Automata and Formal Languages

Cambridge University Mathematical Tripos: Part II

17th May 2024

Contents

1 Introduction

1.1 Exposition

Computation, or computability, is central to modern mathematics. However, we very rarely think about the precise definition of what it means for something to be ‘computable’. There is an important difference between existence and algorithmic access to a witness. Contrast the statements ‘every polynomial of order n has a root’, and ‘there is an algorithm that, given a polynomial of order n, we can find a root’. In many cases, there is an existence proof but no algorithm to construct the relevant object.

In 1900, Hilbert gave a talk in Paris known as *Mathematical Problems*, in which he described a list of 100 problems to be worked on in the coming 100 years. One of these problems, the tenth, relates to an algorithm to determine whether solutions of Diophantine equations, those in $\mathbb{Z}[X]$, exist. In 1928, Ackermann wrote the book *Grundzüge der theoretischen Logik*, in which he described the famous *Entscheidungsproblem*: given a formula φ, determine whether φ is a tautology (true regardless of how the variables are interpreted).

In both cases, Hilbert expected that solutions to these questions exist. Positive solutions to such problems do not require a definition of words like ‘algorithm’ or ‘procedure’, because we can agree on what an algorithm is when we see an example. However, to disprove such statements, we need to rigorously define what an algorithm is, in order to rule all possible algorithms out.

1.2 Basic definitions

To talk about computation, we must first define the objects on which computation takes place. Naturally, one would assume the objects to be some kind of number, but even the above two examples do not have inputs as numbers; instead, we see polynomials and formulas. Modern computation relies on encodings of complicated objects as strings of a finite set of symbols, such as the bits 0 and 1. We use a similar approach, using a set Ω, which is usually assumed to be finite, called the set of *symbols*, and then we define $\Omega^\star$ to be the set of finite sequences of objects of Ω, called the set of Ω-*strings*.

1.3 Revisiting Numbers and Sets

Recall that a set X is called *countable* if there is a surjection $\mathbb{N} \to X$, and that X is called *infinite* if there is an injection $\mathbb{N} \to X$.

Proposition. If X is nonempty and countable, then $X^\star$ is infinite and countable.

Proof. Since $X \neq \varnothing$, there exists $x \in X$. $X^\star$ is infinite, as the function mapping $n \in \mathbb{N}$ to $\underbrace{xx \dots x}_{n \text{ times}}$ is injective. Because X is countable, there exists a surjection $\pi : \mathbb{N} \to X$. Each natural $k \in \mathbb{N}$ has a unique prime number decomposition $\prod_{i \in \mathbb{N}} p_i^{k_i}$ where $p_0 = 2, p_1 = 3, p_2 = 5, \dots$ are the primes indexed by the naturals. We will interpret the k_i as encoding a sequence of elements of X, taking care to preserve the relevance of zero. Reading k_0 as the length of a sequence, the sequence $(k_1, \dots, k_{k_0})$ is a sequence of naturals. We then obtain the sequence $(\pi(k_1), \dots, \pi(k_{k_0}))$ in $X^\star$. By surjectivity of π, the function we have constructed $k \mapsto (\pi(k_1), \dots, \pi(k_{k_0}))$ is also surjective. □

Theorem (Cantor's theorem)**.** Let X be infinite. Then its power set $\mathcal{P}(X)$ is uncountable.

Proof. A simple diagonalisation argument shows there is no surjection from the naturals to the power set $\mathcal{P}(X)$. □

Proposition. If X is countable, then the set $\mathrm{Fin}(X) \subseteq \mathcal{P}(X)$ of all finite subsets of X is countable.

Proof. We construct a surjection from $X^\star$ to $\mathrm{Fin}(X)$; then by composition with the surjection obtained in the first proposition we construct a surjection $\mathbb{N} \to \mathrm{Fin}(X)$. Consider the forgetful function $f : X^\star \to \mathrm{Fin}(X)$, mapping $(x_1, \dots, x_n)$ to $\{x_1, \dots, x_n\}$. Since X is countable, $\pi : \mathbb{N} \to X$ is surjective, hence for $x \in X$, $\pi^{-1}(x) \subseteq \mathbb{N}$ is a nonempty set of naturals. Therefore, let n_x be the least element of $\pi^{-1}(x)$. Then, given $F \in \mathrm{Fin}(X)$, consider the set $\{n_x \mid x \in F\}$, order it in the usual way, and represent this as a sequence. This is a sequence of naturals with $|F|$ elements, and its π-image is exactly F. □

1.4 Notation

We will use the following notational conventions.

- The natural numbers $\mathbb{N}$ are defined as $\{0, 1, 2, \dots\}$.
- We use the standard set-theoretic construction of naturals as Von Neumann ordinals, $n = \{0, 1, \dots, n-1\}$. Therefore, a natural is the set of all lower naturals.
- X^n is the set of sequence of X-strings of length n, defined as $X^n = n \to X$, treating n as a set as above.
- We write $|\alpha| = \mathrm{domain}(\alpha)$ for the length of a sequence.
- $X^0 = 0 \to X$ is a type with only one element ε, which is the empty sequence.
- We can write $X^\star = \bigcup_{n\in\mathbb{N}} X^n$.
- Truncation of a sequence $\alpha \in X^n$ to the length $k \leq n$ is exactly $\alpha|_k$: the unique sequence of length k such that $\alpha|_k \subseteq \alpha$.
- Concatenation of sequences $\alpha, \beta \in X^\star$ where $|\alpha| = m, |\beta| = n$, is denoted $\alpha\beta \in X^{m+n}$, defined piecewise in the natural way.
- By recursion, we define $\alpha^0 = \varepsilon$ and $\alpha^{n+1} = \alpha\alpha^n$.
- We identify the sequence of length one with its entry: $x \in X$ can represent the sequence $(x) \in X^1$.
- If $Y, Z \subseteq X^\star$, we write $YZ = \{\alpha\beta \mid \alpha \in Y, \beta \in Z\}$.
- Similarly, if $Y = \{\alpha\}$, we can write $\alpha Z = \{\alpha\beta \mid \beta \in Z\}$.
- If $f : X \to Y$, we can lift this function to the space $X^\star \to Y^\star$ functorially to the function $\hat{f}$. Often, the hat is omitted.

2 Rewrite systems

2.1 Definitions

Definition. Let Ω be a finite set of symbols, and let $\Omega^\star$ be the set of Ω-strings. We call elements of $\Omega^\star \times \Omega^\star$ *rewrite rules* or *production rules*. Such elements (α, β) are written $\alpha \to \beta$.

Informally, we interpret a rewrite rule $\alpha \to \beta$ as a procedure that replaces an occurrence of α in a string with β.

Definition. A pair $R = (\Omega, P)$ is called a *rewrite system* if P is a finite set of rewrite rules.

Proposition. If Ω is finite, there are only countably many rewrite systems on Ω.

Proof. $\Omega^\star$ is countable, so $\Omega^\star \times \Omega^\star$ is countable. Every P is an element of $\mathrm{Fin}(\Omega^\star \times \Omega^\star)$, hence this is countable. □

Definition. If $R = (\Omega, P)$ is a rewrite system, and $\sigma, \tau \in \Omega^\star$, we write $\sigma \overset{R}{\to}_1 \tau$, pronounced '$\sigma$ is rewritten to τ in one step' or 'R produces τ from σ in one step', if there exist $\alpha, \beta, \gamma, \delta \in \Omega^\star$ such that $\sigma = \alpha\gamma\beta$, $\tau = \alpha\delta\beta$, and $\gamma \to \delta \in P$.

The relation $\overset{R}{\to}$ is the reflexive and transitive closure of $\overset{R}{\to}_1$. The sequence $\sigma_0 \overset{R}{\to}_1 \sigma_1 \overset{R}{\to}_1 \dots \overset{R}{\to}_1 \sigma_n$ is called a *R-derivation of length n* of σ_n from σ_0. We write

$$\mathcal{D}(R, \sigma) = \left\{ \tau \in \Omega^\star \mid \sigma \overset{R}{\to} \tau \right\}$$

for the set of strings that can be rewritten, produced, or derived from σ.

2.2 Relation to languages

In language, we can think of Ω as representing letters, and $\Omega^\star$ representing words. We could alternatively consider Ω to represent words, and $\Omega^\star$ to represent sentences. Further, Ω could represent sentences, and then $\Omega^\star$ would represent texts.

However, not all elements of $\Omega^\star$ in each level is a valid word, sentence, or text. We therefore would like to describe which elements of $\Omega^\star$ are *well-formed*. Natural languages spoken by humans are finite, and normally the way we determine whether a string is a word is by consulting a dictionary, which at its core is a lookup table that determines whether any given string is or is not a word.

Even though in practice languages are finite, Chomsky realised that it makes more sense to model them as infinite sets, due to a property known as *linguistic recursion* that seems to be an important feature of human language. Linguistic recursion can be seen through the following example: when X is a sentence in English, 'E observes that X' is also a grammatical sentence in English. If we define an upper sentence length in English, we have to arbitrarily define an upper limit on this form of recursion.

There is a difference between a sentence being grammatical and being meaningful. One notable example is the grammatically correct ‘colourless green ideas sleep furiously’ that does not have meaning, to contrast with ‘furiously sleep ideas green colourless’ which is neither grammatically correct or meaningful. We can use grammar to distinguish these two sentences, but we cannot distinguish algebraically whether a sentence has meaning.

Example. Consider the following *generative grammar* of rewrite rules for English.

$$\begin{aligned} S &\to \text{NP VP} \\ \text{NP} &\to \text{Adj NP} \\ \text{NP} &\to \text{Noun} \\ \text{VP} &\to \text{Verb} \\ \text{VP} &\to \text{Verb Adv} \end{aligned}$$

This rewrite system allows us to derive the sentence ‘colourless green ideas sleep furiously’ from S.

2.3 Grammars

Definition. Let Σ be an *alphabet* of *letters* or *terminal symbols*, and let V be a set of *variables* or *nonterminal symbols*, such that Σ, V are nonempty and disjoint. Let $\Omega = \Sigma \cup V$. $a, b, c, \dots$ refer to letters and $A, B, C, \dots$ refer to variables. Elements of $\mathbb{W} = \Sigma^\star \subseteq \Omega^\star$ are called *words*. $u, v, w, \dots$ refer to words. We denote $\mathbb{W}^+ = \Sigma^\star \setminus \{\varepsilon\}$ for the set of nonempty words. A subset of $\mathbb{W}$ is called a *language*.

Note that there are uncountably many languages over any nonempty alphabet.

Definition. A tuple $G = (\Sigma, V, P, S)$ is called a *grammar* if Σ, V are nonempty and disjoint denoting $\Omega = \Sigma \cup V$, such that $R = (\Omega, P)$ is a rewrite system, and $S \in V$ is the *start symbol*. Since grammars give rise to a natural rewrite system, our notation for rewrite systems may also be used for grammars. For example,

$$\mathcal{D}(G, \sigma) = \mathcal{D}(R, \sigma); \quad \sigma \overset{G}{\to}_{(1)} \tau \iff \sigma \overset{R}{\to}_{(1)} \tau$$

We define the *language generated by the grammar* to be

$$\mathcal{L}(G) = \mathcal{D}(G, S) \cap \mathbb{W}$$

Example. If there is no rule of the form $S \to \alpha$ in P, then $\mathcal{D}(G, S) = \{S\}$ and thus $\mathcal{L}(G) = \varnothing$ because the start symbol is not a word. Likewise, if there is no rule of the form $\alpha \to w$ for $w \in \mathbb{W}$ in P, then $\mathcal{D}(G, S)$ contains no words, so $\mathcal{L}(G) = \varnothing$.

Example. Let $\Sigma = \{a\}$, $V = \{S\}$, $P_0 = \{S \to aaS, S \to a\}$, $G_0 = (\Sigma, V, P, S)$. We will show $\mathcal{L}(G_0) = \left\{a^{2n+1} \mid n \in \mathbb{N}\right\}$. First, every element of $\mathcal{D}(G, S)$ that is produced by G_0 is of odd length, which can be seen by induction on the length of the derivation, since each production rule preserves parity of length. Conversely, each a^{2n+1} can be produced by the rewrite rules, by applying $S \to aaS$ a total of n times, and then applying $S \to a$.

Note that the only requirement of the proof was that odd length is preserved. Thus, the following sets of production rules also produce the same language.

- $P_1 = \{S \to aSa, S \to a\}$
- $P_2 = \{S \to Saa, S \to a\}$
- $P_3 = \{S \to aaS, S \to aaSaa, S \to a\}$

This notion is called *equivalence of grammars*.

2.4 Equivalent grammars

Definition. Grammars G, G' are *equivalent* if $\mathcal{L}(G) = \mathcal{L}(G')$.

We intend to show that for a fixed finite set Σ, there are only countably many languages of the form $\mathcal{L}(G)$ for a grammar G (which may have arbitrary variable sets V).

Definition. Let $G = (\Sigma, V, P, S), G' = (\Sigma, V', P', S')$ be grammars on the same alphabet Σ. A function $f : \Omega \to \Omega' = \Sigma \cup V \to \Sigma \cup V'$ is called an *isomorphism* if
(i) $f|_\Sigma = \text{id}$;
(ii) $f(S) = S'$;
(iii) $f|_V$ is a bijection from V to V';
(iv) $\alpha \to \beta \in P \iff f(\alpha) \to f(\beta) \in P'$.
Note that here, since $\alpha, \beta \in \Omega^\star$, $f(\alpha) = \hat{f}(\alpha)$ is the extension of f to $\Omega^\star$.

Proposition. Isomorphic grammars are equivalent.

Proof. If f is an isomorphism from G to G', f^{-1} is an isomorphism from G' to G. Thus, by antisymmetry of $\subseteq$, it suffices to show that $\mathcal{L}(G) \subseteq \mathcal{L}(G')$. Let $w \in \mathcal{L}(G)$. Then there is a derivation in G of w from S:

$$S = \sigma_0 \overset{G}{\longrightarrow}_1 \sigma_1 \overset{G}{\longrightarrow}_1 \dots \overset{G}{\longrightarrow}_1 \sigma_n = w$$

Applying f to each element of this sequence,

$$S' = f(\sigma_0) \overset{G'}{\longrightarrow}_1 f(\sigma_1) \overset{G'}{\longrightarrow}_1 \dots \overset{G'}{\longrightarrow}_1 f(\sigma_n) = w$$

The start and end symbols take these values due to property (i) and (ii). Each arrow holds by property (iv). This is a derivation of w from S' in G. Hence $w \in \mathcal{L}(G')$. $\square$

Proposition. If $G = (\Sigma, V, P, S)$ and V' is such that $|V| = |V'|$, then there exist P', S' such that $\mathcal{L}(G) = \mathcal{L}(G')$ with $G' = (\Sigma, V', P', S')$.

Proof. Since $|V| = |V'|$, there exists a bijection $f : V \to V'$. Then, extending this to $\Omega = \Sigma \cup V$ by letting $f(a) = a$ for all $a \in \Sigma$, this satisfies properties (i) and (iii) of the definition of an isomorphism. Define $S' = f(S)$ and $P' = \{f(\alpha) \to f(\beta) \mid \alpha \to \beta \in P\}$, so that properties (ii) and (iv) are satisfied. Then (Σ, V, P, S) is isomorphic to (Σ, V', P', S') and thus they have the same language. $\square$

Proposition. There are only countably many languages of the form $\mathcal{L}(G)$ for some grammar G on a fixed alphabet Σ.

Proof. Let $\mathcal{L}$ be the set of all such languages. For a fixed V, there are only countably many rewrite systems with this choice of Σ and V. Hence, the set $\mathcal{G}_V$ of all grammars with fixed V is a finite union (over all start symbols) of countable sets. Therefore $\mathcal{L}_V = \{\mathcal{L}(G) \mid G \in \mathcal{G}_V\}$ is also countable.

By the previous result, we can define $\mathcal{L}_n = \mathcal{L}_V$ for some n-element set V. Now, $\mathcal{L} = \bigcup_{n>0} \mathcal{L}_n$, which is a countable union of countable sets and is thus countable. □

Remark. The set of languages produced by grammars is countable, but the set of all languages $\mathcal{P}(\mathbb{W})$ is uncountable.

2.5 The Chomsky hierarchy

Production rules may have certain properties.

Definition. Let $\alpha \to \beta$ be a production rule. We call this rule:
(i) *noncontracting*, if $|\alpha| \leq |\beta|$;
(ii) *context-sensitive*, if $\exists A \in V$, $\exists \gamma, \delta \in \Omega^\star$, $\exists \eta \in \Omega^+$, $\alpha = \gamma A \delta, \beta = \gamma \eta \delta$;
(iii) *context-free*, if $\alpha = A \in V$ and $|\beta| > 0$;
(iv) *regular*, if $\alpha = A \in V$ and β is either $a \in \Sigma$ or $aB \in \Sigma V$.
Regular implies context-free, context-free implies context-sensitive, context-sensitive implies noncontracting. Let $\mathbb{Q}$ be any of the above four properties. We say that a grammar is $\mathbb{Q}$ if all its production rules are $\mathbb{Q}$. A language is $\mathbb{Q}$ if it admits a grammar which is $\mathbb{Q}$.

Theorem (Chomsky)**.** A language is noncontracting if and only if it is context-sensitive.

