathbb{K}$ is recursively enumerable, but $\mathbb{K}$ and $\mathbb{H}$ are not recursive.*

Proof. To see that $\mathbb{K}$ is recursively enumerable, given $n \in \mathbb{N}$ check if it codes some program P_n, else go into a loop. If it does, start the computation of $f_{n,1}(n)$, which will halt if and only if $n \in \mathbb{K}$.

However, $\mathbb{N} \setminus \mathbb{K}$ is not recursively enumerable. If it were, then there would exists some N such that $W_N \coloneqq \{x \in \mathbb{N} : f_{N,1}(x) \downarrow\} = \mathbb{N} \setminus \mathbb{K}$. Now we ask where N is - $N \in K \iff f_{N,1}(N) \downarrow \iff N \in W_N \iff N \in \mathbb{N} \setminus \mathbb{K}$ ↯.

So $\mathbb{K}$ is not recursively enumerable, i.e. $\chi_{\mathbb{K}}$ is not recursive. now suppose that $\chi_{\mathbb{H}} : \mathbb{N}^2 \to \mathbb{N}$ is a recursive function. Then $\chi_{\mathbb{H}}(n,n) = \chi_{\mathbb{K}}(n)$ is recursive ↯. □

Reductions

Given $A, B \subseteq \mathbb{N}$, ***a many-one reduction of A to B*** is a total recursive function $f : \mathbb{N} \to \mathbb{N}$ such that:

$$\forall n \in \mathbb{N}, n \in A \iff f(n) \in B \text{ (i.e. } A = f^{-1}(B))$$

If so, we say that A ***many-one reduces*** to B and write $A \leq_m B$.

Note that $A \leq_m B \iff A^c \leq_m B^c$, using the same function.

Lemma 1.16. *Suppose $A \leq_m B$. Then:*

1. *B recursively enumerable $\implies$ A recursively enumerable.*

2. B recursive $\implies$ A recursive.

Proof.

1. b recursively enumerable implies we have a partial recursive function g with domain B, so $A = \operatorname{dom} g \circ f$ which is partial recursive, and hence A is recursively enumerable.

2. We have by *1.* that A is recursively enumerable. We have B^c recursively enumerable and so A^c recursively enumerable, and hence A is recursive.

□

Suppose we have some partial recursive function h on $m + k$ variables, so there exists n with $h = f_{n,m+k}$. Fix the first m inputs of h as $a_1, \ldots, a_m$, and then let:

$$g(x_1, \ldots, x_k) \coloneqq f(a_1, \ldots, a_m, x_1, \ldots, x_k)$$

Then this is partial recursive by composition, and so there exists N with $g = f_{N,k}$. We show that N depends on a nice way on $(a_1, \ldots, a_m)$.

Theorem 1.17 (Kleene's $s-m-n$ theorem)**.** *For all $m, k > 0$, a partial function $h : \mathbb{N}^{m+k} \to \mathbb{N}$ is partial recursive if and only if there is a total recursive function $\tau : \mathbb{N}^m \to \mathbb{N}$ such that, for all $(a_1, \ldots, a_m, x_1, \ldots, x_k) \in \mathbb{N}^{m+k}$, we have:*

$$h(a_1, \ldots, a_m, x_1, \ldots, x_k) = f_{\tau(a_1,\ldots,a_m),k}(x_1, \ldots, x_k)$$

Proof. Given an input for h, first compute τ which does halt with output M, say, and then run program P_m in input $(x_1, \ldots, x_k)$. If this halts, this the output of h, so h is partial computable, and hence partial recursive.

For the other direction, given $(a_1, \ldots, a_m) \in \mathbb{N}^m$, and h partial recursive so partial computable, so we have a program P which computes h on $m+k$ variables. To describe a program $P_{(a_1,\ldots,a_n)}$ which computes the function $h(a_1, \ldots, a_m, \cdot, \ldots, \cdot) = P_{(a_1,\ldots,a_m)}(\cdot, \ldots, \cdot)$, take instructions for P but first move the contents of $R_1, \ldots, R_k$ to $R_{m+1}, \ldots, R_{m+k}$, then load the fixed $a_1, \ldots, a_m$ into registers $R_1, \ldots, R_m$, then run P. So, given any $(a_1, \ldots, a_m) \in \mathbb{N}^m$, we have described a process to obtain a code $\tau(a_1, \ldots, a_m)$ for the function $P_{(a_1,\ldots,a_m)}$. Now τ is total and, by Church, recursive. □

Theorem 1.18. *A set $X \subseteq \mathbb{N}$ is recursively enumerable if and only if $X \leq_m \mathbb{K}$.*

Proof. The if part is immediate by **1.15**. For the only if, take X recursively enumerable and define $h : \mathbb{N}^2 \to \mathbb{N}$ as $h(x, n) = \begin{cases} 1 & x \in X \\ \uparrow & \text{otherwise} \end{cases}$. So h partial recursive. By (s-m-n) we have a total recursive function $\tau : \mathbb{N} \to \mathbb{N}$ with $h(x, n) = f_{\tau(x),1}(n)$. Now $x \in X$ iff $h(x, \tau(x)) \downarrow$, which happens iff $f_{\tau(x),1}(\tau(x)) \downarrow$, i.e. $\tau(x) \in \mathbb{K}$, and so $X \leq_m \mathbb{K}$. □

