University of Cambridge

Part II of the Mathematical Tripos

Automata & Formal Languages

Lectured by Benedikt Löwe, Michaelmas 2024–25

Notes by Avish Kumar
ak2461@cam.ac.uk
https://ak1089.github.io/maths/notes

Version 2.0

These notes are unofficial and may contain errors. While they are written and published with permission, they are not endorsed by the lecturer or University.

Contents

1 Introduction . . . 3

2 Formal Languages & Grammars . . . 4
2.1 Notation and Preliminaries . . . 4
2.2 Rewrite Systems . . . 6
2.3 Grammars . . . 7
2.4 The Chomsky Hierarchy . . . 8
2.5 Decision Problems . . . 9
2.6 Closure Properties . . . 9
2.7 The Empty Word . . . 11

3 Regular Languages . . . 12
3.1 Regular Grammars . . . 12
3.2 Deterministic Automata . . . 12
3.3 Closure Properties . . . 14
3.4 Non-Deterministic Automata . . . 15
3.5 The Pumping Lemma . . . 16
3.6 The Equivalence Problem . . . 18
3.7 Regular Expressions . . . 21

4 Context-Free Languages . . . 22
4.1 Parse Trees . . . 22
4.2 Chomsky Normal Form . . . 23
4.3 The Context-Free Pumping Lemma . . . 24
4.4 Closure Properties and Decision Problems . . . 26

5 Computability Theory Part 1: Hardware . . . 27
5.1 Register Machines . . . 27
5.2 Performing Operations and Answering Questions . . . 29
5.3 Computable Functions and Sets . . . 31
5.4 Coding Numbers . . . 33
5.5 Primitive Recursive Functions . . . 34
5.6 Coding Languages and Machines . . . 36

6 Computability Theory Part 2: Software . . . 38
6.1 The Software Principle . . . 38
6.2 Computably Enumerable Sets . . . 39
6.3 Closure Properties . . . 41
6.4 The Church-Turing Thesis . . . 42
6.5 Reductions and Solvability . . . 44
6.6 Index Sets and Rice's Theorem . . . 45

7 A Recap of Computation . . . 47

1 Introduction

This course covers a range of topics which mostly fall under computer science, as opposed to mathematics. It's fundamentally about *computation*, and computability: what can be computed, and what can't be? Automata are a model of computation, and will be the approach we take to formalise computation. Meanwhile, formal languages are a framework we use to understand the object of computation.

Really, *languages* and *grammars* are collections of symbols and the rules governing them.

Example 1.1 (Motivating Examples)

We're going to look at a lot of decision problems, which in the most abstract case (for a fixed domain X and property Φ of elements of X) take the form "given $x \in X$, does x satisfy Φ?"

One such basic decision problem is "given a group with n elements, is it abelian?" The brute force algorithm (a general term for algorithms which try every possibility) will enumerate all n^2 pairs and make the relevant comparison. It works, providing an affirmative or negative answer after these n^2 comparisons (or even before, if it finds an early mismatch)!

Another example is the quadratic polynomial decision problem "given integers a, b, c, is there an integer solution to $ax^2 + bx + c = 0$?" In this case, the brute-force algorithm will confirm an integer solution when it gets to it, but if there is in fact no integer solution, it will never halt! Importantly, you can never be sure of a "no" with this algorithm: even checking up to one trillion does not guarantee there are no solutions beyond there.

(Of course, we can find a better algorithm: using the quadratic formula!)

In this second case, we have proved infinitely many theorems simultaneously, using algorithmic solvability. The important thing is we have *one* method by which the decision program is solved.

Remark 1.2 (Hilbert's Tenth Problem)

In 1900, at a conference in Paris, David Hilbert gave a talk called *Mathematical Problems*: highlighting the defining problems of the upcoming twentieth century. The tenth problem of the twenty-three asked: "given a polynomial, is there a process to determine whether the problem has an integer solution in finite steps?"

The mathematicians Davis, Matiyasevich, Putnam, and Robinson eventually came back with the answer: **no**!

This is surprising because the problem is wildly *asymmetric*. It's easy to find a decision procedure, but comparatively much harder to both define decision procedures formally and prove that one *cannot* exist for a particular problem.

Understanding computation well enough to define decision problems and analyse them is essentially what the course is about.