Chomsky used the following notation: a language $\mathcal{L}$ is

- *type 0*, if it is of the form $\mathcal{L}(G)$ for some G;
- *type 1*, if it is of the form $\mathcal{L}(G)$ for some G context-sensitive;
- *type 2*, if it is of the form $\mathcal{L}(G)$ for some G context-free;
- *type 3*, if it is of the form $\mathcal{L}(G)$ for some G regular.

We can easily find production rules that are context-sensitive but not context-free, for example. However, it is less obvious to show that there is a *language* that can be defined using a context-sensitive grammar but no context-free grammar. One thing motivating our work will be the development of techniques to distinguish the different classes of languages in the Chomsky hierarchy.

2.6 Decision problems

We present three important decision problems.

(i) Consider the *word problem*. The input to this problem is a grammar and a word; the question is to determine whether the word lies in the language generated by the grammar.

(ii) The *emptiness problem* considers a grammar G. The question is whether $\mathcal{L}(G) = \varnothing$.

(iii) The *equivalence problem* asks whether two grammars G, G' are equivalent.

Definition. We call a problem *solvable* if there is an algorithm that gives the correct answer. Otherwise, we call such a problem *unsolvable*.

Posed for all grammars, all three problems above are unsolvable. However, when restricted to certain classes of the Chomsky hierarchy, the problems are more approachable.

Lemma. If G is a noncontracting grammar and $w \in \mathbb{W}$, there exists a bound $N \in \mathbb{N}$ depending only on $|w|$ and $|\Omega|$ such that $w \in \mathcal{L}(G)$ if and only if w has a G-derivation of length at most N.

Proof. Consider a G-derivation $S = \sigma_0, \dots, \sigma_n = w$ of w, and consider the length of each element of the sequence. As the grammar is noncontracting, the sequence $1 = |\sigma_0|, \dots, |\sigma_n| = |w|$ is non-decreasing. Consider a part of the derivation $\sigma_i, \dots, \sigma_{i+k}$ for which the length of the $|\sigma_i|$ does not change, so $|\sigma_i| = |\sigma_{i+k}|$. If $\sigma_r = \sigma_s$ for some $r \neq s \in \{i, \dots, i+k\}$, we can shrink the derivation to $\sigma_i, \dots, \sigma_r, \sigma_{s+1}, \dots, \sigma_{i+k}$.

Therefore, without loss of generality, we can assume $\sigma_0, \dots, \sigma_n$ is a derivation of minimal length, so all σ_i are distinct. Then by the pigeonhole principle,

$$n \leq \sum_{\ell=1}^{|w|} |\Omega|^\ell = N$$

□

Corollary. The word problem is solvable on noncontracting, context-sensitive, context-free, and regular grammars.

Proof. Let $w \in \mathbb{W}$. There is a finite, enumerable collection of possible derivations for w by the above lemma. Check each derivation manually. □

2.7 Closure problems

Closure problems are concerned with operations on languages to produce new languages. Let L, M be languages. Commonly used operations include $LM, L \cup M, L \cap M, L \setminus M, \mathbb{W}^+ \setminus L$. Note that we use $\mathbb{W}^+ \setminus L$ instead of L^c because noncontracting languages cannot contain the empty word. If $\mathcal{C}$ is a class of languages, such as the class of all regular languages, we say that $\mathcal{C}$ is *closed* under an operation if applying that operation to elements of $\mathcal{C}$ yields a result which also lies in $\mathcal{C}$. We would like to see which classes are closed under which operations. Note that some closure properties imply others; for instance, closure under complement and intersection implies closure under union by De Morgan's laws.

Definition. Let $G = (\Sigma, V, P, S), G' = (\Sigma, V', P', S')$ be grammars. Then $H = (\Sigma, V \cup V' \cup$

$\{T\}, P^\star, T)$ is called the *concatenation grammar*, where T is a new variable, and

$$P^\star = P \cup P' \cup \{T \to SS'\}$$

$H' = (\Sigma, V \cup V' \cup \{T\}, P^{\star\star}, T)$ is called the *union grammar*, where T is a new variable, and

$$P^{\star\star} = P \cup P' \cup \{T \to S, T \to S'\}$$

Remark. $\mathcal{L}(G)\mathcal{L}(G') \subseteq \mathcal{L}(H)$ by construction, and $\mathcal{L}(G) \cup \mathcal{L}(G') \subseteq \mathcal{L}(H')$, but it is not true *a priori* that the converse holds, because P and P' could share some variables. We can assume that V, V' are disjoint by relabelling, but that is insufficient for the converses to hold, because there may be interaction on the level of letters in Σ, which cannot be relabelled. The concatenation grammar on context-free grammars is context-free, and the union grammar on regular languages is regular.

Definition. A production rule $\alpha \to \beta$ is *variable-based* if all symbols occurring in α are variables. A grammar is called variable-based if all its rules are variable-based.

Remark. Regular and context-free languages are variable-based. Context-sensitive languages are not all variable-based.

Lemma. Every grammar is equivalent to a variable-based grammar.

Proof. Let $G = (\Sigma, V, P, S)$. For each letter $a \in \Sigma$, we allocate a new variable X_a. We define the map $X : \Omega \to \Omega$ by $X(a) = X_a$ for $a \in \Sigma$, and $X(A) = A$ for $A \in V$. Then X extends in the natural way to a map $X : \Omega^\star \to \Omega^\star$. We can map each production rule in G to a version that uses only variables and no letters by applying X to both sides. Hence, we define $P' = \{X(\alpha) \to X(\beta) \mid \alpha \to \beta \in P\}$. Then, defining $P'' = \{X_a \to a \mid a \in \Sigma\}$, let $G' = (\Sigma, V \cup \{X_a \mid a \in \Sigma\}, P' \cup P'', S)$. This grammar is variable-based and so it suffices to show that it defines the same language as G.

Any G-derivation of w is transformed into a G'-derivation of $X(w)$ by the operation $\alpha \to X(\alpha)$. Similarly, if we have a G'-derivation that contains no letters anywhere, all strings occurring are of the form $X(\alpha)$ for some $\alpha \in \Omega^\star$, and the operation of replacing all occurrences of X_a with a transforms that derivation into a G-derivation. Thus, $w \in \mathcal{L}(G)$ if and only if $X(w) \in \mathcal{D}(G', S)$.

If $X(w) \in \mathcal{D}(G', S)$ then, by applying rules of the form $X_a \to a$ as needed, we have $w \in \mathcal{L}(G')$.

Conversely, suppose $w \in \mathcal{L}(G')$ and let $S = \sigma_0, \dots, \sigma_m = w$ be a G'-derivation of w. Applying the operation X to this derivation, we obtain a sequence $S = \tau_0, \dots, \tau_m$. This sequence is not necessarily a G'-derivation. If $\sigma_i \xrightarrow{G'}_1 \sigma_{i+1}$ was an application of a rule of the form $X(\alpha) \to X(\beta)$, then the same rule gives $X(\sigma_i) \xrightarrow{G'}_1 X(\sigma_{i+1})$. In the other case, $\sigma_i \xrightarrow{G'}_1 \sigma_{i+1}$ was an application of a rule of the form $X_a \to a$, so applying X gives $X(\sigma_i) = X(\sigma_{i+1})$. Since for each letter a there is only one production rule that produces a, we know that $|w|$-many steps of the derivation must be of this form. Thus, removing these steps will make the remainder of the sequence $\tau_0, \dots, \tau_m$ a G'-derivation of length $m - |w|$ of $X(w)$. Then $w \in \mathcal{L}(G)$ as required. □

Remark. The classes of context-free, context-sensitive, and noncontracting grammars are stable under the action of turning a grammar into its equivalent variable-based grammar; all added rules are regular. Regularity is not necessarily preserved.

Theorem. Let $G = (\Sigma, V, P, S), G' = (\Sigma, V', P', S')$ be variable-based grammars with $V \cap V' = \varnothing$. Then $\mathcal{L}(H) = \mathcal{L}(G)\mathcal{L}(G')$, and $\mathcal{L}(H') = \mathcal{L}(G) \cup \mathcal{L}(G')$. The classes of regular, context-free, context-sensitive, and noncontracting languages are closed under union. The classes of context-free, context-sensitive, and noncontracting languages are closed under concatenation.

Proof. First, convert G, G' into variable-based grammars with disjoint variable sets. Then, the concatenation grammar and union grammar for G, G' must produce the required language by disjointness. □

2.8 The empty word

Remark. By induction, we can easily show that noncontracting grammars cannot produce the empty word, so we usually work with $\mathbb{W}^+$ in place of $\mathbb{W}$. In general, adding the rule $S \to \varepsilon$ to a grammar in order to allow the empty word may introduce side-effects due to reuse of S.

Definition. A rule $\alpha \to \beta$ is *S-safe* if it does not contain S in β. A grammar is *ε-adequate* if all rules are S-safe.

An ε-adequate grammar admits the addition of the rule $S \to \varepsilon$ converting the language $\mathcal{L}(G)$ into $\mathcal{L}(G) \cup \{\varepsilon\}$. We can easily convert a grammar into an equivalent ε-adequate grammar by mapping $G = (\Sigma, V, P, S)$ to $G' = (\Sigma, V \cup \{T\}, P \cup \{T \to S\}, T)$ where T is a new variable.

3 Regular languages

3.1 Regular derivations

Definition. A rule of the form $A \to a$ is called a *terminal rule*. A rule of the form $A \to aB$ is called a *nonterminal rule*.

Lemma. Let G be a regular grammar. If $S \overset{G}{\longrightarrow} \alpha$, then $\alpha \in \mathbb{W} \cup \mathbb{W}V$.

Proof. This is shown by induction on the length of the derivation. The length-zero derivation gives $\alpha = S = \varepsilon S \in \mathbb{W}V$. Suppose $S \overset{G}{\longrightarrow} \beta \overset{G}{\longrightarrow}_1 \alpha$ where $\beta \in \mathbb{W} \cup \mathbb{W}V$. If $\beta \in \mathbb{W}$, β contains no variables, but all rules rewrite a variable. This contradicts that $\beta \overset{G}{\longrightarrow}_1 \alpha$. So suppose $\beta = wA$ for $w \in \mathbb{W}, A \in V$. Then the rule must be of the form $A \to a$ or $A \to aB$. Hence $\beta = wa$ or $\beta = waB$. In either case, the required invariant holds. □

Lemma. If $S \overset{G}{\longrightarrow} w$ for $w \in \mathbb{W}$, then the derivation has length $|w|$ and consists of precisely $|w| - 1$ nonterminal rules and one final terminal rule.

Proof. Terminal rules preserve the length of a string, and decrement the amount of variables. Nonterminal rules increment the length of a string, and preserve the amount of variables. Given that S is a string of length one with one variable, we must apply $|w| - 1$ nonterminal rules to increment the length of the string $|w| - 1$ times. By the previous lemma, the number of variables in each derived string is always 0 or 1. If the number ever reaches zero, nothing can be rewritten. Given $w \in \mathbb{W}$, the number must reach zero, so a single terminal rule must be applied at the end. □

Note that the derivation is not uniquely determined.

Lemma. Regular languages are closed under concatenation.

Proof. Let $G = (\Sigma, V, P, S), G' = (\Sigma, V', P', S')$, where without loss of generality $V \cap V' = \varnothing$. Let $P^\star$ be the set of production rules given by P, but for each terminal rule $A \to a$ in P, replace it with a nonterminal rule $A \to aS'$. Then let $H = (\Sigma, V \cup V', P^\star \cup P', S)$. We claim $\mathcal{L}(H) = \mathcal{L}(G)\mathcal{L}(G')$.

Suppose $S \xrightarrow{G} v$ and $S' \xrightarrow{G'} w$. Then $S \xrightarrow{H} vS'$, and so $S \xrightarrow{H} vw$ as required.

Conversely, suppose $S \xrightarrow{u}$ for $u \in \mathbb{W}$. By the above lemma, the derivation is of the form

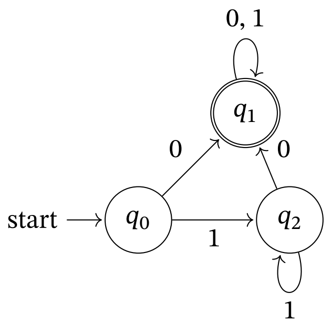
$$S = \sigma_0 \xrightarrow{H}_1 \dots \xrightarrow{H}_1 \sigma_n = u$$

where $\sigma_i = w_i X_i$ for some $w_i \in \mathbb{W}, X_i \in V$. An initial segment of the X_i belongs to V, until rewritten as S' by a rule added into $P^\star$. Then, the rest of the X_i belong to V', because only the new rules in $P^\star$ map variables between V and V'. Hence the derivation splits into two halves, $u = vw$ where $S \xrightarrow{G} v, S' \xrightarrow{G'} w$, giving the concatenation as required. □

3.2 Deterministic automata

Definition. Let Σ be an alphabet. Then a *deterministic automaton* is a tuple of the form $D = (\Sigma, Q, \delta, q_0, F)$ where Q is a finite set of *states*, $q_0 \in Q$ is the *start state*, $F \subseteq Q \setminus \{q_0\}$ is the *accept states*, and $\delta : Q \times \Sigma \to Q$ is the *transition function*.

We graphically represent deterministic automata using labelled directed graphs. The nodes are elements of Q, circled twice for accept states and circled once for other states. Each node has $|\Sigma|$-many outgoing arrows labelled with the corresponding letter.



We intuitively interpret a deterministic automaton as a machine that starts at q_0 and reads a word $w \in \mathbb{W}$ symbol-by-symbol, transitioning to a new state according to δ at each step. After all symbols in the word are parsed, we check whether the machine lies in an accept state or not. We say the automaton *accepts* w if the final state is an accept state; otherwise, it *rejects* w.

Definition. We define by recursion a function $\hat{\delta} \colon Q \times \mathbb{W} \to Q$ by $\hat{\delta}(q, \varepsilon) = q$ and $\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w)$. The *language accepted by D* is

$$\mathcal{L}(D) = \left\{w \mid \hat{\delta}(q_0, w) \in F\right\}$$

The sequence of states produced from q_0 and reading w is uniquely determined and of length $|w| + 1$, known as the *state sequence* of the computation.

We claim that in the example above, $\mathcal{L}(D) = \{w \mid w \text{ contains at least one } 0\}$. Note that $\hat{\delta}(q_0, w) = q_0$ if and only if $w = \varepsilon$. There are three transitions in the diagram for the letter 0, but all such 0-transitions lead to q_1 hence every string with a zero goes to q_1. All transitions from q_1 go back to q_1, so any string containing a zero must end at q_1. All other strings are of the form $1111 \dots 1$, which end at q_2.

Definition. Let $D = (\Sigma, Q, \delta, q_0, F), D' = (\Sigma, Q', \delta', q'_0, F')$ be deterministic automata. Then a map $f \colon Q \to Q'$ is called a *homomorphism* from D to D' if

(i) for all $q \in Q$ and $a \in \Sigma$, we have $\delta'(f(q), a) = f(\delta(q, a))$;
(ii) $f(q_0) = q'_0$;
(iii) $q \in F$ if and only if $f(q) \in F'$.

In particular, if f is bijective, it has an inverse and is called an *isomorphism*. We can show by induction that $\hat{\delta}'(f(q), w) = f(\hat{\delta}(q, w))$. Note that if a homomorphism f is not surjective, the states not in its range are not *accessible* from q'_0. If f is not injective, so $f(p) = f(q)$ for $p \neq q$, then we have $f(\hat{\delta}(p, w)) = \hat{\delta}'(f(p), w) = \hat{\delta}'(f(q), w) = f(\hat{\delta}(q, w))$; we will say that such states p, q are *indistinguishable*. We will observe that failure to be bijective only affects inaccessible states or pairs of indistinguishable states.

Proposition. Let f be a homomorphism (not *a priori* an isomorphism) from D to D'. Then $\mathcal{L}(D) = \mathcal{L}(D')$.

Proof. Let $w \in \mathcal{L}(D)$, so $\hat{\delta}(q_0, w) \in F$. Applying f, $f(\hat{\delta}(q_0, w)) = \hat{\delta}'(f(q_0), w) = \hat{\delta}'(q'_0, w) \in F'$ as required. All implications are bi-implications, so the converse holds. □

Theorem. Any language of the form $\mathcal{L}(D)$ for a deterministic automaton D is regular.