Theorem 1.19 (The Recursion Theorem)**.** *For each total recursive function $g : \mathbb{N} \to \mathbb{N}$, there is some $n_0 \in \mathbb{N}$ with $f_{n_0,1} = f_{g(n_0),1}$ as partial functions.*

Proof. For a universal program u, define a partial function $h : \mathbb{N}^2 \to \mathbb{N}$ as

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

By (s-m-n) we have a total recursive function $\tau : \mathbb{N} \to \mathbb{N}$ such that $\tau = f_{m,1}$ for some $m \in \mathbb{N}$, with $h(x, y) = f_{f_{m,1}(x),1}(y)$. Our "fixed point" n_0 will be $f_{m,1}(m)$, as then, if $f_{n_0,1}(y) \downarrow$, we have $f_{n_0,1}(y) = h(m, y) = u(g(u(m, 1, 3^m)), 1, 3^y) = u(g(f_{m,1}(m)), 1, 3^y) = u(g(n_0), 1, 3^y) = f_{g(n_0),1}(y)$ as desired. If $f_{n_0,1}(y)$ does not halt, then neither will $f_{g(n_0),1}(y)$, and so $f_{n_0,1} = f_{g(n_0),1}$ as partial functions. □

Rice's Theorem

We consider properties of recursively enumerable sets $X \subseteq \mathbb{N}$. If $\mathcal{S} \subseteq \mathcal{P}(\mathbb{N})$ is the collection of all recursively enumerable sets, then a ***property*** of recursively enumerable sets is a map $\rho : \mathcal{S} \to \{0, 1\}$, e.g. being infinite/empty/containing a prime.

Given a property ρ we'd like a process to tell us if a given recursively enumerable set X has ρ, namely a machine with a 1-variable input n, returning 1 if the n^{th} recursively enumerable set W_n has ρ. The constant maps $\rho \equiv 0$ and $\rho \equiv 1$ are the ***trivial*** properties, otherwise ρ is non-trivial.

Theorem 1.20 (Rice's Theorem)**.** *Let ρ be a non-trivial property of recursively enumerable sets and let:*

$$I_\rho \coloneqq \{n \in \mathbb{N} : n \textit{ codes a program and } \rho(W_n) = 1\} \subseteq \mathbb{N}$$

Then if $\rho(\emptyset) = 0$ we have $\mathbb{K} \subseteq_m I_\rho$, and if $\rho(\emptyset) = 1$ then $\mathbb{K} \subseteq_m \mathbb{N} \setminus I_\rho$.

Proof. If $\emptyset$ does not have ρ (i.e. $\rho(\emptyset) = 0$) then take any recursively enumerable non-empty set E which does have ρ, and define a partial recursive function $h : \mathbb{N}^2 \to \mathbb{N}$ as:

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

By (s-m-n) we get a total recursive function $\tau : \mathbb{N} \to \mathbb{N}$ such that $h(n, x) = f_{\tau(n),1}(x)$. Now $n \in \mathbb{K}$ implies that $f_{\tau(n),1} = \phi_E$, $W_{\tau(n)} = E$, and so $W_{\tau(n)}$ has ρ. $n \notin \mathbb{K}$ implies that $W_{\tau(n)} = \emptyset$ implies $W_{\tau(n)}$ does not have ρ, and so $\mathbb{K} \leq_m I_\rho$ via τ.

If $\emptyset$ has ρ and there is some recursively enumerable set E without ρ. Take the same h and argument as before to get that $n \in \mathbb{K} \iff W_{\tau(n)}$ does not have ρ. So $\mathbb{K} \leq_m \mathbb{N} \setminus I_\rho$ via τ. □

Corollary 1.21. *If ρ is a non-trivial property of recursively enumerable sets, then I_ρ is non-recursive/*

Proof. Suppose I_ρ is recursive. Then so is $\mathbb{N} \setminus I_\rho$. As $\mathbb{K} \leq_m I_\rho or \mathbb{N} \setminus I_\rho$, and so $\mathbb{K}$ is recursive $\lightning$. □

So in general, we cannot decide *any* non-trivial property of recursively enumerable sets. Of course you might for some inputs, but cannot do it algorithmically for arbitrary inputs.

2 Regular Languages and Finite-State Automata

Deterministic Finite State Automata

We've seen the most general type of computing, but it did require handling arbitrary integers. We now consider a weaker by more practical theory.

An ***alphabet*** Σ is a finite non-empty set - think of the elements as ***symbols***, or even ***letters***.

A ***word*** or ***string*** of length n over Σ is a collection of n letters from Σ. We denote the set of all words over Σ of length n by Σ^n.

We denote the ***empty word*** as ϵ (at least if ϵ is not a letter of Σ), and so $\Sigma^0 = \{\epsilon\}$.

Let $\Sigma^* = \cup_{n \in \mathbb{N}} \Sigma^n$ including ϵ be all words over Σ.

A ***language*** L over Σ is any subset of Σ^*. For instance, $\Sigma = \{a, b, \ldots, z\}, L = \{$all words in your dictionary$\}$.