2 Formal Languages & Grammars

2.1 Notation and Preliminaries

Note: In this course, we use the set theoretic convention that $\mathbb{N}$ includes 0. This is as the definition of natural numbers includes the set of all smaller natural numbers (ie. $3 = \{0, 1, 2\}$).

Here, X denotes a finite set of elements, which we call *symbols*. We say X^n (the set of n-tuples of elements of X) are the set of "X-strings of length n". We now formalise this definition.

Definition 2.1 (X-strings)

We again use the set-theoretic convention that

$$\alpha \in X^n \implies \alpha : \{0 \dots n-1\} \to X \text{ with } |\alpha| = n.$$

That is, we define these tuples to be *functions* from an n-element ordered set to X. This means there is precisely one empty string:

$$\varepsilon : \varnothing \to X \text{ (the empty function) } \implies X^0 = \{\varepsilon\}$$

The sets X^* and X^+ are the sets of all X-strings of any (or any positive) length:

$$X^* = \bigcup n \in N X^n \quad X^+ = X^* \setminus \{\varepsilon\}$$

This is of course a (countably) infinite set, but all its *elements* are themselves finite.

For $|\alpha| = n$ and $k < n$, we write the *k-restriction* of α as $\alpha \restriction k$. This is defined naturally, as the first k letters of α: the unique initial segment of α with length k.

Note: For $x \in X$ a letter, we can write the n-string consisting of n copies of x by x^n. (With a bit of notation abuse, we often use x and x^1 interchangeably, even though the former is a symbol and the latter is a string.) We do similarly for X-strings.

Definition 2.2 (Concatenation)

For $\alpha, \beta \in X^*$ with $|\alpha| = m$ and $|\beta| = n$, their *concatenation*, written $\alpha\beta \in X^*$, is the X-string given by the function

$$(\alpha\beta)(k) = \begin{cases} \alpha(k) & k < m \\ \beta(k-m) & \text{otherwise} \end{cases}$$

We can also use a recursive definition for concatenated sequences: $\alpha^0 = \varepsilon$ and $\alpha^{n+1} = \alpha^n\alpha$.

Corollary: This definition has some obvious properties: for example, $\alpha^a\alpha^b = \alpha^{a+b}$.

We can extend functions $X \to Y$ between sets of symbols in the obvious way.

Definition 2.3 (Lift)

If $f : X \to Y$, there is a natural $\hat{f} : X^* \to Y^*$ by concatenation of the function strings. This constructed function is called the *lift* or *extension* of f, and can be defined recursively by:

$$\hat{f}(\varepsilon) = \varepsilon$$
$$\hat{f}(\alpha x) = f(\alpha)f(x) \quad (\alpha \in X^*, x \in X)$$

Now, we consider a special pair of properties of sets. Finite sets are those with a finite number of elements, and infinite sets are those without. In fact, there are special kinds of infinite sets. We formalise this notion here.

Definition 2.4 (Infinite, Countable)

A set X is called *infinite* if there is an injection $\mathbb{N} \to X$.

A set X is called *countable* if there is an surjection $\mathbb{N} \to X$ (or $X = \varnothing$). Otherwise, we say that X is *uncountable*.

Corollary: Any finite set is countable.

Proposition 2.5 (Countability of Products)

Suppose X and Y are countable. Then so is the Cartesian product $X \times Y$.

Proof: If X or Y is empty, then so is the product, so we can assume both are nonempty.

Choose surjections $\pi_X : \mathbb{N} \to X$ and $\pi_Y : \mathbb{N} \to Y$. Then you can trivially construct a surjection $\pi : \mathbb{N} \to X \times Y$ using Cantor's "zigzag" bijection, or $\pi(2^a \cdot 3^b) = (\pi_X(a), \pi_Y(b))$, with $\pi(n)$ taking some other arbitrary output for where n is not of this form. □

Proposition 2.6 (Countability of X^*)

If $X \neq \varnothing$ is countable, then X^* is infinite and countable.

Proof: $\{\varepsilon, x, x^2, \ldots\} \subseteq X^*$ is obviously infinite for $x \in X$, so X^* is infinite. For any n, X^n is countable. X^* is the countable union of all such X^n, and is therefore countable. □

Theorem 2.7 (Cantor's Theorem)

If X is infinite, then its powerset $\wp(X)$ is uncountable.