Proof. Let $D = (\Sigma, Q, \delta, q_0, F)$, and define a grammar $G = (\Sigma, V, P, S)$ by $V = Q, S = q_0$, and

$$P = \{p \to aq \mid \delta(p, a) = q\} \cup \{p \to a \mid \delta(p, a) \in F\}$$

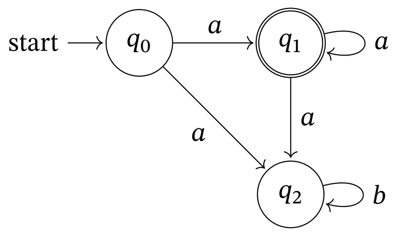
We will show $\mathcal{L}(D) = \mathcal{L}(G)$. Suppose $w = a_0 \dots a_n \in \mathcal{L}(D)$. Then $\hat{\delta}(q_0, w) \in F$, so there exist $q_0, \dots, q_{n+1}$ such that $q_{i+1} = \delta(q_i, a_i)$, and $q_{n+1} \in F$. By definition of G, this holds if and only if there exist $q_0, \dots, q_{n+1}$ such that $q_i \to a_i q_{i+1} \in P$ and $q_n \to a_n \in P$. This holds if and only if there exists $q_0, \dots, q_{n+1}$ such that $q_0 \overset{G}{\longrightarrow}_1 a_0 q_1 \overset{G}{\longrightarrow}_1 \dots \overset{G}{\longrightarrow}_1 a_0 \dots a_{n-1} q_n \overset{G}{\longrightarrow}_1 w$, so there exists a derivation $w \in \mathcal{L}(G)$. By regularity of G, all derivations are of this form, so we have bi-implications, and $\mathcal{L}(D) = \mathcal{L}(G)$. □

We will show that if L is a regular language, we can find a deterministic automaton D such that $L = \mathcal{L}(D)$. However, regular grammars can have multiple rules that may be used when reaching a single symbol, for instance $p \to aq$ and $p \to aq'$, so we cannot perform an obvious translation from this grammar into a deterministic automaton. To encapsulate this notion, we introduce nondeterministic automata.

3.3 Nondeterministic automata

Definition. A *nondeterministic automaton* is a tuple of the form $N = (\Sigma, Q, \delta, q_0, F)$ where Q is a finite set of states, $q_0 \in Q$, $F \subseteq Q \setminus \{q_0\}$, but in contrast with deterministic automata, we have $\delta : Q \times \Sigma \to \mathcal{P}(Q)$.

We interpret $\delta(q, a)$ as the set of possible states that the machine can transition into when reading a from state q. The graphical representation of such an automaton is the same.



Here, we define $\hat{\delta} : Q \times \mathbb{W} \to \mathcal{P}(Q)$, by the equations

$$\hat{\delta}(q, \varepsilon) = \{q\}; \quad \hat{\delta}(q, wa) = \bigcup_{p \in \hat{\delta}(q,w)} \delta(p, a)$$

This produces a unique *state set sequence*, not a deterministic state sequence. We define

$$\mathcal{L}(N) = \left\{ w \mid \hat{\delta}(q_0, w) \cap F \neq \varnothing \right\}$$

Remark. Deterministic automata can be seen as a special case of nondeterministic automata.

Theorem. Let N be a nondeterministic automaton. Then there exists a deterministic automaton D such that $\mathcal{L}(N) = \mathcal{L}(D)$.

Our proof will involve a *subset construction*.

Proof. Let $N = (\Sigma, Q, \delta, q_0, F)$. We define $D = (\Sigma, \mathcal{P}(Q), \Delta, \{q_0\}, G)$, where

$$\Delta(X, a) = \bigcup_{q \in X} \delta(q, a); \quad G = \{X \in \mathcal{P}(Q) \mid X \cap F \neq \varnothing\}$$

We show that these two automata produce the same language. Consider the state sequence of D on input w.

$$X_0 = \{q_0\}; \quad X_{i+1} = \bigcup_{q \in X_i} \delta(q, a_i)$$

The state set sequence of N on input w is

$$Y_0 = \{q_0\}; \quad Y_{i+1} = \bigcup_{q \in Y_i} \delta(q, a_i)$$

Clearly, these exactly match, so $X_i = Y_i$. So w is accepted by D if and only if it is accepted by N. □

Remark. Although this construction always works, we have transformed an automaton on n states into one on 2^n states.

Theorem. Let G be a regular grammar. Then there exists a nondeterministic automaton N such that $\mathcal{L}(G) = \mathcal{L}(N)$.

Proof. Let $G = (\Sigma, V, P, S)$. Let $H \notin \Sigma \cup V$ be a new symbol, known as the *halt state*. Let $Q = V \cup \{H\}$. Define $N = (\Sigma, Q, \delta, S, \{H\})$ where

$$\delta(A, a) = \begin{cases} \{B \mid A \to aB \in P\} & \text{if } A \to a \notin P \\ \{B \mid A \to aB \in P\} \cup \{H\} & \text{if } A \to a \in P \end{cases}$$

We claim that $\mathcal{L}(G) = \mathcal{L}(N)$. If $w \in L(G)$, we have a sequence $A_0, \dots, A_{n+1}$ of variables such that

$$S = A_0 \overset{G}{\longrightarrow}_1 \dots \overset{G}{\longrightarrow}_1 a_0 \dots a_{n+1} A_{n+1} \overset{G}{\longrightarrow}_1 w$$

In particular, $A_i \to a_i A_{i+1} \in P$ and $A_{n+1} \to a_n \in P$. By definition of δ, there exists a sequence $A_1, \dots, A_{n+1}$ such that $A_{i+1} \in \delta(A_i, a_i)$ and $H \in \delta(A_n, a_n)$. Hence $H \in \hat{\delta}(S, w)$, so $w \in \mathcal{L}(N)$. All implications are bi-implications so the converse holds. □

3.4 The pumping lemma for regular languages

Definition. A language L *satisfies the regular pumping lemma with pumping number n* if every word $w \in L$ with length at least n can be split into three parts $w = xyz$, such that $|y| > 0$, $|xy| \le n$ and for all $k \in \mathbb{N}$, we have $xy^kz \in L$. We call y a *pump* for the word xyz.

Theorem (regular pumping lemma)**.** Every regular language satisfies the pumping lemma.

Remark. If any word can be pumped, the language must be infinite.

Proof. Let L be a regular language. Then there exists a deterministic automaton D such that $L = \mathcal{L}(D)$. We show that L has pumping number $n = |Q|$. Let $w \in L(D)$ be a word with $|w| \ge n$. We can write $w = a_0 a_1 \dots a_{n-1} v$ where $v \in \mathbb{W}$.

The state sequence of D reading $a_0, \dots, a_{n-1}$ is $q_0, \dots, q_n$; it has length $n + 1$ since there are n state transitions. But there are only n states, so by the pigeonhole principle, one state must repeat. Let

$i < j \leq n$ such that $q_i = q_j$. Let $x = a_0 \dots a_{i-1}, y = a_i \dots a_{j-1}, z = a_j \dots a_{n-1}v$, so we have $xyz = w$, $|y| > 0$, $|xy| \leq n$ by construction.

We show that we can pump the word. After reading x, we have $\hat{\delta}(q_0, x) = q_i$, and $\hat{\delta}(q_i, y) = q_j = q_i$, and finally $\hat{\delta}(q_i, z) = \hat{\delta}(q_j, z) \in F$. Hence, $\hat{\delta}(q_0, xy^k) = q_i$ by induction on k. In particular, $\hat{\delta}(q_0, xy^k z) \in F$ as required. □

Example. Let $L = \{0^k 1^k, k > 0\}$. We claim this is not a regular language. Suppose L is regular, and has pumping number N. Consider the word $0^N 1^N \in L$; this word has more than N letters, so the word can be pumped. The pump must lie in the first N letters, all of which are zeroes. Pumping down, $0^{N-\ell} 1^N \in L$ where ℓ is the length of the pump. This is a contradiction since the length of the pump is nonzero. Note that this language is context-free, so we know that the inclusion of regular languages in context-free languages is proper.

Example. Let $n > 0$, and let $L = \{0^n w, w \in \mathbb{W}\}$. We show this is regular, but any deterministic automaton D such that $L = \mathcal{L}(D)$ has more than n states. For regularity, we can simply write down a grammar.

$$P = \{S \to 0X_0, X_0 \to 0X_1, \dots, X_{n-2} \to 0X_{n-1}, X_{n-2} \to 0,$$
$$X_{n-1} \to 0, X_{n-1} \to 1, X_{n-1} \to 0X_{n-1}, X_{n-1} \to 1X_{n-1}\}$$

This has exactly $n+1$ states. Suppose that an automaton with at most n states has the same language. Then L satisfies the pumping lemma with pumping number n. In particular, we can pump down the word 0^n, obtaining a word with fewer zeroes, and this is not in the language.

Example. Some non-regular languages also satisfy the pumping lemma. Let $\Sigma = \{0, 1\}$. If a word $w \in \mathbb{W}$ contains at least one zero, we say the *tail* of the word is the number of ones that follow the last zero. Let $X \subseteq \mathbb{N}$ be an arbitrary set of naturals, and define L_X to be the set of words that contain no zeroes, or have a tail which lies in X. If $X \neq Y$, we have $L_X \neq L_Y$, so L is an injection from $\mathcal{P}(\mathbb{N})$ to the space of languages on Σ. Since there are uncountably many $X \subseteq \mathbb{N}$, but there are only countably many regular languages, there must be some non-regular languages of the form L_X.

We claim that all L_X satisfy the pumping lemma, so then there must be some L_X which are non-regular which satisfy the pumping lemma. Let $X \subseteq \mathbb{N}$; we claim this has pumping number 2. Let $w \in L_x$ such that $|w| \geq 2$.

Suppose w starts with a zero, so $w = 0z$. Then let $x = \varepsilon, y = 0$, so $w = xyz$. Pumping up does not change the tail; pumping down either does not change the tail or there are now no zeroes, but in either case, the new word lies in the language.

Conversely, suppose w starts with a one, so $w = 1z$. Let $x = \varepsilon, y = 1$, so $w = xyz$ as before. If z contains no zeroes, after pumping y, there are still no zeroes, so the new word is in the language. If z contains a zero, there is a tail, and pumping y does not influence the tail. Hence, the pumping lemma is satisfied.

3.5 Closure properties

We have already shown that regular languages are closed under concatenation and union. We will now show that they are closed under complement, intersection, and difference. For this, it suffices to show they are closed under complement, because intersection and difference can be expressed in terms of complement and union.

Consider an automaton $D = (\Sigma, Q, \delta, q_0, F)$. Without loss of generality, we can ensure that $\delta(q_i, a) \neq q_0$ for all i, a. Now define $D' = (\Sigma, Q, \delta, q_0, \mathbb{Q} \setminus (F \cup \{q_0\}))$. Then, $\mathcal{L}(D') = \mathbb{W}^+ \setminus \mathcal{L}(D)$.

There is an alternative construction to obtain union and intersection, known as the *product automaton* construction. Let $D = (\Sigma, Q, \delta, q_0, F)$ and $D' = (\Sigma, Q', \delta', q_0', F')$. We can define the pointwise product $D'' = (\Sigma, Q \times Q', \delta \times \delta', (q_0, q_0'), F'')$, where $(\delta \times \delta')((q, q'), a) = (\delta(q, a), \delta'(q', a))$, and either $F'' = \{(q, q') \mid q \in F, q' \in F'\}$ or $F'' = \{(q, q') \mid q \in F \text{ or } q' \in F'\}$. We can see that the language generated by this new automaton is $\mathcal{L}(D) \cap \mathcal{L}(D')$ or $\mathcal{L}(D) \cup \mathcal{L}(D')$.

3.6 Emptiness problem

Lemma. Let L be a nonempty regular language with pumping number n. Then there is a word $w \in L$ such that $|w| < n$.

Proof. Let w be a word in L. If $|w| < n$, we are already done. Otherwise, it can be pumped down into a smaller word. By induction, we can obtain a word of length less than n. □

Corollary. The emptiness problem for regular grammars is solvable.

Proof. Given a regular grammar, we can obtain its pumping number. We can check every word below this length because the word problem is solvable; if no words are accepted, the language is empty. □

3.7 Regular expressions

Definition. The *Kleene star* operation on a language L, written $L^\star$, is given by

$$L^\star = \{w \mid \exists \text{ sequence of words in } L, w = \text{their concatenation}\}$$

In particular $\varepsilon \in L^\star$. The *Kleene plus* operation is $L^+ = L^\star \setminus \{\varepsilon\}$.

Definition. A *regular expression* on an alphabet Σ is defined inductively by:

(i) the symbol $\varnothing$ is a regular expression;
(ii) ε is a regular expression;
(iii) for all a in Σ, a is a regular expression;
(iv) if R, S are regular expressions, $(R + S)$ is a regular expression;
(v) if R, S are regular expressions, (RS) is a regular expression;
(vi) if R is a regular expression, $R^\star$ is a regular expression;
(vii) if R is a regular expression, R^+ is a regular expression.

By definition, nothing else is a regular expression. By recursion, we can assign a language $\mathcal{L}(E)$ to each regular expression E.

(i) $\mathcal{L}(\varnothing) = \varnothing$;
(ii) $\mathcal{L}(\varepsilon) = \{\varepsilon\}$;
(iii) for $a \in \Sigma$, $\mathcal{L}(a) = \{a\}$;
(iv) if R, S are regular expressions, $\mathcal{L}(R + S) = \mathcal{L}(R) \cup \mathcal{L}(S)$;

(v) if R, S are regular expressions, $\mathcal{L}(RS) = \mathcal{L}(R)\mathcal{L}(S)$;
(vi) if R is a regular expression, $\mathcal{L}(R^\star) = \mathcal{L}(R)^\star$;
(vii) if R is a regular expression, $\mathcal{L}(R^+) = \mathcal{L}(R)^+$.

Note that rules (iv) and (v) introduce parentheses, occasionally unnecessarily. When the meaning is unambiguous, these parentheses are omitted. The binding power of concatenation RS is higher than union $R + S$, so we can write $RS + T$ for $((RS) + T)$.

We say that a language is *essentially regular* if there is a regular language L' such that $L = L'$ or $L = L' \cup \{\varepsilon\}$.

Theorem. If E is a regular expression, $\mathcal{L}(E)$ is essentially regular.

This is an equivalence, but the converse (often called Kleene's algorithm) is not required for this course.

Proof. Observe that (i), (ii), (iii) are essentially regular languages, so it suffices to show that essentially regular languages are closed under (iv), (v), (vi), (vii). We have already shown that regular languages are closed under union and concatenation, and the proof for essentially regular languages follows easily. Note that $\mathcal{L}(E^\star) = \mathcal{L}(E + E^+)$, so it suffices to show closure of regular languages under the Kleene plus; we can then perform case analysis to prove the same for essentially regular languages.

Let $G = (\Sigma, V, P, S)$ be a regular grammar. Let $P^+ = P \cup \{A \to aS \mid A \to a \in P\}$. It suffices to show that $G^+ = (\Sigma, V, P^+, S)$ has the language $\mathcal{L}(G^+) = \mathcal{L}(G)^+$.

Suppose $w \in \mathcal{L}(G)^+$, so $w = w_0 \dots w_n$ for $w_i \in \mathcal{L}(G)$. If $n = 0$, $w \in \mathcal{L}(G)$ and any derivation can be translated easily into G^+. Otherwise, suppose $w_0 \dots w_{n-1} \in \mathcal{L}(G^+)$ by induction. Therefore there is a derivation $S \xrightarrow{G^+} w_0 \dots w_{n-1}$. This derivation ends with a terminal rule $A \to a$, so we can replace it with a nonterminal rule $A \to aS$, giving $S \xrightarrow{G^+} w_0 \dots w_{n-1}S \xrightarrow{G} w_0 \dots w_{n-1}w_n$, so $w \in \mathcal{L}(G)$ as required.

Now suppose $w \in \mathcal{L}(G^+)$. Without loss of generality we can assume that G is ε-adequate, so S does not occur on the right-hand side of a rule. Suppose we have a derivation $S \xrightarrow{G^+} w$. Let n be the number of times that S occurs in the derivation. We then prove $w \in \mathcal{L}(G)^+$ by induction on n. n cannot be zero. Suppose all words $v \in \mathcal{L}(G^+)$ lie in $\mathcal{L}(G)^+$ if they have a derivation with $n - 1$ occurrences of S. Since $n \geq 1$, we have $S \xrightarrow{G^+} vS \xrightarrow{G^+} w$ where vS is the last occurrence of S in the derivation of w. In particular, $S \xrightarrow{G^+} v$ with $n - 1$ occurrences, since the last rule of $S \xrightarrow{G^+} vS$ is one of the added rules in P^+. By induction, $v \in \mathcal{L}(G)^+$. Since $vS \xrightarrow{G^+} w$, we know that $w = vw'$ by considering the possible derivations in regular languages. Hence $S \xrightarrow{G^+} w'$ with only one occurrence of S at the start. In particular none of our new rules were used, so $S \xrightarrow{G} w'$, so $w' \in \mathcal{L}(G)^+$, hence $w \in \mathcal{L}(G)^+$. □

3.8 Minimisation of deterministic automata

Definition. A state q is called *accessible* if there is a word w such that $q = \hat{\delta}(q_0, w)$. A state that is not accessible is called *inaccessible*.

Definition. States q and q' are *distinguished* by a word w if $\hat{\delta}(q, w) \in F$ and $\hat{\delta}(q', w) \notin F$, or vice versa. States that are distinguished by some word are called *distinguishable*. States that are not distinguished by any word are called *indistinguishable*.

If $f : Q \to Q'$ is a homomorphism, then

(i) if p, q are distinguishable, $f(p) \neq f(q)$;

(ii) if $q' \in Q'$ is accessible, q' lies in the range of f.

In particular, if f is a homomorphism from D to D' and all pairs of nonequal states in D are distinguishable, f is injective; if all states in D' are accessible, f is surjective.

Definition. An automaton D is called *irreducible* if all pairs of nonequal states are distinguishable and all states are accessible.

Hence, any homomorphism between irreducible automata is an isomorphism.