If $\Sigma = \{0\}$ then we can think of any $L \subseteq \mathbb{N} = \Sigma^*$.

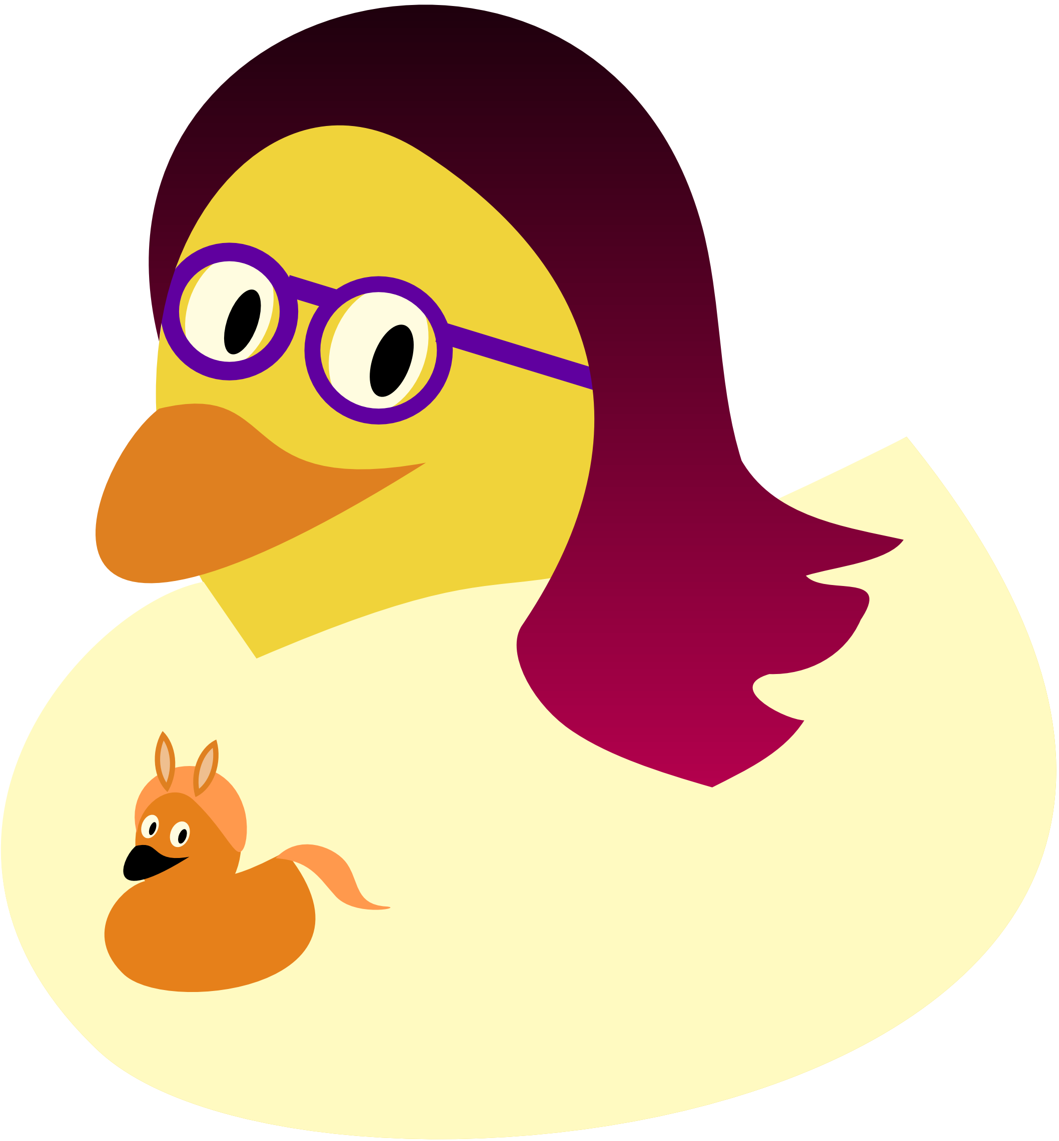
We can now give a computational method to determine some languages:
A ***deterministic finite-state automaton (DFA)*** is a structure $D = (Q, \Sigma, \delta, q_0, F)$ consisting of:

1. A finite non-empty set of states Q
2. A finite non-empty alphabet Σ
3. A transition function $\delta : Q \times \Sigma \to Q$
4. A start state $q_0 \in Q$
5. A subset $F \subseteq Q$ of accept states