Proof: Obviously, if X is uncountable, then $\{\{x\} : x \in X\} \subseteq \wp(X)$ is uncountable. So it suffices to show the case where X is countable.

Suppose X and $\wp(X)$ are both infinite and countable. Then there is a surjection $f_1 : X \to \mathbb{N}$, and a surjection $f_2 : \mathbb{N} \to \wp(X)$. By composition, $f = f_2 f_1$ is a surjection $X \to \wp(X)$.

However, consider the set $A = \{x \in X : x \notin f(x)\} \in \wp(X)$. As f is surjective, there is some $x \in X$ with $f(x) = A$. But this is clearly paradoxical, as $x \in A \implies x \notin f(x) = A \implies x \in A$, so no such surjection can exist. Thus f cannot exist, so $\wp(X)$ cannot have been countable. □

Note: This is a general version of Cantor's "diagonal argument". Famously, he used a similar construction to show that the set of real numbers $\mathbb{R}$ is uncountable.

Proposition 2.8 (Finite Subsets Countable)

If X is countable, then the set of *finite* subsets of X is also countable.

Proof: Define S_n as the set of cardinality-n subsets of X, which is clearly countable for all n. Then the set of all finite subsets of X is the countable union of the S_n for all natural n, which is clearly countable. □

Note: This is a similar proof to the one used in Proposition 2.6.

2.2 Rewrite Systems

A language is composed of two basic sets of rules: *syntax* and *semantics*.

1. Semantics define whether a statement is "meaningful" or "meaningless". Importantly, this is an orthogonal question to whether the statement is "true" or "false". "Dogs are cute" is a meaningful sentence with no defined truth value, the sentence "blorgles are gnarbled" does not correspond to anything in reality, and the sentence "I am eight feet tall" is simply false.
2. Syntax, however, merely verifies whether the sentence is "grammatically correct".

Example 2.9 (Syntax $\neq$ Semantics)

The famous sentence

> *"Colourless green ideas sleep furiously."*

was composed by Noam Chomsky in Syntactic Structures (1957). It is the canonical example of a grammatically well-formed sentence that is nevertheless incoherent.

Of course, an idea can neither be colourless nor green, let alone both at the same time; an idea cannot sleep or do so furiously; the sentence does not *mean* anything. However, the pattern *adjective-adjective-noun-verb-adverb* is perfectly legitimate in English.

Note: In this course, we will be studying *syntax*.

Definition 2.10 (Alphabets, Symbols)

An *alphabet* Ω is the finite set of letters, also called *symbols* in a language of interest.

As we saw in 2.1, the set Ω^* is the set of Ω-strings, or *words*, and ε is the *empty word*.

Here, we write $\Omega^+ = \Omega^* \setminus \{\varepsilon\}$ for the set of non-empty strings.

Definition 2.11 (Rewrite Rule)

A *rewrite rule* (also known as a *production rule*) over an alphabet Ω is a member of $\Omega^+ \times \Omega^*$. Equivalently, a rewrite rule over an alphabet Ω is a pair of Ω-strings, where the first string is not the empty word ε.

We often write $\alpha \to \beta$ for the rewrite rule (α, β). Heuristically, this rule means "α can be freely replaced with β".

A tuple (Ω, P), where Ω is an alpabet, is then called a rewrite *system* if P is a finite set of rewrite rules over Ω.

Note: We can think of Ω as being a list of letters, and P as being a list of ways to manipulate words, for example by replacing one letter with two different letters.

Proposition 2.12 (Rewrite Systems Countable)

For any given alphabet Ω, there are only countably many possible rewrite systems over Ω.

Proof: Ω^* is countable. Thus $(\Omega^+ \times \Omega^*) \subseteq (\Omega^* \times \Omega^*)$ is countable. The set of possible rewrite rules is precisely the finite subsets of this set, which is countable by Proposition 2.8. $\square$

How can we think about these?

Remark 2.13 (Interpretation of Rewrite Systems)

Suppose $\sigma, \tau \in \Omega^*$ and $R = (\Omega, P)$ is a rewrite system. Then we write

$$\sigma \xrightarrow{R}_1 \tau \iff \sigma \text{ can be rewritten to } \tau \text{ in one step}$$

Equivalently, there exist $\alpha, \beta, \gamma, \delta \in \Omega^*$ such that $\sigma = \alpha\gamma\beta$, $\tau = \alpha\delta\beta$, and $\gamma \to \delta \in P$.