Defining $q \sim q'$ if q and q' are indistinguishable, $\sim$ is an equivalence relation. As usual, we write $[q]$ for the equivalence class of states indistinguishable from q. We can therefore define the *quotient automaton* by

$$D/\sim \; = \left(\Sigma, Q/\sim, [\delta], [q_0], [F]\right); \quad [\delta]([q], a) = [\delta(q, a)]; \quad [F] = \{[q] \mid q \in F\}$$

Note that if an equivalence class contains an accept state, the class is completely contained in F, so being an accept state is a class property. The map $[\delta]$ is well-defined: indeed, if $q \sim q'$, we have $\delta(q, a) \sim \delta(q', a)$, because if $\delta(q, a) \nsim \delta(q', a)$, there would exist a word w that distinguishes these two states, but then aw would distinguish q and q'.

If $q \nsim q'$, we can show the two states are distinguished in the quotient automaton. By induction, $[\hat{\delta}]([q], w) = [\hat{\delta}(q, w)]$. Suppose without loss of generality that $\hat{\delta}(q, w) \in F$, $\hat{\delta}(q', w) \notin F$. Then $[\hat{\delta}]([q], w) \in [F]$, but $[\hat{\delta}]([q'], w) \notin [F]$. So w distinguishes $[q]$ and $[q']$. In particular, each pair of nonequal states is distinguishable.

Note further that $\mathcal{L}(D) = \mathcal{L}\left(D/\sim\right)$, because the quotient map $q \mapsto [q]$ is a homomorphism. If D had no inaccessible states, $D/\sim$ also has no inaccessible states, since the quotient map is surjective.

Theorem. For every deterministic automaton, there is an irreducible deterministic automaton I such that $\mathcal{L}(D) = \mathcal{L}(I)$.

Proof. Let $A \subseteq Q$ be the set of accessible states in D. Let $D^\star = \left(\Sigma, A, \delta|_{A \times \Sigma}, q_0, F \cap A\right)$. The inclusion map from $D^\star$ to D is a homomorphism, so their languages are the same. Now let $I = D^\star/\sim$. By the above discussion, I is irreducible and has the same language as $D^\star$. □

Remark. The number of states in I is at most the number of states in D.

Theorem. If I, I' are irreducible deterministic automata and $\mathcal{L}(I) = \mathcal{L}(I')$, then I and I' are isomorphic.

Proof. It suffices to construct a homomorphism between the two automata, since any homomorphism between irreducible automata is an isomorphism. Let $I = (\Sigma, Q, \delta, q_0, F)$ and $I' = (\Sigma, Q', \delta', q_0', F')$, and without loss of generality let $Q \cap Q' = \varnothing$. We can extend $\sim$ to $Q \cup Q'$, by defining $q \sim q'$ if for all w, $\hat{\delta}(q, w) \in F$ if and only if $\hat{\delta}'(q', w) \in F'$. We know $q_0 \sim q_0'$, because by assumption, the languages of the two automata are the same.

We show that for all $q \in Q$, there exists $q' \in Q'$ such that $q \sim q'$. Let $\mathrm{sp}(q)$ be the length of the shortest path from q_0 to q. Since I is irreducible, this is well-defined and finite for all $q \in Q$. We prove this claim by induction on $\mathrm{sp}(q)$. The base case is $\mathrm{sp}(q) = 0$ so $q = q_0$, and we have already shown $q_0 \sim q_0'$ as required.

Now suppose $\mathrm{sp}(q) = k + 1$. Then there exists $p \in Q$ and $a \in \Sigma$ such that $\delta(p, a) = q$ and $\mathrm{sp}(p) = k$. By the induction hypothesis, we can find $p' \in Q'$ such that $p \sim p'$. Then let $q' = \delta'(p', a)$, then $q' \sim \delta(p, a) = q$. Hence each $q \in Q$ has a $q' \in Q'$ such that $q \sim q'$.

We now will show that if $q' \sim q \sim p'$, we have $q' = p'$. By transitivity, $q' \sim p'$, but by irreducibility of I', $q' = p'$.

Because of the above results, we can construct a function $f : Q \to Q'$ defined by $q \mapsto q'$ where $q \sim q'$. This is well-defined and unique. We now claim f is a homomorphism. Since $q_0 \sim q_0'$, we have $f(q_0) = q_0'$. The requirement $q \in F \iff f(q) \in F'$ follows by definition of $\sim$. Now fix $q \in Q$ and $q' = f(q)$, so $q \sim q'$. Then, $\delta(q, a) \sim \delta'(q', a)$, so $f(\delta(q, a)) \sim \delta'(q', a) = \delta'(f(q), a)$. □

Remark. There is a unique (up to isomorphism) irreducible automaton that accepts a given regular language, and its size is smaller than all other automata that accept the same language.

3.9 Equivalence problem

We have already solved the word problem for noncontracting grammars and the emptiness problem for regular grammars. To solve the equivalence problem, we will construct minimal automata for two given regular grammars, and check whether they are isomorphic; if so, the languages are the same, and otherwise, the languages are different. We must check that this idea can be formulated into an algorithm which must complete in finitely many steps.

Proposition. Let D be a deterministic automaton and $q \in Q$ a state. Then it is solvable whether q is accessible.

Proof. If there is a word w such that $\hat{\delta}(q_0, w) = q$, then the shortest such word has length at most $|Q|$, which can be easily proven using the technique from the pumping lemma. We can explicitly check each word of length at most $|Q|$. □

Theorem (the table filling algorithm)**.** Let D be a deterministic automaton and $q, q' \in Q$ states. Then the proposition $q \sim q'$ is solvable.

Proof. Form a matrix A with entries indexed by $Q \times Q$. The entry indexed by (q, q') contains information about distinguishability of q, q'. In particular, $A_{q,q'}$ contains either nothing or a word w distinguishing q and q'. Since $\sim$ is an equivalence relation, it suffices to consider the upper triangular part of the matrix, excluding the diagonal. To initialise the matrix, if $q \in F$ and $q' \notin F$ we set $A_{q,q'} = \varepsilon$, since the empty word distinguishes q, q'.

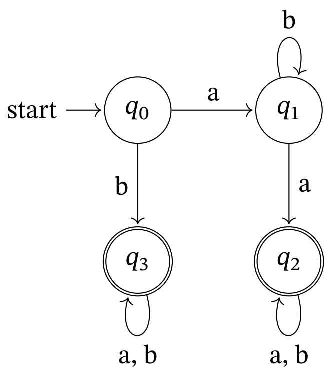
Then, for each $q, q' \in Q$ that do not have a filled entry $A_{q,q'}$ already, and for each $a \in \Sigma$, we can check the entry indexed by $(\delta(q, a), \delta(q', a))$. If these two states are distinguished by a word w, q and q' are distinguished by aw. So we can set $A_{q,q'} = aw$. This single step will terminate in a finite amount of time, on the order of $|Q|^2|\Sigma|$-many steps.

We then repeat this inductive step until no more assignments into the matrix can be made in an single iteration. This will happen in finitely many steps.

We now must show that after this process completes, $A_{q,q'}$ contains a word w if and only if q and q' are distinguishable, and in this case, w distinguishes q and q'. If $A_{q,q'}$ contains a word w, it is clear that w distinguishes q and q', since $\hat{\delta}(q, w) \in F$ and $\hat{\delta}(q', w) \notin F$ or vice versa. Now suppose there exists a word w that distinguishes some states q and q', but q, q' are unmarked in A. Let q, q' be a pair of states with a distinguishing word w of minimal length.

Either $w = \varepsilon$ or $w = av$. If $w = \varepsilon$, $q \in F$ and $q' \notin F$ or vice versa, so $A_{q,q'}$ is marked. Otherwise, $w = av$. Since v is shorter than the smallest word that distinguishes two states that are not marked in A, we must have that the entry $(\delta(q, a), \delta(q', a))$ is marked with some word in A. So at some step in the algorithm, this entry was added into A. But then the algorithm would mark q, q' with a word in the next step. □

Example. Consider the following automaton.



In step zero, we find

	q_0	q_1	q_2	q_3
q_0			ε	ε
q_1			ε	ε
q_2				
q_3				

In step one, checking $\delta(q_0, a)$ and $\delta(q_1, a)$, we arrive at

	q_0	q_1	q_2	q_3
q_0		a	ε	ε
q_1			ε	ε
q_2				
q_3				

The only remaining entry is (q_2, q_3), and this is not filled in a single step. Hence $q_2 \sim q_3$.

Corollary. The equivalence problem for regular grammars is solvable.

Hence, for regular grammars, all of our desirable closure properties are true, and all of our motivating decision problems are solvable.

4 Context-free languages

4.1 Trees

Recall that the language $\{0^k 1^k \mid k > 0\}$ is context-free but not regular, so context-free languages are indeed a proper superset of regular languages. The structure of regular derivations was very simple; each intermediate step was of the form wA for a word w and a variable $A \in V$. However, the structure of context-free derivations is more complicated: we use a parse tree instead of a linear derivation.

Definition. A set $T \subseteq \mathbb{N}^\star$ is called a *(finitely-branching) tree* if it is closed under initial segments, and for every $t \in T$, there is a *branching number* $n \in \mathbb{N}$ such that for all k, the sequence tk lies in T if and only if $k < n$. A node $t \in T$ with no sucessors is called a *leaf*. The empty sequence, which is an element of every tree, is called the *root*. A node $t \in T$ has *level* k if the length of the sequence is k, so $|t| = k$. If T is finite, there is a maximum level, called the *height* of the tree. For a node $t \in T$, the sequence $t|_0, t|_1, \dots, t|_{|t|} = t$ is called the *branch* leading to t.

Example. This is an example of a tree.

$$\begin{tikzcd}
& & & \varepsilon \arrow[dl] \arrow[d] \arrow[dr] & & & & \\
& & 0 \arrow[dll] \arrow[dl] & 1 \arrow[d] & 2 \arrow[d] \arrow[dr] \arrow[drr] & & & \\
00 \arrow[d] & 01 \arrow[d] & & 10 \arrow[dl] \arrow[d] & 20 & 21 \arrow[d] & 22 \arrow[d] \arrow[dr] & \\
000 & 001 \arrow[d] & 100 & 101 & & 210 & 220 & 221 \\
& 0010 & & & & & &
\end{tikzcd}$$

Definition. Let T be a tree and $t \in T$. Then $T_t = \{s \mid ts \in T\}$ is the *subtree starting from* t.

Definition. We define a partial order on T by $t < s$ if $t \neq s$ and if there exists k such that $t(k) \neq s(k)$ and k_0 is minimal with this property, then $t(k_0) < s(k_0)$. This is called the *left-to-right order*.

Remark. This order is only a partial order since it does not order two distinct nodes that lie on the same branch, for example, 0 and 00. For each level k, the nodes of length k are totally ordered. The leaves are totally ordered.

4.2 Parse trees

Definition. Let G be a context-free grammar. A pair $\mathbb{T} = (T, \ell)$ is a *G-parse tree* if T is a finitely-branching tree and $\ell : T \to \Omega$ is a *labelling function* such that:
(i) $\ell(\varepsilon) \in V$, we say T *starts with* $\ell(\varepsilon)$;
(ii) if $\ell(t) \in \Sigma$, t has no successors;
(iii) if t has $n + 1$ successors and $\ell(t) = A \in V$, then $A \to \ell(t0)\ell(t1) \dots \ell(tn) \in P$.
If $\mathbb{T} = (T, \ell)$ is a G-parse tree, and $t_0, \dots, t_m$ are its leaves written in the left-to-right order, then the *string parsed by* $\mathbb{T}$ is $\sigma_{\mathbb{T}} = \ell(t_0) \dots \ell(t_m)$.

Remark. If $t \in T$, $\sigma_{\mathbb{T}} = \alpha\sigma_{\mathbb{T}_t}\beta$ where $\mathbb{T}_t = (T_t, \ell_t)$, $\ell_t(s) = \ell(ts)$.

Proposition. Let G be a context-free grammar. Then $w \in \mathcal{L}(G)$ if and only if there is a G-parse tree $\mathbb{T}$ starting from S such that $\sigma_{\mathbb{T}} = w$.

Proof. Observe that certain sequences of parse trees correspond to derivations. In particular, a sequence $\mathbb{T}_0, \dots, \mathbb{T}_n$ of G-parse trees is *derivative* if $\mathbb{T}_0 = (\{\varepsilon\}, \ell_0)$ with $\ell_0(\varepsilon) = S$, and $T_{i+1} \supseteq T_i$ is constructed by considering a leaf $t \in T_i$ such that $\ell_i(t) = A \in V$ and $A \to x_0 \dots x_n \in P$, and giving it $n + 1$ successors with $\ell_{i+1}(tk) = x_k$. There is a one-to-one correspondence between G-derivations starting from S and such derivative sequences of parse trees. In particular, any derivation yields a derivative sequence of parse trees, and hence the last parse tree in the sequence has $\sigma_{\mathbb{T}_n} = w$.

Conversely, given a parse tree $\mathbb{T}$, it suffices to construct such a derivative sequence of parse trees, because then the correspondence yields a derivation as required. We start with the trivial tree $\mathbb{T}_0 = \left(\{\varepsilon\}, \ell|_{\{\varepsilon\}}\right)$. In each step, suppose $\mathbb{T}_0, \dots, \mathbb{T}_i$ already form a derivative sequence, and $T_i \neq T$. Let $t \in T \setminus T_i$. Then there is a terminal node in T_i on the branch containing t in T, which is not a terminal node in T. We can then create T_{i+1} by adding the T-successors of t to T_i. Since T is finite, after finitely many steps we are done. In particular, $\mathbb{T}_0, \dots, \mathbb{T}_n$ is a derivative sequence, and thus $S \xrightarrow{G} \sigma_{\mathbb{T}_n} = \sigma_{\mathbb{T}} = w$ as required. □

Suppose $\mathbb{T}$ is a parse tree and $t \in T$ such that $\ell(t) = A$, and $\mathbb{T}'$ is a parse tree starting from A. Then, we can remove the subtree T_t, and replace it with T', which also yields a parse tree. This technique is known as *grafting*.

Definition. We define $\mathrm{graft}(\mathbb{T}, t, \mathbb{T}') = (S, \ell^\star)$ where

$$S = \{s \in T \mid t \not\subseteq s\} \cup \{tu \mid u \in T'\}$$

and

$$\ell^\star(s) = \begin{cases} \ell(s) & t \not\subseteq s \\ \ell'(u) & \exists u \in T',\, s = tu \end{cases}$$

Then we have

$$\sigma_{\mathrm{graft}(\mathbb{T},t,\mathbb{T}')} = \alpha\sigma_{\mathbb{T}'}\beta; \quad \sigma_{\mathbb{T}} = \alpha\sigma_{\mathbb{T}_t}\beta$$

4.3 Chomsky normal form

Definition. A grammar is in *Chomsky normal form* if all of its rules are of the form $A \to BC$ or $A \to a$. Rules of the form $A \to BC$ are called *binary*; rules of the form $A \to a$ are called *unary*.

Every grammar in Chomsky normal form is context-free.

Lemma. Let G be a grammar in Chomsky normal form, and $w \in \mathcal{L}(G)$ with $|w| = n$. Then every G-derivation of w has length $2n - 1$.

Proof. Binary rules increment the length, and increment the variable count. Unary rules preserve the length, and decrement the variable count. Since w is comprised only of letters, exactly $n - 1$ binary rules and n unary rules were used. $\square$

We will show that every context-free grammar is equivalent to a Chomsky normal form grammar, and there is an algorithm to produce such a grammar. There are three types of rules that are obstructions to a context-free grammar being in Chomsky normal form:

(i) rules $A \to \alpha$ where $|\alpha| \geq 2$ and α contains a letter;

(ii) rules of the form $A \to B$, called *unit rules*.

(iii) rules $A \to \alpha$ where $|\alpha| > 2$ and α contains only variables.

Suppose we have a rule of the form $A \to \alpha$ where $|\alpha| \geq 2$, and α contains a letter. For each letter $a \in \Sigma$, we can add a variable X_a and a rule $X_a \mapsto a$. Then we convert α to $X(\alpha)$, where X is the map converting each a into X_a. Then α contains no letter. We can therefore suppose without loss of generality that a given context-free grammar has no rules of this form.

Now consider a unit rule $A \to B$. A grammar is called *unit closed* if for all $A \to B \in P$ and $B \to \alpha \in P$, we also have $A \to \alpha \in P$. We can easily convert each grammar into an equivalent unit closed grammar by adding at most $|V| \cdot |P|$ new rules. If a context-free grammar G is unit closed, we will show that we can remove all unit rules to give a grammar G' without changing the language. Clearly $\mathcal{L}(G') \subseteq \mathcal{L}(G)$. Suppose $w \in \mathcal{L}(G)$, then w has a shortest G-derivation. Suppose this G-derivation of w contains a unit rule, so

$$S \xrightarrow{G} \alpha A\beta \xrightarrow{G}_1 \alpha B\beta \xrightarrow{G} w$$

where this use of the unit rule is the last such usage. Since w contains no variables, we must have applied a rule $B \overset{G}{\longrightarrow}_1 \zeta$.

$$S \overset{G}{\longrightarrow} \alpha A\beta \overset{G}{\longrightarrow}_1 \alpha B\beta \overset{G}{\longrightarrow} \gamma B\delta \overset{G}{\longrightarrow}_1 \gamma\zeta\delta \overset{G}{\longrightarrow} w$$

where the $B \overset{G}{\longrightarrow}_1 \zeta$ is the first usage of a rule for B. Since $\alpha B\beta \overset{\gamma}{\to} B\delta$ did not use any B-rule by assumption, by unit closure we can replace this derivation with

$$S \overset{G}{\longrightarrow} \alpha A\beta \overset{G}{\longrightarrow} \gamma A\delta \overset{G}{\longrightarrow}_1 \gamma\zeta\delta \overset{G}{\longrightarrow} w$$

This is clearly shorter. So the shortest G-derivation contains no use of a unit rule, so is also a G'-derivation.