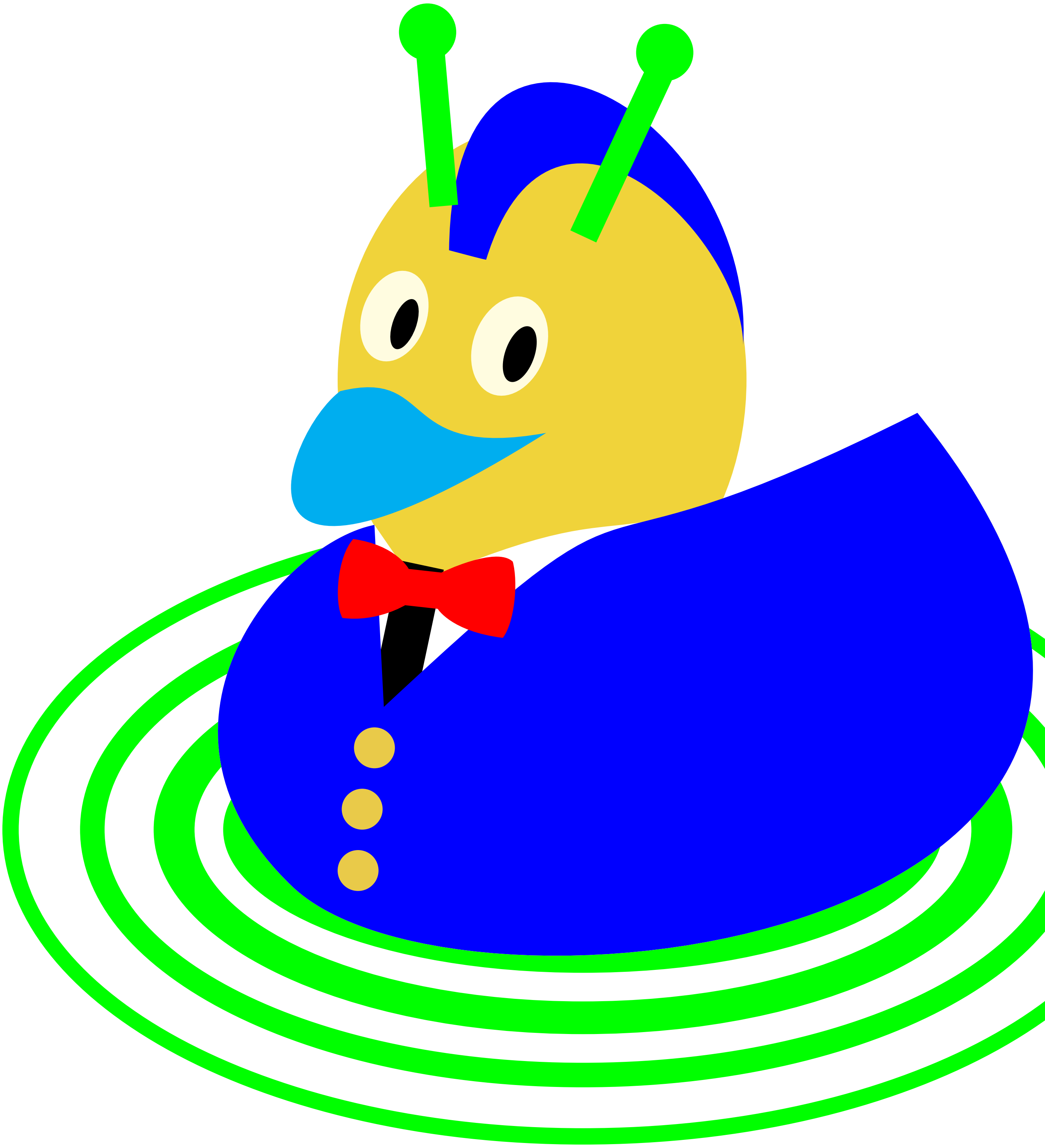
The idea is that an input to a DFA D over Σ is any word $w \in \Sigma^*$. The DFA starts in state q_0 and reads the 1$^{\text{st}}$ letter x_1 of w. It moves to state $\delta(q_0, x_1)$ and reads x_2, then moves to state $\delta(\delta(q_0, x_1), x_2)$, etc.

When all of w is read, if the DFA is in an accept state then w is accepted. Otherwise it is rejected.

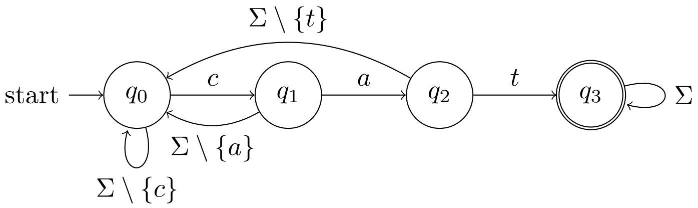
Duck Interlude



Wendi



A ***transition diagram*** for a DFA, D is a directed graph such that the vertices are states in Q, and for each $(q,\sigma) \in Q \times \Sigma)$, draw a directed edge from q to $\delta(q,\sigma)$ and label this as σ. Then add an arrow from nowhere to q_0 and label it start. We draw a vertex in a double circle if it is an accept state. For instance, $\Sigma = \{a,\dots,z,\_\}$, the DFA corresponding to the regex /cat/ is:



A ***transition table*** for a DFA is a table with symbols in Σ against states in q where the $(i,j)^{\text{th}}$ entry is $\delta(q_i,\sigma_j) \in Q$.

Regular Languages

We describe a way to "follow" a word through a DFA. The ***extended transition function*** of a DFA D is $\widehat{\delta} : Q \times \Sigma^* \to Q$ defined inductively via

- $\widehat{\delta}(q,\epsilon) := q$
- $\widehat{\delta}(q,\sigma_1) := \delta(q,\sigma_1)$
- $\widehat{\delta}(q,\sigma_1 \dots \sigma_k) = \delta(\widehat{\delta}(q,\sigma_1 \dots \sigma_{k-1}),\sigma_k)$

So for any $w \in \Sigma^\star$, $\widehat{\delta}(q_0,w)$ is just the end state on input w.