The relation $\xrightarrow{R}$ is then the reflexive and transitive closure of $\xrightarrow{R}_1$. If $\sigma \xrightarrow{R} \tau$, then either $\sigma \xrightarrow{R}_1 \tau$, or there are $\sigma_1 \dots \sigma_j$ such that

$$\sigma \xrightarrow{R}_1 \sigma_1 \xrightarrow{R}_1 \dots \xrightarrow{R}_1 \sigma_j \xrightarrow{R}_1 \tau$$

This sequence is known as an R-derivation of τ from σ. We say that R *derives* τ from σ in n steps, where n is the number of rewrite rules used.

Of course, this is not how the English language defines its syntactic rules!

2.3 Grammars

From now on, we take Σ to be a set of *letters*, and V to be a set of *variables* disjoint to Σ: we have $\Sigma \cap V = \varnothing$. For the time being, these are merely abstract symbols. We call the letters *terminal* symbols, and the variables *non-terminal* symbols.

Let $\Omega = \Sigma \cup V$ be the set of characters, and $\mathbb{W} = \Sigma^*$ be the set of possible words.

Definition 2.14 (Formal Language)

A *language* over an alphabet Σ is a subset $L \subseteq \mathbb{W} = \Sigma^*$.

Definition 2.15 (Formal Grammar)

A tuple $G = (\Sigma, V, S, P)$ is called a *grammar* if $\Sigma \cap V = \varnothing$ and $(\Sigma \cup V, P)$ is a rewrite system. $S \in V$ is then called the *start symbol*.

We let $\mathcal{D}(G, \alpha)$ be the set of strings derivable from a string α under G. $\mathcal{L}(G) = \mathcal{D}(G, S) \cap \mathbb{W}$ is then the language *generated* by G.

Note: While each grammar defines a language, a language can be any subset of $\mathbb{W}$. We will see later that not every language can be generated by a grammar!

Note: One special case of this is the binary language. $\Sigma_{01} = \{0, 1\}$ is the binary alphabet, and $\mathbb{B} = (\Sigma_{01})^*$ is the set of binary words.

Example 2.16 (Production Rules)

If no production rule is of the form $S \to \cdot$, then the set of derivable strings is simply $\{S\}$, and the language derived is empty.

If every production rule is of the form $\cdot \to \alpha v \beta$, where v is a variable, then the language defined is empty.

If G is a grammar over Σ_{01}, and the rules are $S \to 00S$ and $S \to 0$, then the generated language is precisely odd length strings containing only 0.

Definition 2.17 (Equivalence of Grammars)

Two grammars G and G' are *equivalent* if $\mathcal{L}(G) = \mathcal{L}(G')$.

Proposition 2.18 (Equivalence of Isomorphic Grammars)

If G and G' are isomorphic grammars, then they are equivalent.

Proof: By symmetry, we need only show $\mathcal{L}(G) \subseteq \mathcal{L}(G')$.

Suppose $S \xrightarrow{G} w$. Then there exists some sequence of production rules which takes S to w. By isomorphism, the corresponding rules in G' take $f(S) = S'$ to $f(w)$ as required. □

So far, we have taken V to be a set of arbitrary formal symbols. In fact, this is because only the size of V matters: the elements in it are irrelevant, as long as they are not also in Σ.

Note: Define $\mathcal{G}(\Sigma, V)$ to be the set of all grammars with Σ, V. If $\mathcal{R}$ is the set of rewrite systems over $\Sigma \cup V$, which is countable, then there is a surjection $V \times \mathcal{R} \to \mathcal{G}(\Sigma, V)$.

Proposition 2.19 (Size Matters)

If $|V| = |V'|$, then $\mathcal{L}(\Sigma, V) = \mathcal{L}(\Sigma, V')$.

Proof: Take a bijection $f : V \to V'$ and extend it in the natural way by setting $f(a) = a$ for all $a \in \Sigma$. This is an isomorphism between $G \in \mathcal{G}(\Sigma, V)$ and an equivalent grammar. □

Corollary: As the countable union of $\mathcal{L}_n$ for $n \in \mathbb{N}$ (each of which is countable), the set of languages generated by grammars $\mathcal{L}(\Sigma)$ is countable.