Finally, let us consider a rule $A \to \alpha$ where $|\alpha| > 2$ and α contains only variables. Suppose $\alpha = A_1 \dots A_n$. We define new variables $X_1, \dots, X_{n-2}$, and add new rules $A \to A_1X_1, X_1 \to A_2X_2, \dots X_{n-2} \to A_{n-1}A_n$. Then, performing this for all such rules, we obtain a grammar without any such rules. This grammar is in Chomsky normal form.

Theorem (Chomsky). Let G be a context-free grammar. Then we can compute an equivalent context-free grammar G' in Chomsky normal form.

Proof. Remove problems due to rules of the form $A \to \alpha$ where α contains a letter and has length at least 2. Form the unit closure, then remove unit rules. Remove problems due to rules of the form $A \to A_1A_2A_3 \dots A_n$. $\square$

4.4 The pumping lemma for context-free languages

Definition. Let L be a context-free language, and let $n \in \mathbb{N}$. Suppose that for all $w \in L$ such that $|w| \geq n$, there are x, y, z, u, v such that $w = xuyvz$, and $|uyv| \leq n$, $|uv| > 0$, and for all k, $xu^kyv^kz \in L$. Then L satisfies the *pumping lemma for context-free languages with pumping number n*. We call u, v the *pump*.

Remark. The pump now has two parts, and one part may be the empty string. There is no longer a constraint on the position of the pump in a word; x and z may be of any length. The regular pumping lemma implies the context-free pumping lemma. Since there are uncountably many languages satisfying the regular pumping lemma, there are also uncountably many languages satisfying the context-free pumping lemma. In particular, the context-free pumping lemma cannot characterise the countable class of all context-free languages.

Theorem. Every context-free language satisfies the context-free pumping lemma for some pumping number n.

Proof. Let L be a context-free language. Then L has a Chomsky normal form grammar $G = (\Sigma, V, P, S)$, so $\mathcal{L}(G) = L$. Let $m = |V|$, and $n = 2^m$. We claim n is a pumping number for G.

If $\mathbb{T}$ is a G-parse tree where the height of $\mathbb{T}$ is $h+1$ and $\sigma_{\mathbb{T}}$ is a word, then $|\sigma_{\mathbb{T}}| \leq 2^h$. Indeed, the largest possible tree of height $h + 1$ has 2^{h+1} leaves. Since $\sigma_{\mathbb{T}}$ is a word, we must have applied a unary rule for each letter. Every unary rule reduces the amount of leaves by one. Thus, the tree must contain $2^{h+1} - |\sigma_{\mathbb{T}}|$ leaves. Hence $|\sigma_{\mathbb{T}}| \leq 2^h$.

Consider a word $w \in \mathcal{L}(G)$ with $|w| \geq 2^m = n$. Then, if $\mathbb{T}$ is a G-parse tree of w, so $\sigma_{\mathbb{T}} = w$, we know by the previous claim that the height of $\mathbb{T}$ is at least $m + 1$. Let $t \in T$ such that the length of the branch to t is the height $h \geq m+1$ of the tree. Then, the path from ε to t has $h+1$ labels, so contains h variables and one letter.

Let s be an element of the branch of t such that the height of the subtree T_s is exactly $m + 1$. Hence $\left|\sigma_{\mathbb{T}_s}\right| \leq 2^m = n$. In particular, the path from s to t has $m+2$ labels, so contains exactly $m+1$ variables and one letter. By the pigeonhole principle, there are two nodes $t_0 \subsetneq t_1$ on the branch from s to t with the same label $A \in V$. Let

$$\sigma_{\mathbb{T}} = a\sigma_{\mathbb{T}_s}b; \quad \sigma_{\mathbb{T}_s} = x'\sigma_{\mathbb{T}_{t_0}}z'; \quad \sigma_{\mathbb{T}_{t_0}} = u\sigma_{\mathbb{T}_{t_1}}v; \quad \sigma_{\mathbb{T}_{t_1}} = y$$

Then let $x = ax'; \quad z = z'b$ in the definition of the pumping lemma. Since $t_0 \neq t_1$, we have $|uv| > 0$. Note that $uyv = \sigma_{\mathbb{T}_{t_0}}$, which has length at most $\left|\sigma_{\mathbb{T}_s}\right|$, which has length at most $2^m = n$.

Pumping down is accomplished by grafting T_{t_1} into T_{t_0}; conversely, pumping up is accomplished by iteratively grafting T_{t_0} into T_{t_1}. Define $\mathbb{T}_{(0)} = \mathbb{T}_{t_1}$, and $\mathbb{T}_{(i+1)} = \operatorname{graft}(\mathbb{T}_{t_0}, t_1, \mathbb{T}_{(i)})$. Then $\mathbb{T}_k = \operatorname{graft}(\mathbb{T}, t_1, \mathbb{T}_{(k)})$, and $\sigma_{\mathbb{T}_k} = xu^k yv^k z$, thus proving the pumping lemma. □

Example. Consider the language $L = \{a^n b^n c^n \mid n > 0\}$ is not context-free. Suppose it is context-free. Then it has a pumping number $N \in \mathbb{N}$. Consider the word $a^N b^N c^N \in L$. Then $a^N b^N c^N = xuyvz$ where $|uyv| \leq N$, $|uv| > 0$. Since $|uyv| \leq N$, the string uyv cannot consist of all three letters a, b, c. In any case, pumping up the string will increase the quantity of one letter, but not increase the quantity of some other letter. Then the new word does not lie in L.

4.5 Closure properties

We have seen that $L = \{a^n b^n c^n \mid n > 0\}$ is not context-free. However, $L_0 = \left\{a^n b^n c^k \mid n, k > 0\right\}$ and $L_1 = \left\{a^k b^n c^n \mid n, k > 0\right\}$ are context-free, as the concatenation of context-free languages. But the intersection $L_0 \cap L_1$ is exactly L, so context-free languages are not closed under intersection. Therefore, they are also not closed under complement or difference, because this, along with closure under union, would imply closure under intersection. Note that any model of computation corresponding to context-free grammars cannot have a product construction, because such a construction would imply closure of context-free languages under intersection.

It can be shown that context-free languages correspond to *pushdown automata*, which are similar to deterministic automata, except that they also have a *stack*, which is a way of storing a sequence of arbitrary symbols. The stack has a *push* operation allowing a symbol to be pushed onto the top of the stack, and a *pop* operation that removes the topmost element currently on the stack. In particular, the stack does not have random access, and any symbol pushed can be popped at most once. Sequences that are pushed onto the stack are popped off in reverse order.

The transition function δ has an additional input denoting the topmost element currently on the stack, and an additional output describing an operation to perform on the stack, if any.

Theorem. A language is context-free if and only if there is a pushdown automaton for the language.

4.6 Decision problems

The word problem is already solved for context-free languages. The emptiness problem can be solved by the pumping lemma, similarly to the solution for regular languages. Indeed, if n is a pumping number, no word with length at most n can be the shortest word, since it can be pumped down. So we can explicitly check each word of length less than n to solve the emptiness problem. Note that the choice of pumping lemma to use does not matter in this argument.

5 Register machines

5.1 Definition

A register machine uses an alphabet Σ, has finitely many states, and finitely many *registers*, which are last-in first-out storage units containing a word $w \in \mathbb{W}$. The machine is able to access the last letter of the word, remove it, or push a new letter. A *configuration* or *snapshot* of length $n + 1$ is a tuple of the form $(q, w_0, \dots, w_n) \in Q \times \mathbb{W}^{n+1}$. A configuration defines the state of the computation at a particular point in time.

The transition function should now be of the form $\delta : Q \times \mathbb{W}^{n+1} \to Q \times \mathbb{W}^{n+1}$. However, not every such function represents a real computation; there are uncountably many such functions, and the action on the registers is unrestricted.

Definition. Let Σ be an alphabet, and Q be a nonempty finite set of states. A tuple of the form

$$\begin{gathered}(0, k, a, q) \in \mathbb{N} \times \mathbb{N} \times \Sigma \times Q \\ (1, k, a, q, q') \in \mathbb{N} \times \mathbb{N} \times \Sigma \times Q \times Q \\ (2, k, q, q') \in \mathbb{N} \times \mathbb{N} \times Q \times Q \\ (3, k, q, q') \in \mathbb{N} \times \mathbb{N} \times Q \times Q\end{gathered}$$

is called a (Σ, Q)-*instruction*. For improved readability, we write

$$\begin{aligned}+(k, a, q) &= (0, k, a, q) \\ ?(k, a, q, q') &= (1, k, a, q, q') \\ ?(k, \varepsilon, q, q') &= (2, k, q, q') \\ -(k, q, q') &= (3, k, q, q')\end{aligned}$$

Intuitively,

- $+(k, a, q)$ represents pushing the letter a onto register k, then advancing to state q.
- $?(k, a, q, q')$ checks if the letter a is currently at the top of register k. If so, we advance to state q, and otherwise, we advance to state q'.
- $?(k, \varepsilon, q, q')$ checks if register k is empty. If so, we advance to state q, and otherwise, we advance to state q'.
- $-(k, q, q')$ pops the topmost letter from register k. If the register was already empty, we advance to state q, and otherwise, we advance to state q'.

These semantics are defined formally later. Let $\mathrm{Instr}(\Sigma, Q)$ be the set of (Σ, Q)-instructions. This is in principle an infinite set, but finite if k is bounded.

Definition. A tuple $M = (\Sigma, Q, P)$ is called a *Σ-register machine* if Σ is an alphabet, Q is a finite set of *states* with two distinguished states $q_S \neq q_H$, called the *start state* and *halt state* respectively, and $P : Q \to \mathrm{Instr}(\Sigma, Q)$ is the *program*. If $Q = \{q_0, \dots, q_n\}$, we can describe P as a finite collection of *program lines* $q_i \mapsto P(q_i)$. Since Q is finite, only finitely many registers k are referenced by P; we call the largest such k the *upper register index* of M.

Definition. Let M be a register machine with upper register index n and $\mathbf{w} = (w_0, \dots, w_n) \in \mathbb{W}^{n+1}$. For configurations C, C', we say M *transforms* C into C' if one of the following holds.

- $P(q) = +(k, a, q')$ and $C' = (q', w_0, \dots, w_{k-1}, w_k a, w_{k+1}, \dots, w_m)$.
- $P(q) = \ ?(k, a, q', q'')$, and
  - $w_k = wa$ for some w and $C' = (q', w_0, \dots, w_m)$, or
  - $w_k \neq wa$ for all w and $C' = (q'', w_0, \dots, w_m)$.
- $P(q) = \ ?(k, \varepsilon, q', q'')$, and
  - $w_k = \varepsilon$ and $C' = (q', w_0, \dots, w_m)$, or
  - $w_k \neq \varepsilon$ and $C' = (q'', w_0, \dots, w_m)$.
- $P(q) = -(k, q', q'')$, and
  - $w_k = \varepsilon$ and $C' = (q', w_0, \dots, w_m)$, or
  - $w_k = wa$ and $C' = (q'', w_0, \dots, w_{k-1}, w, w_{k+1}, \dots, w_m)$.

Then we define the *computation sequence* of M with input $\mathbf{w}$ by $C(0, M, \mathbf{w}) = (q_S, \mathbf{w})$, $C(k+1, M, \mathbf{w}) = C'$ where M transforms $C(k, M, \mathbf{w})$ into C'.

Remark. This recursive definition requires that the length of $\mathbf{w}$ is at least $n + 1$, where n is the upper register index. By convention, if $\mathbf{w}$ is too short, we pad it with copies of the empty word ε.

Remark. As defined above, all computation sequences are infinite, because every configuration is transformed by M into some other.

Definition. We say that the computation of M with input $\mathbf{w}$ *halts at time k* or *in k steps* if k is the smallest natural such that $C(k, M, \mathbf{w}) = (q_H, \mathbf{v})$. In this case, we say that $\mathbf{v}$ is the *register content at time of halting*, or the *output* of the computation. If such a k does not exist, we say the computation *does not halt*.

5.2 Strong equivalence

Definition. We say that register machines M, M' are *strongly equivalent* if for all k and $\mathbf{w}$, $C(k, M, \mathbf{w})$ and $C(k, M', \mathbf{w})$ have the same register content, and for all $\mathbf{w}$, we have that M halts after k steps with input $\mathbf{w}$ if and only if M' halts after k steps with input $\mathbf{w}$.

Remark. If $|Q| = |Q'|$, then for every (Σ, Q, P) there exists a strongly equivalent register machine (Σ, Q', P') by relabelling the states in P.

Proposition (the padding lemma)**.** Let M be a register machine. Then there are infinitely many different register machines that are strongly equivalent to M.

Proof. Let $M = (\Sigma, Q, P)$. The register machine completely determines the computation sequence, so after adding a new state $\hat{q}$ to Q, $\hat{q}$ is never a state in any computation sequence. So $(\Sigma, Q \cup \{\hat{q}\}, P \cup \{\hat{p}\})$ is strongly equivalent to M for any program line $\hat{p}$ for $\hat{q}$. $\square$

Proposition. Up to strong equivalence, there are only countably many register machines.

Proof. Only the cardinality of Q matters up to strong equivalence. Let $M_{n,k}$ be the collection of register machines with a fixed state set with $|Q| = n$ and upper register index at most k. By checking cases, we find $|\mathrm{Instr}(\Sigma, Q)| = (k+1)n|\Sigma| + (k+1)n^2|\Sigma| + (k+1)n^2 + (k+1)n^2 = N_{n,k}$, which is finite. Therefore, there are $N_{n,k}^n$ different programs, and hence $|M_{n,k}| = N_{n,k}^n$ is finite. Then the collection of all register machines up to strong equivalence is $\bigcup_{n,k} M_{n,k}$ which is countable. $\square$

5.3 Performing operations and answering questions

Definition. An *operation* is a partial function $f : \mathbb{W}^{n+1} \rightharpoonup \mathbb{W}^{n+1}$. We write $f(\mathbf{w}) \downarrow$ if $\mathbf{w}$ lies in the domain of f, and we say the operation *is defined* or *converges*. We write $f(\mathbf{w}) \uparrow$ otherwise, and say that the operation is *undefined* or *diverges*. A register machine M *performs* an operation f if for all $\mathbf{w}$, $f(\mathbf{w}) \downarrow$ if and only if M halts on input $\mathbf{w}$, and in this case, the register content at time of halting is $f(\mathbf{w})$.

Example. The operation 'never halt' is the empty function, $\operatorname{dom} f = \varnothing$. Then any program that never references the halt state in the right hand side of a program line performs this operation. For example, $q_S \mapsto +(0, a, q_S)$ and $q_H \mapsto +(0, a, q_S)$ suffices.

Remark. There are many register machines that perform the same operation, including many that are not strongly equivalent.

Example. The operation 'halt without doing anything' is the function $f(\mathbf{w}) = \mathbf{w}$ with $\operatorname{dom} f = \mathbb{W}^{n+1}$. An example of a program to perform this is $q_S \mapsto\ ?(0, a, q_H, q_H)$. This halts after one step, and preserves the register content.

Definition. A *question with* $k+1$ *answers* is a partition of $\mathbb{W}^{n+1}$ into $k+1$ sets A_i. A register machine *answers a question* if it has $k+1$ *answer states* $\overline{q}_i$, and upon input of $\mathbf{w}$, after finitely many steps its configuration is $(\overline{q}_i, \mathbf{w})$ for the value of i where $\mathbf{w} \in A_i$.

Example. The question 'is register i empty' is performed by $q_S \mapsto\ ?(i, \varepsilon, \overline{q}_Y, \overline{q}_N)$. The question 'is the final letter in register i the letter a' is performed by $q_S \mapsto\ ?(i, a, \overline{q}_Y, \overline{q}_N)$.

The following lemma allows us to concatenate register machines, or alternatively, to perform subroutines.

Lemma (concatenation). Let M perform $F : \mathbb{W}^{n+1} \rightharpoonup \mathbb{W}^{n+1}$, and M' perform $F' : \mathbb{W}^{n+1} \rightharpoonup \mathbb{W}^{n+1}$. Then we can construct a register machine $\hat{M}$ which performs $F' \circ F$.

Remark. If $F(\mathbf{w}) \uparrow$, then $(F' \circ F)(\mathbf{w}) \uparrow$. If $F(\mathbf{w}) \downarrow$ and $F'(F(\mathbf{w})) \uparrow$, then $(F' \circ F)(\mathbf{w}) \uparrow$. Otherwise, $(F' \circ F)(\mathbf{w}) \downarrow$.