Lemma 2.1. *For all $q \in Q, \sigma_1,\dots,\sigma_k \in \Sigma^*$ and all $1 \leq l \leq k$, we have $\widehat{\delta}(q,\sigma_1 \dots \sigma_k) = \widehat{\delta}(\widehat{\delta}(q,\sigma_1 \dots \sigma_l),\sigma_{l+1} \dots \sigma_k)$*

Proof. This is clear by the inductive definition of $\widehat{\delta}$ □

Let $D = (Q,\Sigma,\delta,q_0,F)$ be a DFA. The ***accepted language*** $\mathcal{L}(D)$ is the set of all words $w \in \Sigma^*$ such that $\widehat{\delta}(q_0,w) \in F$, i.e. all accepted words. A language is ***regular*** if it is the accepted language for some DFA.

Non Determinism and ϵ-transitions

We now examine a seemingly more general construction where we can choose which state to move to or move to multiple states simultaneously.

A nondeterministic finite state automata (NFA) is a structure $N = (Q,\Sigma,\delta,q_0,F)$ just as for a DFA except that $\delta : Q \times \Sigma \to \mathcal{P}(Q)$.

An NFA has input $w = \sigma_1 \dots \sigma_k \in \Sigma^*$. We start in state q_0, read σ_1 then move to all states in $\delta(q_0,\sigma) \subseteq Q$. Then for all the states $p \in \delta(q_0,\sigma_1)$ we read σ_2 and move to $\delta(p,\sigma_2)$ so we are now at $\cup_{p\in\delta(q_0,\sigma_1)}\delta(p,\sigma_2)$. Then continue this process to the end of w and get the end states $E_w \subseteq Q$. If $E_w \cap F \neq 0$ then w is accepted, else w is rejected.

For an ϵ-NFA we can also define its extended transition function $\widehat{\delta}(q,w)$ to be the set of all states we can get to by reading w starting at state q, reading as many ϵs as we want to.

Now observe that any DFA is a NFA (just only ever move to one state), and any NFA is an ϵ-NFA (let $\delta(q,\epsilon)=\emptyset$).

Given an (ϵ-)NFA, (E or) N, its ***language*** $\mathcal{L}\binom{E}{N}$ is $\{w \in \Sigma^* : \widehat{\delta}(q_0, w) \cap F \neq \emptyset\} \subseteq \Sigma^*$. Note that this definition is consistent whether or not we have a DFA, NFA, or ϵ-NFA.

Equivalence of DFAs and (ϵ-)NFAs

Any regular language is the language of an (ϵ-)NFA. Here we show the converse.

First, suppose that we have an NFA N. We produce a DFA D on the same alphabet Σ such that $\mathcal{L}(N) = \mathcal{L}(D)$.

For the NFA $N = (Q, \Sigma, \delta, q_0, F)$, the ***subset construction*** for N produces the DFA $D = (\mathcal{P}(Q), \Sigma, \Delta, \{q_0\}, \mathfrak{F})$, where $\mathfrak{F} \coloneqq \{S \subseteq Q : S \cap F \neq \emptyset\}$, the subsets of states that contain an accept state, and for $S \subseteq Q, \sigma \in \Sigma$ we have $\Delta(S,\sigma) \coloneqq \cup_{p\in S}\delta(p,\sigma)$. Essentially, we replace what was our states, which we could be in more that one of simultaneously, with sets of states, which we can only be in one of at a time.

Theorem 2.2. *For the NFA N and DFA D obtained by the subset construction, let $\widehat{\delta}, \widehat{\Delta}$ be their respective extended transition function. Then for all $w \in \Sigma^*$, we have $\widehat{\Delta}(\{q_0\}, w) = \widehat{\delta}(q_0, w)$.*

Proof. Induction on length $k = |w|$. The base case ($k = 0$) is $w = \epsilon$, and LHS $= \{q_0\} =$ RHS.

Now suppose that $\widehat{\Delta}(\{q_0\}, \sigma_1, \ldots, \sigma_{k-1}) = \widehat{\delta}(q_0, \sigma_1, \ldots, \sigma_{k-1})$. Then for $w = \sigma_1 \ldots \sigma_{k-1}\sigma_k$ we have $\widehat{\delta}(q_0, w) = \cup_{p\in\delta(q_0,\sigma_1,\ldots,\sigma_{k-1}}\delta(p,\sigma_k)$, which is by definition the same as $\Delta(\widehat{\delta}(q_0, \sigma_1 \ldots \sigma_{k-1}), \sigma_k) = \Delta(\widehat{\Delta}(\{q_0\}, \sigma_1 \ldots \sigma_{k-1}), \sigma_k) = \widehat{\delta}(\{q_0\}, \sigma_1 \ldots \sigma_k)$ □

Corollary 2.3. *A language $\mathcal{L}$ is accepted by some DFA if and only if $\mathcal{L}$ is accepted by some NFA.*

Proof. For the forwards direction, just convert the DFA into an NFA by replacing the states by singletons of the states.