2.4 The Chomsky Hierarchy

Right now, our system of grammars allows for rewriting anything, including letters. We should look at a system of languages which are perhaps more restrictive. In fact, Noam Chomsky created the Chomsky Hierarchy, which does exactly this.

Definition 2.20 (Noncontracting, Contextuality, Regularity)

First, we fix Σ, V, and $S \in V$.

1. A production rule $\alpha \to \beta$ is noncontracting if $|\alpha| \leqslant |\beta|$.
2. With $\gamma, \delta, \nu \in \Omega^*$ and $A \in V$, the rule $\gamma A \delta \to \gamma \eta \delta$ is context-sensitive if $\eta \neq \varepsilon$.
3. A production rule $A \to \beta$ is context-free if $A \in V$ and $|\beta| \geqslant 1$.
4. If $A, B \in V$ and $a \in \Sigma$, then the rules $A \to a$ and $A \to aB$ are regular.

We call a grammar noncontracting, context-sensitive, context-free, or regular if all of its rules satisfy said property, and we give the same title to the language it generates.

Chomsky referred to some languages as being Type 0 (all languages generated by grammars), Type 1 (noncontracting), Type 2 (context-free), and Type 3 (regular).

Note: This is a real hierarchy! Regularity clearly entails context-freedom, which entails context-sensitivity (with $\gamma = \eta = \varepsilon$), which entails being noncontracting (as $|A| = 1 \leqslant |\eta|$).

2.5 Decision Problems

We now return to the idea of decision problems: ones which we might construct algorithms to solve. Right now, we can't yet define that formally: we will return to this shortly.

There are a few basic decision problems related to the definitions we have just seen.

1. The *word problem* asks for an algorithm to determine if $w \in \mathcal{L}(G)$.
2. The *emptiness problem* asks for an algorithm to determine if $\mathcal{L}(G) = \varnothing$.
3. The *equivalence problem* asks for an algorithm to determine if $\mathcal{L}(G) = \mathcal{L}(G')$.

> **Theorem 2.21** (Word Problem Solvable)
>
> The word problem for noncontracting grammars is solvable.

Proof: Firstly, observe that there is a systematic way of listing all G-derivations of length $\leqslant n$, for any n. If there are r rewrite rules, there are r^n possible derivations.

Next, see that for each w, there is an $N \in \mathbb{N}$ such that $w \in \mathcal{L}(G)$ if and only if there is a G-derivation of w with length at most N.

Why? Well, supposing $w \in \mathcal{L}(G)$, we can take a derivation of minimal length. Then using the fact that G is noncontracting, this derivation consists of some words of length 1, then some words of length 2, and so on. Each of these "blocks" of length n has at most $|\Omega|^n$ words: by minimality, there cannot be repetitions.

Then we can define this N precisely:

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

Now we can use the following procedure. List all derivations of length at most N, and check if each of them produces w. If any of them do, return `true`. Otherwise, return `false`. □

2.6 Closure Properties

This subsection is devoted to ways in which we might combine two languages into one, and an analysis of when combining two languages with a particular property might preserve that property.

Take $L, M \subseteq \mathbb{W}$ as languages within the set of words over some alphabet Σ.

1. The *concatenation* language LM is the set $\{vw : v \in L, w \in M\}$.
2. The *union* language $L \cup M$ is the set $L \cup M$.
3. The *intersection* language $L \cap M$ is the set $L \cap M$.
4. The *complement* language $\overline{L}$ is the set $\mathbb{W}^+ \setminus L$.
5. The *difference* language $L \setminus M$ is the set $L \setminus M$.

Note: We a class of languages $\mathcal{C}$ is said to be *closed* under some combination method if any two languages in $\mathcal{C}$ combine in this way to form another language in $\mathcal{C}$.

Note: A class of languages $\mathcal{C}$ which is closed under union and complementation is closed under intersection, and one closed under intersection and complementation is closed under union and difference. This follows from set algebra, and is not unique to languages.

Consider concatenation. Given two grammars $G = (\Sigma, V, S, P)$ and $G' = (\Sigma, V', S', P')$, we take $\Omega = \Sigma \cup V$ and $\Omega' = \Sigma \cup V'$. Is there a new grammar H such that $\mathcal{L}H = \mathcal{L}(G)\mathcal{L}(G')$?