Proof. We may assume without loss of generality that the state sets of the two machines are disjoint. We define $\hat{Q} = Q \cup Q' \setminus \{q_H\}$. We write $P^\star$ for the program P with the rule $q_H \mapsto P(q_H)$ removed, and then all instances of q_H replaced with q'_S. We then define $\hat{P} = P^\star \cup P'$. Then $\hat{M} = (\Sigma, \hat{Q}, \hat{P})$ clearly performs $F' \circ F$. □

Lemma (case distinction). Let Q be a question with $k+1$ answers. Let $F_i : \mathbb{W}^{n+1} \rightharpoonup \mathbb{W}^{n+1}$ be operations for $i \leq k$. Let M be a register machine that answers Q, and let M_i be register machines that perform F_i. Then there is a register machine that performs the operation given by $G(\mathbf{w}) = F_i(\mathbf{w})$ if $\mathbf{w} \in A_i$.

Proof. We assume that Q is disjoint from each Q_i, and $\bigcap_{i \leq k} Q_i = \{q_H\}$. Let $P_i^\star$ be P_i where all occurrences of $q_{S,i}$ are replaced with the ith answer state $\overline{q}_i$. Define $Q^\star = Q \cup \bigcup_{i \leq k} Q_i \setminus \{q_{S,i}\}$ and $P^\star = P \cup \bigcup_{i \leq k} P_i^\star$. Then $M^\star = (\Sigma, Q^\star, P^\star)$ performs G. □

5.4 Register machine API

We can perform many different operations and answer many different questions using register machines. We say that a register is *unused* if no program line references it. A register is *empty* if it contains the empty word. Registers that are used only for computation and not the output are sometimes called *scratch space* or *scratch registers*.

- Consider
$$F(\mathbf{w}) = \begin{cases} \mathbf{w} & w_i \neq \varepsilon \\ \uparrow & w_i = \varepsilon \end{cases}$$
The question 'is register i empty' is performed by a register machine, and in this case, the 'never halt' operation can be performed; in the other case, the 'halt without doing anything' operation can be performed.
- The operation 'delete the final letter of register i if it exists' is performed by the program $q_S \mapsto -(i, q_H, q_H)$.
- The operation 'add letter a to register i' is performed by $q_S \mapsto +(i, a, q_H)$. Note that this machine also performs the operation 'guarantee that the ith register is nonempty'.
- The operation 'delete the content of register i' is performed by $q_S \mapsto -(i, q_H, q_S)$.
- We can perform the operation 'add a fixed word w to register i'. If $w = a_0 \dots a_\ell$, we use the concatenation lemma to perform the operation 'add letter a_j to register i' for each letter in the word.
- The operation 'replace the register content of i with the word w' can be performed by concatenating the operations 'delete the content of register i' and 'add w to register i'.

- We can answer the question 'what is the final letter of register i'. This question has $|\Sigma| + 1$ answers, since the register could be empty. For each letter $a_j \in \Sigma$, we ask the question 'does register i end in letter a_j', and if yes, go to the corresponding answer state $\overline{q}_j$, and if not, go to a state that asks the next question in the sequence. If no question answers 'yes', the register is empty, and we go to an answer state $\overline{q}_\varepsilon$.
- In particular, we can perform the operation 'copy the final letter of register i into register j if it exists', by asking what this letter is, and then in each case, pushing the relevant letter onto register j.
- We can also 'move the final letter of register i into register j if it exists' by first copying the letter and then removing the original from register i.
- The operation 'move the content of register i into register j in reverse order' is accomplished by repeatedly moving a single letter until no more letters lie in register i.
- The operation 'move the content of register i into register j in the correct order' can be performed by considering an unused empty register k. We move the register content from i to k in reverse order and then from k to j in reverse order.
- The operation 'reverse the content of register i' is performed by moving it in reverse order to an unused empty register j, and then moving this into i in the correct order.
- The operation 'move the content of register i into registers j and k in reverse order' is easily performed by copying the final letter of register i into j and then into k, then removing the final letter in register i iteratively until it is empty.
- The operation 'copy the content of register i into register j in reverse order' is accomplished by moving the content of register i into j and an unused empty register k, and then moving the register content of k into i in reverse order.
- The operation 'copy the content of register i into register j in the correct order' is accomplished by copying in the reverse order, and then reversing the content of register j.
- Consider the question 'is the content of register i the word w'. Let $w = a_0 \dots a_k$. We define the subroutine S_ℓ to answer the question 'is a_ℓ the final letter of register i'. If no, move to a state q_N. If yes, move the final letter to an unused empty register k and run subroutine $S_{\ell-1}$, or if $\ell = 0$, move to a state q_Y. At state q_N we move the content of k to i and answer $\overline{q}_N$, and at state q_Y we move the content of k to i and answer $\overline{q}_Y$.

6 Computability theory

6.1 Computable functions and sets

Remark. A lot of computations require the use of scratch space, and we want to reduce the mathematical information related to this scratch space. In the following definition, only register zero is considered real output; all other registers are considered scratch space.

Definition. Let M be a register machine, and let $k \in \mathbb{N}$. Then we define $f_{M,k} : \mathbb{W}^k \rightharpoonup \mathbb{W}$ by $f_{M,k}(\mathbf{w}) \uparrow$ when M does not halt on input $\mathbf{w}$, and $f_{M,k}(\mathbf{w}) = v_0$ when M halts on input $\mathbf{w}$ with halting register content $\mathbf{v}$.

Note that if M, M' are strongly equivalent, $f_{M,k} = f_{M',k}$ for all k. The converse does not hold. For the special case of $k = 1$, we also write $W_M = \operatorname{dom} f_{M,1}$.

Definition. A partial function $f : \mathbb{W}^k \rightharpoonup \mathbb{W}$ is called *computable* if there is a register machine M such that $f = f_{M,k}$.

Remark. There are only countably many computable functions, because there are only countably many register machines up to strong equivalence. For each computable function f, there are infinitely many register machines M such that $f = f_{M,k}$, since any register machine has infinitely many other strongly equivalent register machines. Due to the concatenation lemma and the case distinction lemma, computable functions are closed under concatenation and case distinction.

Example. The identity function on $\mathbb{W}$ is computable. Consider $c : \mathbb{W}^k \to \mathbb{W}$ is given by $c(\mathbf{w}) = v$ for a fixed v. The operation 'replace the content of register 0 with v' is performable on a register machine, so c is computable. The projection $\pi_i : \mathbb{W}^k \to \mathbb{W}$ given by $\pi_i(\mathbf{w}) = w_i$ is computable since the operation 'replace the content of register 0 with register i' can be performed on a register machine by emptying register 0 and then moving the content of register i to register 0.

Definition. Let $X \subseteq \mathbb{W}^k$. We say that a total function $f : \mathbb{W}^k \to \mathbb{W}$ is *a characteristic function of X* if $f(\mathbf{w}) \neq \varepsilon$ if and only if $\mathbf{w} \in X$. Let $a \in \Sigma$. We say that f is *the characteristic function of X* if $f(\mathbf{w}) = a$ if $\mathbf{w} \in X$ and $f(\mathbf{w}) = \varepsilon$ otherwise.

We use the notation χ_X for the characteristic function.

Definition. A set $X \subseteq \mathbb{W}^k$ is *computable* if the characteristic function χ_X of X is computable.

Note that a language is a set of words, so we can now reason about computability of languages.

Definition. Let $X \subseteq \mathbb{W}^k$. A partial function $f : \mathbb{W}^k \rightharpoonup \mathbb{W}$ is called *a pseudocharacteristic function of X* if $\operatorname{dom} f = X$. f is called *the pseudocharacteristic function of X* if $f(\mathbf{w}) = a$ if $\mathbf{w} \in X$, and undefined otherwise.

We use the notation ψ_X for the pseudocharacteristic function.

Definition. A set $X \subseteq \mathbb{W}^k$ is *computably enumerable* if the pseudocharacteristic function ψ_X is computable.

Remark. We will show that every computable set is computably enumerable, but the converse does not hold. We will also show that the computably enumerable sets are exactly the type 0 languages (those languages that have grammars), and that the class of computable languages is properly contained between type 1 and type 0.

6.2 Computability of languages

Proposition. Let $X \subseteq \mathbb{W}^k$. Then:
(i) X is computable if and only if X^c is computable.
(ii) X is computably enumerable if and only if there exists a register machine M such that $X = \operatorname{dom} f_{M,k}$.
(iii) If X is computable, then X is computably enumerable.

Proof. To simplify notation we consider the case $k = 1$. Note that if g and h are computable, then by the case distinction lemma, so is f defined by $f(w) = g(w)$ if $w \neq \varepsilon$, and $f(w) = h(w)$ if $w = \varepsilon$.

For the first part, consider the computable function f_1 given by $g(w) = \varepsilon$ and $h(w) = a$. Then $f_1 \circ \chi_X = \chi_{X^c}$, $f_1 \circ \chi_{X^c} = \chi_X$.

Now consider f_2 given by $g(w) = a$ and $h(w) = \varepsilon$. If $X = \operatorname{dom} f$, then $\psi_X = f_2 \circ f$.

Finally, consider f_3 given by $g(w) = a$ and $h(w) \uparrow$. Then $\psi_X = f_3 \circ \chi_X$. □

Theorem. Every regular language is computable.

Proof. Let L be such a regular language. Let $D = (\Sigma, Q, \delta, q_0, F)$ be a deterministic automaton such that $L = \mathcal{L}(D)$. The first step in our program is to reverse the content of register 0 into register 1, because register machines read words in the opposite order of deterministic automata. For each $q \in Q$, the register machine will have a set of states Q_q that indicate that we are currently mimicking D in state q. We will now move into the state set Q_{q_0}.

When moving into each state set Q_q, our program will read the final letter of register 1. If there are no letters in register 1, go to a fixed accepting state if $q \in F$ and the non-accepting state if $q \notin F$. Otherwise, let b be the last letter in register 1. Remove b from register 1, and go to state set $Q_{\delta(q,b)}$. We implicitly repeat this step, since we have now transitioned into a state set.

If the machine is in the given accepting state, we empty register 0, add a to register 0, and then halt. If the machine is in the non-accepting state, we empty register 0, and then halt. □

6.3 The shortlex ordering

We wish to create an order $<$ on $\mathbb{W}$ such that $(\mathbb{N}, <)$ is order-isomorphic to $(\mathbb{W}, <)$. We first fix an arbitrary total order $<$ on Σ.

Definition. The *shortlex ordering* on $\mathbb{W}$ given by an ordering of Σ is given by $w < v$ when
(i) $|w| < |v|$; or
(ii) $|w| = |v|$ but $w \neq v$, and for the least m such that the mth characters differ, the mth character of w is less than the mth character of v.

This ordering first checks length, then the lexicographic ordering. This is a total ordering on $\mathbb{W}$; it is irreflexive, transitive, and trichotomous. The empty word is the least element.

Example. Let $\Sigma = \{0, 1\}$, and fix $0 < 1$. Then an initial segment of the ordering is

$$\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, \dots$$

We can identify each word with a natural number, given by its index in this sequence, counting from zero. There are 2^k words of length k, so the index of the natural number associated to the word 0^k is exactly $2^k - 1$.

We can naturally extend the operations of addition and multiplication on the set of words by acting on the index of the word in this ordering. For example, $10 + 01 = 010$, because the associated index of 10 is 5, the index of 01 is 4, and the index of 010 is 9. This gives $\mathbb{W}$ the structure of a commutative semiring.

Theorem. The shortlex ordering has the same order type as $\mathbb{N}$. We write $(\mathbb{N}, <) \cong (\mathbb{W}, <)$.

Proof. For a fixed w, the set $\{v \mid v < w\}$ is finite. Therefore, the function $\# : \mathbb{W} \to \mathbb{N}$ given by $\#(w) = |\{v \mid v < w\}|$ is well-defined and is an order isomorphism. □

Theorem. The set $\{(v, w) \mid v < w\}$ is computable. The *successor function* $s : \mathbb{W} \to \mathbb{W}$ with $\#(s(w)) = \#(w) + 1$ is computable.

Proof. The question to determine the ordering of $|w_i|$ and $|w_j|$ can be answered by a register machine by copying i, j into empty registers and repeatedly removing letters until one or both is empty. If they have the same length, we again copy i, j into empty registers in the reverse order, and check and remove each letter until a difference is found.

To compute $s(w)$ for a word w, we find the last letter in w that is not the largest letter in the ordering. Replace this letter with the next letter in the ordering, and replace all subequent letters with the least letter in the ordering. If all k letters are the greatest letter, output the least letter $(k + 1)$-many times. □

6.4 Church's recursive functions

The class of recursive functions is defined inductively.

Definition. The *basic functions* are

$$\begin{aligned} \pi_{k,i} &: \mathbb{W}^k \to \mathbb{W} \\ c_{k,\varepsilon} &: \mathbb{W}^k \to \mathbb{W} \\ s &: \mathbb{W} \to \mathbb{W} \end{aligned}$$

where $\pi_{k,i}(\mathbf{w}) = w_i$, $c_{k,\varepsilon}(\mathbf{w}) = \varepsilon$, and $\#s(w) = \#w + 1$.

We call $\pi_{k,i}$ the *projection* functions, $c_{k,\varepsilon}$ the *constant* functions, and s the *successor* function.

Let $f : \mathbb{W}^m \rightharpoonup \mathbb{W}$ and $g_1, \dots, g_m : \mathbb{W}^k \rightharpoonup \mathbb{W}$. Then their *composition* is the function $h(\mathbf{w}) = f(g_1(\mathbf{w}), \dots, g_m(\mathbf{w}))$.

Let $f\colon \mathbb{W}^k \rightharpoonup \mathbb{W}$ and $g\colon \mathbb{W}^{k+2} \rightharpoonup \mathbb{W}$. Then the partial function $h\colon \mathbb{W}^{k+1} \rightharpoonup \mathbb{W}$ defined by $h(\mathbf{w}, \varepsilon) = f(\mathbf{w})$ and $h(\mathbf{w}, s(v)) = g(\mathbf{w}, v, h(\mathbf{w}, v))$ is a function defined by *recursion*.

Let $f\colon \mathbb{W}^{k+1} \rightharpoonup \mathbb{W}$. Then the function $h\colon \mathbb{W}^k \rightharpoonup \mathbb{W}$ defined by

$$h(\mathbf{w}) = \begin{cases} v & \text{if for all } u \leq v \text{, we have } f(\mathbf{w}, u) \downarrow \text{ and } v \text{ is } <\text{-minimal such that } f(\mathbf{w}, v) = \varepsilon \\ \uparrow & \text{if there is no } v \text{ satisfying the above property} \end{cases}$$

is a function defined by *minimisation*.

Remark. If a class of functions has the basic functions and is closed under composition, it has all constant functions $c_{k,v}(\mathbf{w}) = v$, because if $v = s^k(\varepsilon)$, $c_{k,v} = s^k \circ c_{k,\varepsilon}$.

Definition. A class $\mathcal{C}$ of partial functions is closed under composition, recursion, and minimisation if whenever $f_1, \dots, f_\ell \in \mathcal{C}$, then the results of applying these operations also lie in $\mathcal{C}$.

Remark. The class $\mathcal{P}$ of all partial functions is closed under composition, recursion, and minimisation.

Definition. We call a partial function *recursive* if it lies in the smallest class $\mathcal{C}$ that contains the basic functions and is closed under composition, recursion, and minimisation. A partial function is *primitive recursive* if it lies in the smallest class $\mathcal{C}$ that contains the basic functions and is closed under composition and recursion.

Example. $\pi_{1,0}\colon \mathbb{W}^1 \to \mathbb{W}$ is the identity function, which is primitive recursive. $\pi_{3,2}\colon \mathbb{W}^3 \to \mathbb{W}$ defined by $\pi_{3,2}(u, v, w) = w$ is primitive recursive as it is a basic function. The successor function $s\colon \mathbb{W} \to \mathbb{W}$ is primitive recursive. The function $s \circ \pi_{3,2}$ is primitive recursive, as the composition of primitive recursive functions.

The function h defined by $h(w, \varepsilon) = \pi_{1,0}(w)$ and $h(w, s(v)) = s \circ \pi_{3,2}(w, v, h(w, v)) = s(h(w, v))$ is primitive recursive, which is exactly the addition function $\#h(n, m) = \#n + \#m$. We can define multiplication and exponentiation in a similar way, and so all of these are primitive recursive.

We can encode recursive functions in trees. Let T be a finitely branching tree, and define a labelling ℓ on T with the labels

	label	arity	branching number
projection	$B^\pi_{k,i}$	k	0
constant	$B^c_{k,i}$	k	0
successor	B^s	1	0
composition	$C_{n,k}$	k	$n+1$
recursion	R_k	$k+1$	2
minimisation	M_k	k	1

Definition. A tree T with a labelling ℓ is called a *recursion tree* if the branching of the tree corresponds exactly to the branching numbers of its labels, and

(i) if $\ell(s) = C_{n,k}$, then the first successor of s has a label of arity n and all other have labels with arity k;
(ii) if $\ell(s) = R_k$, then the first successor of s has arity k and the other has arity $k+2$;

(iii) if $\ell(s) = M_k$, then the successor has arity k.

A recursion tree is *primitive* if it has no minimisation labels M_k.

The following recursion tree describes the addition function defined above.

$$\begin{tikzcd}
 & R_1 \arrow[dl, no head] \arrow[dr, no head] & & \\
B^{\pi}_{1,0} & & C_{1,3} \arrow[dl, no head] \arrow[dr, no head] & \\
 & B^s & & B^{\pi}_{3,2}
\end{tikzcd}$$

We can assign a (partial) recursive function $f_{T,\ell}$ to every recursive tree (T, ℓ). If the tree is primitive, the function obtained is primitive recursive.