For the reverse direction, use the subset construction. Then $w \in \Sigma^*$ is accepted iff $\widehat{\delta}(q_0, w \cap F \neq \emptyset \iff \widehat{\Delta}(\{q_0\}, w) \in \mathfrak{F}$. □

Likewise, for an ϵ-NFA, the ***subset construction with ϵ-transitions*** produces a DFA $D = (\mathcal{P}(Q), \Sigma, \Delta, \mathrm{ecl}(q_0), \mathfrak{F}), \Delta(S,\sigma) \coloneqq \mathrm{ecl}(\cup_{p\in S}\delta(p,\sigma))$

Theorem 2.4. *For an ϵ-NFA E and a DFA D obtained by the subset construction with ϵ-transitions, we have $\widehat{\Delta}(\mathrm{ecl}(q_0), w) = \widehat{\delta}(q_0, w)$.*

Proof. Identical to the previous proof for NFA → DFA.

Corollary 2.5. *Any DFA, NFA, ϵ-NFA can be represented by any of the others. The constructions are equivalent.*

□

Theorem 2.6. *For each DFA D there is a regular expression R such that $\mathcal{L}(D) = \mathcal{L}(R)$.*

Proof. Given $D = (Q, \Sigma, \delta, q_0, F)$, label the states wlog by $\{1, 2, \ldots, n\}$ with 1 start state q_0. Now, given $1 \leq i, j \leq n$ and $0 \leq k \leq n$ we construct by induction on k for each i, j a regular expression $R_{ij}^{(k)}$ with language exactly the words w that begin at state i, end at state j, and such that all intermediate states passed through not including the endpoints are at most k. (Think of w as a directed path from i to j.)

For base case we use $k = 0$ we have to go directly from i to j, so let $\{a_1, \ldots, a_l\}$ be all symbols labelling an edge from i to g. If $i \neq j$ then set $R_{ij}^{(0)} = \begin{cases} a_1 + \ldots + a_l & l > 0 \\ \emptyset & \text{otherwise} \end{cases}$

Otherwise if $i = j$ let $R_{ii}^{(0)} = \begin{cases} a_1 + \ldots + a_l + \epsilon & l > 0 \\ \epsilon & \text{otherwise} \end{cases}$

Suppose now that we have $R_{ij}^{(t)}$ defined for all $t < k$. Given a path π from i to j not passing through states $k+1, \ldots, n$ either we have:

1. π does not pass through k, and so a word read from π is in $\mathcal{L}(R_{ij}^{(k-1)})$, and vice versa.
2. *pi* does pass through k at least once, so split path as σ from i to k, then $\pi_1, \ldots, \pi_r$ from k to k, then ω from k to j, so that $\pi = \sigma\pi_1\pi_2\ldots\pi_r\omega$.

   The word read from the subpath σ is in $\mathcal{L}(R_{ik}^{(k-1)})$. Each word from $\pi_1, \ldots, \pi_r$ is in $\mathcal{L}(R_{kk}^{(k-1)})$ and each word from ω is in $\mathcal{L}(R_{kj}^{(k-1)})$. So word read from *pi* is in the language defined by $R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$.

Thus for both cases we have a word read from π is in the language defined by the sum of the two regular expressions.

Finally, if $F = \{j_1, \ldots, j_s\}$ then $\mathcal{L}(D) = \mathcal{L}(R_{ij_1}^{(n)} + \ldots + R_{1j_s}^{(n)}))$ □

Closure Properties

Theorem 2.7. *Let L, M be regular languages over Σ_L, Σ_M. Then*

1. *$L \cup M$ is regular over $\Sigma_L \cup \Sigma_M$.*
2. *LM is regular over $\Sigma_L \cup \Sigma_M$.*
3. *L^* is regular over Σ_L.*

Proof. Take regular expressions R_L, R_M for L, M, so $\mathcal{L}(R_L) = L$, and $\mathcal{L}(R_M) = M$. Then for *1* we have $R_L + R_M$ is regular and $\mathcal{R_L} + \mathcal{R_M} = \mathcal{L}(R_L) \cup \mathcal{L}(R_M) = L \cup M$. For *2* $R_L R_M$ is regular with language $\mathcal{L}(R_L)\mathcal{L}(R_M) = LM$. For *3* $(R_L)^*$ is regular and $\mathcal{L}((R_L)^*) = \mathcal{L}(R_L)^* = L^*$. □

Theorem 2.8. *If L is a regular language over Σ then so is $\bar{L} = \Sigma^* \setminus L$, the* **complement** *of L.*

Proof. Given a DFA D with $\mathcal{L}(D) = L$, consider the DFA $\bar{D}$ which is the same as D but with accept states $Q \setminus F$. Then for any $w \in \Sigma^*$, we have that $w \in \bar{L} \iff w \notin L \iff \widehat{\delta}(q_0, w) \notin F \iff \widehat{\delta}(q_0, w) \in Q \setminus F \iff w \in \mathcal{L}(\bar{D})$. □

Theorem 2.9. *If L, M are regular languages, then so is $L \cap M$.*

Proof. $\bar{L}, \bar{M}$ are regular, so $\overline{\bar{L} \cup \bar{M}} = L \cap M$ is. □

The Pumping Lemma for Regular Languages

How do we show a language is not regular? Suppose a word w is accepted by a DFA D but $|w| > |Q|$ (w has more symbols than D has states). Then w defines a directed path in the transition diagram Γ_D which must visit some state at least twice. So we can write $w = xyz$ for subdivisions x, y, z where y is a loop. But then we can go around the loop twice or more...