Proposition 2.22 (Concatenation Grammars Exist)

If G and G' are *variable-based* (the left hand side of any production rule contains exclusively variables rather than letters), then one can define

- $V^* = V \cup V' \cup \{T\}$ (where T is a new variable not in V or V').
- $P^* = P \cup P' \cup \{T \to SS'\}$.
- $H = (\Sigma, V^*, T, P^*)$ a new grammar.

Then if $V \cap V' = \varnothing$, H is the concatenation grammar satisfying $\mathcal{L}(H) = \mathcal{L}(G)\mathcal{L}(G')$.

Proof: Any word vw in the language $\mathcal{L}(G)\mathcal{L}(G')$ can be derived using $S \xrightarrow{P} v$ and $S' \xrightarrow{P'} w$ within G and G' respectively. We can derive the same word within H using $T \xrightarrow{H} SS' \xrightarrow{H} \dots \xrightarrow{H} vw$.

Now, if v is a word in $\mathcal{L}(H)$, one can derive the two parts of it from S and S' in G and G', which means we can derive it from T. So the two languages are in fact equivalent. $\square$

Let's look at one of the prerequisites for the proof of this proposition, which is that both grammars are *variable-based*. We see that this is a much weaker condition than may be first assumed.

Proposition 2.23 (Variable-Based is Weak)

Every grammar G is equivalent to some grammar G^+ which is variable-based.

Proof: Fix a grammar $G = (\Sigma, V, S, P)$. For each $a \in \Sigma$, add a unique new variable X_a not in V, and define V' by adjoining V with the X_a.

Then, let $X(\alpha)$ be the string in $(V')^*$ with letters $a \in \alpha$ replaced by their X_a. This allows us to define $P' = \{X(\alpha) \to X(\beta) : \alpha \to \beta \in P\}$.

Then $G' = (\Sigma, V', X(S), P')$ is clearly variable-based, with $S \xrightarrow{G} \alpha \iff S' \xrightarrow{G'} X(\alpha)$.

We then add the "recovery rules". Define $P^+ = P' \cup \{X_a \to a : a \in \Sigma\}$.

Finally, take $G^+ = (\Sigma, V', S, P^+)$. We can then derive $X(\alpha) \xrightarrow{G^+} \alpha$ using the recovery rules! So the languages are equivalent. $\square$

Corollary: Type 0, 1, and 2 languages are closed under concatenation.

Proof: If G and G' are both type k, then without loss of generality use Proposition 2.19 to assume that $V \cap V' = \varnothing$. Then apply Proposition 2.23 to assume that both grammars are variable-based. This does not lose the property of being type k.

Then the conclusion holds by the same construction as in the proof of Proposition 2.22. $\square$

Proposition 2.24 (Union Grammars Exist)

For $G = (\Sigma, V, S, P)$ and $G' = (\Sigma, V', S', P')$ variable-based, define:

- $V^+ = V \cup V' \cup \{T\}$ where all three such sets are pairwise disjoint.
- $P^+ = P \cup P' \cup \{T \to S, T \to S'\}$.
- $G^+ = (\Sigma, V^+, T, P^+)$ a new grammar.

Then G^+ is the union grammar satisfying $\mathcal{L}(G^+) = \mathcal{L}(G) \cup \mathcal{L}(G')$.

Proof: Follow the usual steps to find easy derivations of everything required. Heuristically, the new start variable T can be turned into either S or S', then any word in $\mathcal{L}(G)$ or $\mathcal{L}(G')$ can be derived from there using its derivation in G or G'. $\square$

Corollary: Type 0, 1, and 2 languages are closed under unions.

Proof: If G and G' are both type k, then without loss of generality use Proposition 2.19 to assume that $V \cap V' = \varnothing$. Then apply Proposition 2.23 to assume that both grammars are variable-based. This does not lose the property of being type k.

Then the conclusion holds by the same construction as in the proof of Proposition 2.24. □

Note: This is the same proof as the previous corollary (demonstrating closure under unions instead of under concatenation) except for the replacement of Proposition 2.22 with Proposition 2.24.

Note: Type 3 languages are also closed under unions, but we will not prove this until later.