Theorem. A partial function f is recursive if and only if there is a recursion tree (T, ℓ) such that $f = f_{T,\ell}$. It is primitive recursive if it admits a recursion tree that is primitive.

Proof. We can obtain the associated partial function from a recursion tree by induction on the height on the tree. For the converse, it suffices to show that the class of functions $f_{T,\ell}$ contains the basic functions and is closed under composition, recursion, and minimisation, which holds by construction. $\square$

Theorem. Every partial recursive function is computable.

Proof. The basic functions have already been shown to be computable. Computable functions are closed under composition (previously called concatenation). So it suffices to show that the computable functions are closed under recursion and minimisation.

Let f, g be computable functions; we want to show that h defined by $h(\mathbf{w}, \varepsilon) = f(\mathbf{w})$ and $h(\mathbf{w}, s(v)) = g(\mathbf{w}, v, h(\mathbf{w}, v))$ is computable. We describe a register machine.

(i) Let k, ℓ be two empty unused registers.

(ii) Compute $f(\mathbf{w})$, and write the result to register ℓ. Note that if $f(\mathbf{w})$ is undefined, this produces the desired result.

(iii) If $v = \varepsilon$, output the content of register ℓ. Otherwise, apply the successor function s to register k and perform the following subroutine.

   (a) Compute $g(\mathbf{w}, v, u)$ where u is the content of register ℓ, then overwrite register ℓ with the result.

   (b) Check whether v is equal to the register content of k. If so, output register ℓ. Otherwise, apply s to register k and restart the subroutine.

We now consider minimisation. Let f be computable. Let k be empty and unused. Perform the following subroutine.

(i) Compute $f(\mathbf{w}, u)$ where u is the content of register k. If this result is undefined, this is the desired result.

(ii) Check whether the computation result is empty. If it is empty, output the register content of k. Otherwise, apply the successor function s to k then restart the subroutine.

$\square$

Remark. The proof showed that the computable functions are closed under recursion and minimisation, not just that all partial recursive functions are computable. Therefore, we can use recursion and minimisation directly to construct computable functions or register machines.

6.5 Merging and splitting words

There is a bijection $z : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, called the *Cantor zigzag function*.

$$z(i, j) = \frac{(i + j)(i + j + 1)}{2} + j$$

This gives a bijection $\mathbb{W} \times \mathbb{W} \to \mathbb{W}$. All of these operations are computable by register machines.

Definition. Let v, w be words. Then we can *merge* the two words into $v * w$, which is the unique word such that $\#(v * w) = z(\#v, \#w)$. We can *split* a word w into u, v such that $\#w = z(\#u, \#v)$. We write $u = w_{(0)}$ and $v = w_{(1)}$.

Technically, splitting a word is not a computable function, since computable functions are defined to always have codomain $\mathbb{W}$. However, the operation of splitting a word can be performed.

6.6 Universality

Consider an alphabet Σ. We then have a notion of computability for sets $X \subseteq \Sigma^\star = \mathbb{W}$. If $\Sigma \subseteq \Sigma'$, then every Σ-register machine is a Σ'-register machine. However, the notion of Σ'-computability is no stronger than Σ-computability. One can show that computability over any alphabet Σ with $|\Sigma| \geq 2$ is equivalent to computability over the set $\{0, 1\}$ by encoding each letter as a binary string.

In this subsection, we aim to show that there is a *universal* register machine, which is a machine that can mimic every register machine. Let Σ be an alphabet, and add additional symbols

$$\mathbf{0}\ \mathbf{1}\ +\ -\ ?\ (\ )\ ,\ \mapsto\ \square$$

We name the new alphabet Σ'. When we encode a mathematical object o as a word $\Sigma'^\star$, we write the encoded result $\mathrm{code}(o)$.

- We can encode $\mathbb{N}$ in binary using **0** and **1**, for instance, $\mathrm{code}(19) = \mathbf{10011}$.
- If $Q = \{q_0, \dots, q_k\}$, we define $\mathrm{code}(q_k) = \mathrm{code}(k)$.
- We encode instructions $I \in \mathrm{Instr}(\Sigma, Q)$ using $+\ -\ ?\ ,\ (\ )$; for instance, $\mathrm{code}(+(k, a, \ell)) = +(\mathrm{code}(k), a, \mathrm{code}(\ell))$.
- We encode program lines by $\mathrm{code}(q \mapsto I) = \mathrm{code}(q) \mapsto \mathrm{code}(I)$.
- We encode a register machine with program P as $\mathrm{code}(q_0 \mapsto P(q_0)), \dots, \mathrm{code}(q_n \mapsto P(q_n))$.

- We encode sequences of words by $\mathbf{w}$ by $\text{code}(\mathbf{w}) = \square w_0 \square \dots \square w_k \square$.
- We encode configurations $(q, \mathbf{w})$ by $\text{code}(q)\,\text{code}(\mathbf{w})$.

Lemma. The function h defined by

$$h(w, u, v) = \begin{cases} \text{code}(C(M, \mathbf{w}, \#v)) & \text{if } \exists M, \mathbf{w} \text{ such that } w = \text{code}(M), u = \text{code}(\mathbf{w}) \\ \uparrow & \text{otherwise} \end{cases}$$

is computable.

Proof. Define by recursion

$$h(\text{code}(M), \text{code}(\mathbf{w}), \varepsilon) = \text{code}(q_0)\,\text{code}(\mathbf{w}); \quad h(\text{code}(M), \text{code}(\mathbf{w}), s(v)) = \text{code}(C')$$

where C' is the result of transforming $h(\text{code}(M), \text{code}(\mathbf{w}), v)$ by the machine M. □

Corollary. The *truncated computation* function $t_{M,k}$ defined by

$$t_{M,k}(\mathbf{w}, v) = \begin{cases} a & M \text{ has halted before time } \#v \text{ on input } \mathbf{w} \\ \varepsilon & \text{otherwise} \end{cases}$$

is computable.

Proof. Using recursion on the function h from the previous lemma, we check all values of h for words u such that $\#u < \#v$. If any of the values is in state q_H, output a, otherwise, output ε. □

Theorem (the software principle). The function g defined by

$$g(v, u) = \begin{cases} f_{M,k}(\mathbf{w}) & \text{if } v = \text{code}(M), u = \text{code}(\mathbf{w}) \text{ and } \mathbf{w} \text{ has length } k \\ \uparrow & \text{otherwise} \end{cases}$$

is computable.

Proof. We have a computable function f that maps w, u, v to $\text{code}(C(M, \mathbf{w}, \#v))$ if $\text{code}(M) = w$ and $\text{code}(\mathbf{w}) = u$ by the previous lemma. We start by checking whether w is a code for a register machine and u is a code for a k-tuple of words; if not, never halt. Write f' for the computable function mapping w, u, v to a if the state of $f(w, u, v)$ is q_H, and ε otherwise. We minimise f' to obtain the computable function h, such that $h(w, u)$ is the least v such that $f(w, u, v)$ is in state q_H if it exists. If $h(w, u)$ does not halt, then there is no step at which the computation halts, as expected, since $g(w, u)$ should not halt in this case. If $h(w, u)$ halts, consider the configuration $C(M, \mathbf{w}, \#h(w, u))$ and find the code for its 0th register, and write this into the actual 0th register. □

Remark. A register machine U that computes g is called a *universal register machine*. U has a finite amount of used registers and states, but can mimic the behaviour of any register machine using an arbitrarily large amount of registers and states.

This allows us to streamline notation; for a word $v \in \mathbb{W}$, we can write

$$f_{v,k}(\mathbf{w}) = f_{U,2}(v, \operatorname{code}(\mathbf{w})) = f_{M,k}(\mathbf{w})$$

if $\operatorname{code}(M) = v$. Similarly, we can write $W_v = \operatorname{dom} f_{v,1}$, so $\{W_v \mid v \in \mathbb{W}\}$ is the set of computably enumerable sets.

Theorem (*s–m–n* theorem; parameter theorem)**.** Let $g : \mathbb{W}^{k+1} \rightharpoonup \mathbb{W}$ be computable. Then there exists a total computable function $h : \mathbb{W} \to \mathbb{W}$ such that $f_{h(v),k}(\mathbf{w}) = g(\mathbf{w}, v)$.

This process is called *currying*, after Haskell Curry.

Remark. $g_v(\mathbf{w}) = g(\mathbf{w}, v)$ is a function in k variables. This is computable, so there is a mathematical function h such that $g_v = f_{h(x),k}$, but this h is not *a priori* computable.

Proof. First, the operation $\mathbf{w} \mapsto (\mathbf{w}, v)$ is performed by a register machine M_v; this is the register machine that writes v into register k. Therefore, we have a computable function $v \mapsto \operatorname{code}(M_v)$. Now, since g is computable, there is a register machine M such that $f_{M,k+1} = g$. Therefore, g_v is computed by the sequence of register machines M_v then M. We can computably concatenate two register machines, so we can compute a code for $M \circ M_v$. Hence the function $h(v) = \operatorname{code}(M \circ M_v)$ is total and computable.

We must show that $f_{h(v),k}(\mathbf{w}) = g(\mathbf{w}, v)$. Indeed,

$$f_{h(v),k}(\mathbf{w}) = f_{\operatorname{code}(M \circ M_v),k}(\mathbf{w}) = f_{M \circ M_v,k}(\mathbf{w}) = g_v(\mathbf{w}) = g(\mathbf{w}, v)$$

as required. □

6.7 The halting problem

Consider the sets

$$\mathbb{K}_0 = \{(w, v) \mid f_{w,1}(v)\downarrow\}; \quad \mathbb{K} = \{w \mid f_{w,1}(w)\downarrow\}$$

Theorem. $\mathbb{K}_0$ and $\mathbb{K}$ are computably enumerable.

Proof. It suffices to show that $\mathbb{K}_0, \mathbb{K}$ are the domains of computable functions. By the software principle, $f_{U,2}(w, v) = f_{w,1}(v)$ and $\operatorname{dom} f_{U,2} = \mathbb{K}_0$ as required. Observe that the diagonal function $\Delta(w) = (w, w)$ is computable, so $f_{U,2} \circ \Delta$ is computable, and $\operatorname{dom}(f_{U,2} \circ \Delta) = \mathbb{K}$. □

Theorem (the halting problem)**.** Neither $\mathbb{K}_0$ nor $\mathbb{K}$ are computable.

Proof. We prove the result for $\mathbb{K}_0$. Suppose that $\mathbb{K}_0$ is computable, so the characteristic function $\chi_{\mathbb{K}_0}$ is computable. Now, define

$$f(w) = \begin{cases} \uparrow & \text{if } \chi_{\mathbb{K}_0}(w, w) = a \\ \varepsilon & \text{if } \chi_{\mathbb{K}_0}(w, w) = \varepsilon \end{cases}$$

This is a computable function, so there is a machine $d \in \mathbb{W}$ such that $f_{d,1} = f$. Now,

$$f(d)\downarrow \iff f_{d,1}(d)\downarrow \iff (d, d) \in \mathbb{K}_0 \iff \chi_{\mathbb{K}_0}(d, d) = a \iff f(d)\uparrow$$

The proof is almost exactly the same for $\mathbb{K}$. □

6.8 Sets with quantifiers

Definition. $X \subseteq \mathbb{W}^k$ is called Σ_1 if there is a computable set $Y \subseteq \mathbb{W}^{k+1}$ such that $\mathbf{w} \in X \iff \exists y, (\mathbf{w}, y) \in Y$. We say $X = p(Y) = \{\mathbf{w} \mid \exists y,\, (\mathbf{w}, y) \in Y\}$ is the *projection* of Y. We say X is Π_1 if it is the complement of a Σ_1 set. We say X is Δ_1 if it is Σ_1 and it is Π_1.

Remark. The notation Σ is chosen to symbolise an existential quantifier, and Π symbolises the universal quantifier. In logic, sums and existentials are related, and products and universal quantifiers are also related. Δ is chosen for the German word *Durchschnitt* ('intersection'), as Δ_1 is the intersection of Σ_1 and Π_1.

Proposition. Every computable set is Δ_1.

Proof. By closure under complement, it suffices to show every computable set is Σ_1. The computable set $Y = \{(\mathbf{w}, y) \mid \mathbf{w} \in X\}$ has projection X. Logically, this adds a trivial existential quantification. ☐

Theorem. The computably enumerable sets are exactly the Σ_1 sets.

Proof. Suppose X is computably enumerable. Then by definition, the pseudocharacteristic function ψ_X is computable. Then there exists a register machine M such that $\psi_X = f_{M,k}$. We define $Y = \{(\mathbf{w}, y) \mid t_{M,k}(\mathbf{w}, y) = a\}$ where $t_{M,k}$ is the truncated computation function for the register machine M. Y is computable, since $t_{M,k} = \chi_Y$. Then $\mathbf{w} \in X \iff \psi_X(\mathbf{w}) \downarrow \iff \exists y,\, (\mathbf{w}, y) \in Y$ as required.

Now suppose X is Σ_1. Let Y be a computable set such that $X = p(Y)$. As the computable sets are closed under complement, the characteristic function χ_{Y^c} is computable. We apply minimisation to χ_{Y^c} to obtain a function h such that $h(\mathbf{w})$ is the minimal y such that $(\mathbf{w}, y) \in Y$. Then $\operatorname{dom} h = p(Y) = X$, so X is the domain of a partial computable function as required. ☐

Example. Let $f : \mathbb{W}^2 \rightharpoonup \mathbb{W}$ be a partial computable function in two variables. Then $X = \{w \mid \exists v,\, f(w, v) \downarrow\}$ is computably enumerable. Note that $f(w, v) \downarrow$ is not a computable predicate. Let M be a register machine such that $f = f_{M,2}$, and let

$$Z = \{(w, v_0, v_1) \mid t_{M,2}(w, v_0, v_1) = a\}$$

Clearly Z is computable. Define

$$Y = \{(w, u) \mid (w, u_{(0)}, u_{(1)}) \in Z\}$$

This is also computable. Now,

$$\begin{aligned}
\exists v,\, f(w, v) \downarrow &\iff \exists v_0,\, \exists v_1,\, (w, v_0, v_1) \in Z \\
&\iff \exists u,\, (w, u_{(0)}, u_{(1)}) \in Z \\
&\iff (w, u) \in Y \\
&\iff w \in p(Y)
\end{aligned}$$

So X is Σ_1 as required.

Remark. The previous argument is sometimes known as a *zigzag argument*; a pair of existential quantifiers can be merged into a single existential by merging the two words. Hence, we can perform infinitely many computations in parallel.

Corollary. The computable sets are exactly the Δ_1 sets.

Proof. If X is computable, it must be Δ_1 by a previous result. If X is Δ_1, we can use a zigzag technique. We know that there are machines M, M' such that $w \in X \iff \exists v,\ t_{M,k}(\mathbf{w}, v) = a$ and $w \notin X \iff \exists v,\ t_{M',k}(\mathbf{w}, v) = a$. Now, consider

$$f(\mathbf{w}, v) = \begin{cases} t_{M,k}(\mathbf{w}, v_{(1)}) & \#v_{(0)} \text{ is even} \\ t_{M',k}(\mathbf{w}, v_{(1)}) & \#v_{(0)} \text{ is odd} \end{cases}$$

This is computable. Apply minimisation to f to obtain a function h where $h(\mathbf{w})$ is the least v such that $f(\mathbf{w}, v) \neq \varepsilon$. We output a if $\#h(\mathbf{w})_{(0)}$ is even, and ε if $\#h(\mathbf{w})_{(0)}$ is odd. □

Corollary. Σ_1 is not closed under complement.

Proof. The complement of the halting set $\mathbb{W} \setminus \mathbb{K}$ is Π_1 and not Δ_1, so not Σ_1. □

Theorem. Every type 0 language is computably enumerable.

Proof. Let $G = (\Sigma, V, P, S)$ and let $\Sigma' = \Omega \cup \{\rightarrow\}$. We encode derivations as $\sigma_0 \rightarrow \cdots \rightarrow \sigma_n$; this is a Σ'-word. We say $w \in (\Sigma')^\star$ is a *derivation code* if w is of this form with $(\sigma_0, \dots, \sigma_n)$ a G-derivation. In this case, we call σ_0 the *initial string* and σ_n the *final string*. Let

$$Y = \{(w, v) \mid v \text{ is a derivation code with initial string } S \text{ and final string } w\}$$

Y is computable since we can produce a register machine that tests if a given derivation code can be produced from a fixed given grammar. But $w \in \mathcal{L}(G) \iff \exists v,\ (w, v) \in Y$. This is Σ_1, as required. □

Remark. The converse also holds; every computably enumerable set $X \subseteq \mathbb{W}$ is a type 0 language. This will not be proven rigorously in this course; a sketch will be provided later.

6.9 Closure properties

Proposition. The computable sets are closed under intersection, union, complement, difference, and concatenation.