Theorem 2.10 (Pumping Lemma for Regular Languages)**.** *If L is a regular language then there is some $n \geq 1$ such that for all words $w \in L$ with $|w| \geq n$ we can write $w = xyz$ for subwords x, y, z where*

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. *For all* $k \geq 0$ *we have* $xy^kz \in L$

Proof. We have $L = \mathcal{L}(D)$ for a DFA $D = (Q, \Sigma, \delta, q_0, F)$, and set $n = |Q|$. Now, take an accepted word $w \in L$ with $w = \sigma_1 \dots \sigma_m$ where $m \geq n$. Let $p_i = \widehat{\delta}(q_0, \sigma_1 \dots \sigma_i)$ be the state we're in after reading the first i letters of w, for $0 \leq i \leq m$, so that $p_0 = q_0$.

Then by the pigeonhole principle there is some $r < s$ such that $p_r = p_s$.

But then let $w^{(n)} := \sigma_1 \dots \sigma_{r-1}(\sigma_r \dots \sigma_s)^n \sigma_{s+1} \dots \sigma_n$. Then $w^{(n)} \in L$ □

Example:
The language $L = \{0^i1^i : i \geq 0\}$ is *not* regular. If it were, then we have arbitrarily long words, so one is longer than the double number of states in the DFA, say $w = 0^n1^n$ where the DFA has n states. Then by the pumping lemma $w = xyz$ where $|xy| \leq n$ (so that y is non-empty), with xz is accepted. Note that z is all the ones, and possibly some of the zeroes. So xz has more 1s than 0s, so is not in L ↯.

Equivalence and Minimisation of DFAs

Given a DFA D over an alphabet Σ, there are many other DFAs for the same language. For instance, we can just noodle around doing not much and add weird bits that are inaccessible. We want one with the minimal possible number of states.

An ***accessible state*** $q \in Q$ of a DFA D is one where there is some word that leads the DFA to that state.

From D we can form a DFA A with no inaccessible states and $\mathcal{L}(A) = \mathcal{D}$: if $n = |Q_D|$, then form $S_i \subseteq Q_D$ via $S_0 = \{q_0\}, S_{i+1} = \cup_{q \in S_i}(\cup_{\sigma \in \Sigma} \delta_D(q, \sigma))$, so S_i is the states reachable from q_0 after i states. But any accessible state can be reached after $\leq n - 1$ steps, removing closed loops of repeated states. So the stet $S = \cup_{j=1}^{n-1} S_j$, so set $S = \cup_{j=0}^{n-1} S_j$, and $Q_A = S \subseteq Q_D$, $F_A = F_D \cap S$, and δ_A be the restriction of δ_D to $Q_A \times \Sigma$. However, this might not be enough. Moreover, what about uniqueness?

For A, B both DFAs over Σ, we say that A, B are ***equivalent***, written $A \equiv B$ if, up to relabelling of non-start states, they are the same DFA. Now for a single DFA D, we group together equivalent states.

For a DFA D, we call two states $p, q \in Q_D$ ***equivalent*** or ***indistinguishable*** if for all $w \in \Sigma^*$ we have that $\widehat{\delta}_D(p, w) \in F \iff \widehat{\delta}_D(q, w) \in F$, written $p \sim q$, otherwise we say that p, q are ***distinguished*** by w.

Proposition 2.11. *For a DFA D and states $p, q \in Q$, then:*

1. *$\sim$ is an equivalence relation on Q*
2. *If $p \sim q$ then $\forall \sigma \in \Sigma$ we have $\delta(p, \sigma) \sim \delta(q, \sigma)$ and $\forall w \in \Sigma^*$ we have $\widehat{\delta}(p, w) \sim \widehat{\delta}(p, w)$.*
3. *If $p \sim q$ then p is an accepting state if and only q*

Proof.

1. For transitivity, we say $p \sim q$ and $q \sim r$ and take a word $w \in \Sigma^*$. Then both $\widehat{\delta}(p, w)$, $\widehat{\delta}(q, w)$ are (non)accepting, as are both $\widehat{\delta}(q, w), \widehat{\delta}(r, w)$. So it is true for $\widehat{\delta}(p, w)$ and $\widehat{\delta}(r, w)$.
2. Suppose we have that $\sigma \in \Sigma$ with $\delta(p, \sigma) \not\sim \delta(q, \sigma)$, and so without loss of generality we have some $w \in \Sigma^*$ with $\widehat{\delta}(\delta(p, \sigma), w)$ accepted, but $\widehat{\delta}(\delta(q, \sigma), w)$ is not. Then p, q are distinguished by word σw. Then induction on $|w|$.
3. Use $w = \epsilon$ in the definition.

□

Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, we define the ***minimal DFA*** $D/\sim$ for D as $(Q/\sim, \Sigma, \delta', [q_0], F')$ where $\delta'([q], \sigma) = [\delta(q, \sigma)]$, well defined by the previous proposition, and the accept states are given by $F' = \{[p] : p \in F\}$ with $[p] \subseteq F$.

Lemma 2.12. *For a DFA D and $D/\sim$ as above, take $q \in Q$ and $w \in \Sigma^*$. Then $\widehat{\delta}'([q], w) = [\widehat{\delta}(q, w)]$.*

Proof. Induction on k for $w = \sigma_1 \dots \sigma_k$. Check the base case. So we have $LHS = \delta'(\widehat{\delta}'([q], \sigma_1 \dots \sigma_{k-1}), \sigma_k)$. Then the induction step follows immediately from the definitions. □

Theorem 2.13. *For a DFA D and its minimal DFA $D/\sim$ we have that $\mathcal{L}(D/\sim) = \mathcal{L}(D)$.*

Proof. For all $w \in \Sigma^*$, we have $w \in \mathcal{L}(D/\sim) \iff \widehat{\delta}/([q_0], w) \in F' \iff [\widehat{\delta}(q, w)] \in F' \iff \widehat{\delta}(q, w) \in F \iff w \in \mathcal{L}(D)$. □

Lemma 2.14. *For a DFA D and DFA $D/\sim$ no two distinct states of $D/\sim$ are equivalent.*

Proof. Suppose $p, q \in Q$ have $[p] \sim [q]$ in $D/\sim$. Then for all words w we have $\widehat{\delta}'([p], w), \widehat{\delta}'([q], w)$ are both (non)accepting in $D/\sim$. So $[\widehat{\delta}(p, w)], [\widehat{\delta}(q, w)]$ are both (non)accepting in $D/\sim$ by the **2.10**, and so $\widehat{\delta}(p, w), \widehat{\delta}(q, w)$ are both (non)accepting. So $p \sim q$ in D, so $[p] = [q]$ in $D/\sim$. □

Corollary 2.15. *If D is a DFA then $(D/\sim)/\sim \equiv D/\sim$.*

Proof. By the previous lemma, we don't remove any states, and so we can relabel $[q]$ in the former to just q and this is well-defined, giving the identical transition function. □

Note that this reduction of a DFA does not get rid of inaccessible states, but we will see that $D/\sim$ with all inaccessible states removed is indeed minimal for its language.

Theorem 2.16. *If the DFA D has no inaccessible states and A is a DFA with $\mathcal{L}(A) = \mathcal{L}(D)$, then A has at least as many states as $D/\sim$ and if they have the same number of states they are equivalent.*

Proof. Take a DFA B with the fewest number of states subject to $\mathcal{L}(B) = \mathcal{L}(D)$. Then B has no inaccessible states and $D/\sim$ has none, so for any state $[q]$ of $D/\sim$ we have some word w with $\widehat{\delta}(q_0, w) = q$, since D has no inaccessible states, and so $\widehat{\delta'}([q_0], w) = [q]$.

So we have $D/\sim = (Q/\sim, \Sigma, \delta', [q_0], F')$ and $B = (S, \Sigma, d, s_0, G)$.

Then form the disjoint union DFA of $D/\sim, B$ given by:

$U = ((Q/\sim) \coprod S, \Sigma, \Delta, [q_0], F' \coprod G)$. As $\mathcal{L}(D/\sim) = \mathcal{L}(D) = \mathcal{L}(B)$, this means that states $[q_0]$ and s_0 are equivalent, since $\widehat{\Delta}([q_0], w)$ is the same as $\widehat{\Delta}(s_0, w)$ - we accept/reject the same words. Hence $s_0 \sim [q_0]$.

Then if $p = \widehat{\Delta}([q_0], w) \sim \widehat{\delta}(s_0, w) = \widehat{d}(s_0, w) = s \in S$, say. The first equality is valid for all p as we have no inaccessible states, so every $p \in Q/\sim$ has some state $s \in S$ with $p \sim s$.

Likewise, for every $s \in S$ there is some $p \in Q/\sim$ with $s \sim p$.

Now, since $p \not\sim q$ for $p \neq q$ in $Q/\sim$, and since $\sim$ is an equivalence relation, we must have $|B| \geq |D/\sim|$, with equality precisely when we have a one-to-one equivalence pairing between $Q/\sim$ and S, i.e. the DFAs are themselves equivalent. □

What can we do with algorithms?

We can remove inaccessible states: to find equivalent states, we can use ***Hopcroft's table filling algorithm***: given a DFA D, we have a table $Q \times Q$ with $(p, q)^{\text{th}}$ entry marked with an × when we have shown that $p \not\sim q$. We only fill in the lower left part, starting will all blanks.

First off, put in × whenever we have (p, q) with one in F and the other not. Then take any (p, q) which is empty. If we have $\sigma \in \Sigma$ with $\delta(p, \sigma) = r, \delta(q, \sigma)$ and (r, s) marked × then mark with a ×. Halt when this fails for all unmarked pairs. This works by obviously working and I don't really feel like copying down half an hour of proof.

We can also test if two DFAs are the same - minimise both and check if they have the same number of states, and check if they are equivalent, and if yes they have the same language.