Proof. Let A, B be computable sets, so χ_A, χ_B are computable functions. We obtain

$$\chi_{A\cap B}(\mathbf{w}) = \begin{cases} a & \chi_A(\mathbf{w}) = a \text{ and } \chi_B(\mathbf{w}) = a \\ \varepsilon & \text{otherwise} \end{cases}$$

For complement,

$$\chi_{\mathbb{W}\setminus A}(\mathbf{w}) = \begin{cases} a & \chi_A(\mathbf{w}) = \varepsilon \\ \varepsilon & \text{otherwise} \end{cases}$$

For concatenation, we suppose $A, B \subseteq \mathbb{W}$ are one-dimensional. Given a word w, we can iterate over all possible decompositions $w = vu$ and check if $v \in A, u \in B$. There are $(|w| + 1)$-many such decompositions, so this minimisation will always halt. □

Remark. The result for intersection is analogous to the product construction from deterministic automata; two computable functions can be evaluated in parallel since they always terminate, and then their results may be combined.

Proposition. The computably enumerable sets are closed under intersection, union, and concatenation. They are not closed under complement or difference.

Proof. We have already shown that the complement of the halting set $\mathbb{K}$ is Π_1 but not Σ_1, so the computably enumerable sets are not closed under complement or difference. For intersection, the same construction as before works.

$$\chi_{A\cap B}(\mathbf{w}) = \begin{cases} a & \psi_A(\mathbf{w}) = a \text{ and } \psi_B(\mathbf{w}) = a \\ \uparrow & \text{otherwise} \end{cases}$$

This is because if ψ_A or ψ_B diverge, the result is $\uparrow$ as desired. For union, we cannot compute ψ_A and ψ_B serially, since if $\psi_A \uparrow$ we never run ψ_B at all. Using the zigzag technique, we can check $\psi_A(\mathbf{w})$ and $\psi_B(\mathbf{w})$ in parallel, halting if either halts at any time index. This idea is elaborated on an example sheet.

For concatenation, consider the set Z of triples (w, v, u) such that v is an initial segment of w, and after $\#u$ steps, $\psi_A(v) = a$ and $\psi_B(v') = a$, where $w = vv'$. Now define $Y = \{(w, u) \mid (w, u_{(0)}, u_{(1)}) \in Z\}$, so $w \in AB$ if and only if there exists v such that $(w, v) \in Y$. □

Proposition. X is computably enumerable if and only if there is a partial computable function f such that $X = \operatorname{Im} f$.

Remark. In fact, a stronger result is true: X is computably enumerable if and only if there is a *total* computable function f such that $X = \operatorname{Im} f$. This is seen on an example sheet. This result justifies the name 'computably enumerable'.

Proof. If ψ_X is computable, then so is

$$f(w) = \begin{cases} w & \psi_X(w) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

Clearly $\operatorname{Im} f = X$ as required.

Conversely, suppose $f : \mathbb{W} \rightharpoonup \mathbb{W}$ with $X = \operatorname{Im} f$. Suppose $f = f_{c,1}$. We use the zigzag technique. Define the set Z of tuples (w, v, u) such that $t_{c,1}(v, u) = a$ and $f_{c,1}(v) = w$. Let $Y = \{(w, v) \mid (w, v_{(0)}, v_{(1)}) \in Z\}$, so $\operatorname{Im} f = p(Y)$. □

6.10 The Church–Turing thesis

Register machines and recursive functions can both be used to define computability. Historically, *Turing machines* were also used to define and analyse computability. There is another alternative, known as *while programs*. Notably, in this model, there is no special 'halt state'; the program halts simply when there are no more instructions to execute. Therefore the computation sequence in this model may be finite. This gives rise to a notion of while computable functions, the functions computed by a while program.

Theorem. Let $f : \mathbb{W}^k \rightharpoonup \mathbb{W}$. Then, the following are equivalent.
(i) f is (register machine) computable.
(ii) f is partial recursive.
(iii) f is Turing computable.
(iv) f is while computable.

Turing machines, register machines, recursive functions, and while programs are superficially completely different approaches, yet the classes of computable functions that they define are exactly identical. The *Church–Turing thesis* is that this is universal; any reasonable notion of computation is equivalent. Unfortunately, this is a nonmathematical statement, and cannot be made precise; this is simply a statement that describes our intuition about what computation means. Accepting this thesis allows us to freely choose which notion of computability we would like to use for a given task.

The following is a proof sketch of the fact that computably enumerable sets are type 0 languages. The sketch makes use of the fact that Turing computability is exactly register machine computability. For more detail, see *Formal Languages* (Salomaa 1973).

Proof sketch. Let M be a Turing machine computing ψ_X. Without loss of generality, let the read-write head be then moved to the front, so $q_s\square w\square \xrightarrow{M} q_H\square a\square$. This is a rewrite system with the rules described by the definition of the Turing machine, transforming $q_S\square w\square$ into $q_H\square a\square$

We define a grammar which starts from S, with $S \to q_H\square a\square$, and performs all Turing instructions backwards. When q_S is seen, it deletes everything except w. □

6.11 Solvability of decision problems

We can use the Church–Turing thesis to give precise statements of our decision problems, without relying on an informal notion of 'algorithm'. First, we encode grammars in such a way that for all $w \in \mathbb{W}$, there exists a grammar G such that code$(G) = w$; we write G_w for the associated grammar for a word. We require that all grammars are of the form G_w for some word $w \in \mathbb{W}$. Now,

(i) the word problem is $\{(w, v) \mid w \in \mathcal{L}(G_v)\}$;

(ii) the emptiness problem is $\{w \mid \mathcal{L}(G_w) = \varnothing\}$;

(iii) the equivalence problem is $\{(w, v) \mid \mathcal{L}(G_w) = \mathcal{L}(G_v)\}$.

These are sets of tuples of words, so we can use our notion of computability. We can now concretely define that such a problem is *solvable* if the set is computable.

Theorem. The word problem for type 0 grammars is unsolvable.

Proof. Let $W = \{(w, v) \mid w \in \mathcal{L}(G_v)\}$. We want to show that W is not computable. Recall that $\mathbb{K}_0 = \left\{(w, v) \mid f_{w,1}(v) \downarrow\right\}$; we will use a proof analogous to the one used for this set. Suppose W is computable, so let

$$f(w) = \begin{cases} \uparrow & w \in \mathcal{L}(G_w) \\ a & w \notin \mathcal{L}(G_w) \end{cases}$$

Then f is a computable function. Hence, $\operatorname{dom} f$ is computably enumerable. So there exists a grammar G such that $\mathcal{L}(G) = \operatorname{dom} f$. Let $d \in \mathbb{W}$ be such that $G = G_d$. Then

$$d \in \mathcal{L}(G_d) \iff d \in \operatorname{dom} f \iff d \notin \mathcal{L}(G_d)$$

□

6.12 Reduction functions

Definition. Let $A, B \subseteq \mathbb{W}$. A function $f : \mathbb{W} \to \mathbb{W}$ is called a *reduction* from A to B if f is total computable and $w \in A$ if and only if $f(w) \in B$. We write $A \leq_m B$ if there is a reduction from A to B.

Remark. Given a reduction f from A to B, the set A is intuitively 'at most as complicated as B'. Note that $f^{-1}(B) = A$.

The subscript m in the notation $A \leq_m B$ stands for 'many-one'; the function f need not be injective. Note that $\leq_m$ is reflexive and transitive. This relation respects complements: $A \leq_m B$ implies $\mathbb{W} \setminus A \leq_m \mathbb{W} \setminus B$. The relation is not in general antisymmetric, so this does not form a partial order. Instead, $\leq_m$ forms a (partial) preorder.

If $\leq$ is a preorder on a set X, we can define the equivalence relation $x \sim y$ when $x \leq y$ and $y \leq x$. Then $\left(X/\sim, \leq\right)$ is a partial order. A preorder can therefore be understood as a partial order, except that instead of ordering single elements, it orders clusters of equivalent elements.

If $A \leq_m B$ and B is computable, then A is also computable. Similarly, if $A \leq_m B$ and B is computably enumerable, then A is also computably enumerable. This demonstrates the fact that $\chi_A = \chi_B \circ f$ and $\psi_A = \psi_B \circ f$, where f is the reduction.

Note that if $A \leq_m B$ and A is not computable, then B is also not computable, and a similar result holds for sets that are not computably enumerable. In particular, if $\mathbb{K} \leq_m A$, then A is not computable. If $\mathbb{W} \setminus \mathbb{K} \leq_m A$, then A is not computably enumerable.

Remark. Many of the previous proofs in this section have implicitly used the notion of a reduction function, for instance, the claim that solvability of the set $\{(w, v) \mid w \in \mathcal{L}(G_v)\}$ is equivalent to solvability of the set $\{(w, v) \mid w \in W_v\}$.

Proposition. Let A be a computable set, and $B \neq \varnothing, \mathbb{W}$. Then $A \leq_m B$.

Proof. Since $B \neq \varnothing, \mathbb{W}$, let $v \in B, u \notin B$. Since A is computable, we have the computable function

$$f(w) = \begin{cases} v & w \in A \\ u & w \notin A \end{cases}$$

This is a reduction from A to B as required. □

Note that $\mathbb{W} \setminus \mathbb{K} \not\leq_m \mathbb{K}$, otherwise $\mathbb{K}$ is not computably enumerable. We also have $\mathbb{K} \not\leq_m \mathbb{W} \setminus \mathbb{K}$ from the first result, after considering complements. There are therefore various different *degrees of unsolvability*: equivalence classes of $\leq_m$ that are strictly larger than the class of computable sets.

There are many more such classes than the ones containing $\mathbb{K}$ and $\mathbb{W} \setminus \mathbb{K}$. Let $\{0, 1\} \subseteq \Sigma$. If A, B are sets, we can define the *Turing join* $A \oplus B = 0A \cup 1B$. Then $A \leq_m A \oplus B$ and $B \leq_m A \oplus B$. The Turing join produces an upper bound in the set of equivalence classes of sets of words, and it can be shown that this is the least upper bound. Hence we obtain another class of sets represented by $\mathbb{K} \oplus \mathbb{W} \setminus \mathbb{K}$. This is neither Σ_1 nor Π_1.

Definition. If $\mathcal{C}$ is a class of sets, we say that A is *$\mathcal{C}$-hard* if for all $B \in \mathcal{C}$, we have $B \leq_m A$. We say that A is *$\mathcal{C}$-complete* if it is $\mathcal{C}$-hard and $A \in \mathcal{C}$.

Remark. A $\mathcal{C}$-hard set is 'at least as hard as $\mathcal{C}$'. A $\mathcal{C}$-complete set is the 'most complicated' $\mathcal{C}$ set.

Corollary. Let A be Δ_1 and $A \neq \varnothing, \mathbb{W}$. Then A is Δ_1-complete.

Proof. The Δ_1 sets are the computable sets, so we simply apply the previous proposition. □

Theorem. $\mathbb{K}$ is Σ_1-complete.

Proof. Clearly $\mathbb{K} \in \Sigma_1$. Now, let X be an arbitrary set in Σ_1, so X is computably enumerable. Let f be a partial computable function such that $X = \operatorname{dom} f$. It suffices to show $X \leq_m \mathbb{K}$.

Consider the function $g(w, u) = f(w)$. This is computable. We can therefore apply the *s–m–n* theorem to obtain a total computable function h such that $f_{h(w)}(u) = g(w, u) = f(w)$. We claim that h is a reduction function from X to $\mathbb{K}$.

Suppose $w \in X$. Then $w \in \operatorname{dom} f$, so $f_{h(w)}$ is the constant function $f(w)$. Hence $W_{h(w)} = \mathbb{W}$. So f is total, and therefore $f_{h(w)}(h(w)) \downarrow$. So $h(w) \in \mathbb{K}$.

Now suppose $w \notin X$, so $w \notin \operatorname{dom} f$. Then $f_{h(w)}$ does not halt for any input, giving $W_{h(w)} = \varnothing$. So $f_{h(w)}(h(w)) \uparrow$ and in particular $h(w) \notin \mathbb{K}$. □

6.13 Rice's theorem

We say that M and M' are weakly equivalent when $\operatorname{dom} f_{M,1} = W_M = W_{M'} = \operatorname{dom} f_{M',1}$. We can extend this to words. Words v, u are weakly equivalent when $W_v = W_u$, and write $v \sim u$.

Definition. A set $I \subseteq \mathbb{W}$ is called an *index set* if it is closed under weak equivalence.

Remark. Index sets are unions of equivalence classes.

Example. $\varnothing$ and $\mathbb{W}$ are the trivial index sets. Other index sets correspond to properties of computably enumerable sets. $\mathbf{Emp} = \{v \mid W_v = \varnothing\}$, $\mathbf{Fin} = \{v \mid W_v \text{ finite}\}$, $\mathbf{Inf} = \{v \mid W_v \text{ infinite}\}$, $\mathbf{Tot} = \{v \mid W_v = \mathbb{W}\}$ are index sets. Note that the emptiness problem is precisely the index set **Emp**.

Theorem (Rice's theorem)**.** No nontrivial index set is computable.

Fix $w \in \mathbb{W}$ and consider the function

$$g(u,v) = \begin{cases} f_{w,1}(v) & f_u(u)\downarrow \text{ or equivalently, } u \in \mathbb{K} \\ \uparrow & \text{otherwise} \end{cases}$$

This is computable, even though the case distinction itself is not computable. By the *s*–*m*–*n* theorem, there is a total function h such that

$$f_{h(u)}(v) = g(u,v) = \begin{cases} f_{w,1}(v) & u \in \mathbb{K} \\ \uparrow & u \notin \mathbb{K} \end{cases}$$

If $u \in \mathbb{K}$, then $W_{h(u)} = W_w$. If $u \notin \mathbb{K}$, then $W_{h(u)} = \varnothing$. This h will be used as a reduction function.

Proof. Let I be an index set. Let e be such that $W_e = \varnothing$. Then either $e \in I$, or $e \notin I$.

Suppose $e \in I$. Since I is nontrivial, there exists $w \notin I$, so $W_w \neq \varnothing$. Consider the function g from the discussion above, instantiated with this choice of w, and apply the *s*–*m*–*n* theorem to obtain a total function h. We claim that h reduces $\mathbb{W} \setminus \mathbb{K}$ to I. If $u \in \mathbb{K}$, then $W_{h(u)} = W_w$. Hence $h(u) \sim w$, so $h(u) \notin I$. If $u \notin \mathbb{K}$, then $W_{h(u)} = \varnothing$, so $h(u) \sim e$, so $h(u) \in I$.

Now suppose $e \notin I$. Then there exists $w \in I$, and $W_w \neq \varnothing$. Take g and h as before. We claim now that h reduces $\mathbb{K}$ to I. If $u \in \mathbb{K}$, then $W_{h(u)} = W_w$, so $h(u) \sim w$, so $h(u) \in I$. If $u \notin \mathbb{K}$, then $W_{h(u)} = \varnothing$, so $h(u) \sim e$, giving $h(u) \notin I$. □

Remark. The proof given for Rice's theorem shows a stronger statement: if $e \in I$ then $\mathbb{W} \setminus \mathbb{K} \leq_m I$, and if $e \notin I$ then $\mathbb{K} \leq_m I$. This allows us to show that certain index sets are not computably enumerable. $e \in$ **Emp** so $\mathbb{W} \setminus \mathbb{K} \leq_m$ **Emp**. Similarly, $\mathbb{W} \setminus \mathbb{K} \leq_m$ **Fin**. For the other two index sets, we can only deduce that $\mathbb{K} \leq_m$ **Inf** and $\mathbb{K} \leq_m$ **Tot**, since e does not lie in these sets.

Corollary. Emp, **Fin**, **Inf**, **Tot** are not computable.

Corollary. The emptiness problem for type 0 grammars is unsolvable.

Corollary. The equivalence problem for type 0 grammars is unsolvable.

Proof. We define **Eq** $= \{(w,v) \mid W_w = W_v\}$. The function $g(w) = (w,e)$ can be performed by a register machine for any e. If $W_e = \varnothing$, then $\chi_{\mathbf{Emp}} = \chi_{\mathbf{Eq}} \circ g$. Hence, if **Eq** is computable, so is **Emp**. □

Remark. One can show that **Emp** is many-one equivalent to $\mathbb{W} \setminus \mathbb{K}$, so it is Π_1-complete, as proven on the last example sheet. The other problems **Tot**, **Inf**, **Fin** are not in Σ_1 or Π_1.

Theorem. Fin is not Σ_1 or Π_1.

Proof. We know $\mathbb{W} \setminus \mathbb{K} \leq_m$ **Fin** by the proof of Rice's theorem, so **Fin** is not Σ_1. To show it is not Π_1, one must show that $\mathbb{K} \leq_m$ **Fin**. Consider

$$g(w, v) = \begin{cases} \uparrow & t_{w,1}(w, v) = a \\ \varepsilon & \text{otherwise} \end{cases}$$

Applying the *s*–*m*–*n* theorem, we obtain a total function h such that $f_{h(w),1}(v) = g(w, v)$. We show h reduces $\mathbb{K}$ to **Fin**. If $w \in \mathbb{K}$, then $f_{h(w),1}$ is undefined from v onwards, where v is the halting time of $f_w(w)$. Hence, $W_{h(w)}$ is finite, so $h(w) \in$ **Fin**. If $w \notin \mathbb{K}$, then $g(w, v) = \varepsilon$ for all v. So $f_{h(w),1}$ is the constant function with value ε. Hence $W_{h(w)} = \mathbb{W}$ which is infinite, so $h(w) \notin$ **Fin**. $\square$