2.7 The Empty Word

If G is noncontracting, and $w \in \mathcal{L}(G)$ (ie. $S \xrightarrow{G} w$) then $|w| \geqslant |S| = 1$. This means that the empty word $\varepsilon \notin \mathcal{L}(G)$ for any grammar G, which has the strange effect that

$$\begin{aligned} \left\{\mathbf{0}^{2n+1} : n \in \mathbb{N}\right\} & \text{ the odd-length 0-strings is a type 1 grammar.} \\ \left\{\mathbf{0}^{2n} : n \in \mathbb{N}\right\} & \text{ the even-length 0-strings is not?} \end{aligned}$$

This is quite annoying. We want to allow creating the empty word, for consistency's sake!

One idea is to allow the rule $S \to \varepsilon$ (the basic ε-production rule), but without allowing any other noncontracting rules. This allows derivation of ε without totally destroying the noncontracting nature of the language, but unfortunately could cause a lot of problems if S occurs in any derivable sequence. To solve this, we consider the idea of ε-adequate grammars.

Definition 2.25 (S-Safety and ε-Adequacy)

We call a production rule S-*safe* if it does not produce any sequence containing an S.

We call a grammar ε-*adequate* if all its rules are S-safe.

Thankfully, this doesn't cause any problems: we do not have to restrict ourselves to any sort of special subclass of grammars, and may simply assume all grammars are ε-adequate.

Proposition 2.26 (ε-Adequacy Easy)

Every grammar G is equivalent to some grammar G' which is ε-adequate.

Proof: Define $V' = V \cup \{T\}$ (where $T \notin V$) and $P' = P \cup \{T \to \alpha : S \to \alpha \in P\}$.

Then $G' = (\Sigma, V', T, P')$ is clearly ε-adequate, as every rule is T-safe. □

Corollary: For an ε-adequate grammar $G = (\Sigma, V, T, P)$, adjoin a single production rule to make the grammar $G^\varepsilon = (\Sigma, V, T, P \cup \{S \to \varepsilon\})$. Then $\mathcal{L}(G^\varepsilon) = \mathcal{L}(G) \cup \{\varepsilon\}$.

3 Regular Languages

3.1 Regular Grammars

Recall from Definition 2.20 that regular grammars have a highly restricted set of production rules, of only two valid forms. There are *terminal* rules, which take $A \to a$, and *nonterminal* rules, which take $A \to aB$. The length is thus increasing, while the number of variables is non-increasing.

It was a source of frustration that the concatenation and union grammars (2.22 and 2.24) didn't preserve regularity. We need to define a similar operation that certainly does.

Definition 3.1 (Regular Concatenation/Union Grammar)

For regular grammars $G = (\Sigma, V, S, P)$ and $G' = (\Sigma, V', S', P')$, we define the concatenation grammar to be $G^+ = (\Sigma, V \cup V', S, P^+)$, where:

$$P^+ = P' \cup \{\text{nonterminal rules in } P\} \cup \{A \to aS' : A \to a \in P\}$$

This involves simply replacing every terminal rule in P by a nonterminal rule which performs the rule as normal then appends the start string of G'.

We further define the regular union grammar to be $G^+ = (\Sigma, V \cup V' \cup \{T\}, T, P^+)$, where:

$$P^+ = P \cup P' \cup \{T \to \alpha : S \to \alpha \in P\} \cup \{T \to \beta : S' \to \beta \in P'\}$$

This adds a new start symbol T, which can follow the rules set out by either original start symbol from G or G', and is never produced again after the first rule application.

Note: All of the production rules define here are regular production rules, so regularity is preserved by these constructions.

Corollary: The languages generated by the regular concatenation and union grammars really are $\mathcal{L}(G)\mathcal{L}(G')$ and $\mathcal{L}(G) \cup \mathcal{L}(G')$ respectively!

Remark 3.2 (Automata)

We can think of a regular grammar as being like a *machine*. It has one variable at the end at all times, except at the end when it terminates, so it can store information. It has rules about what transitions it can make between variables based on what it has already seen, which are analogous to computations.

We can formalise this intuition using the concept of *automata*.

3.2 Deterministic Automata

Definition 3.3 (Deterministic Automaton)

Fix a set Σ. Then $D = (\Sigma, Q, \delta, q_0, F)$ is called a *deterministic automaton* if

1. Q is a finite set, with its elements called *states*.
2. $q_0 \in Q$ is the *start state*.
3. $F \subseteq Q \setminus \{q_0\}$ is the set of *accept states*.
4. $\delta : Q \times \Sigma \to Q$ is the *transition function*.

Remark 3.4 (Graphical Representation)

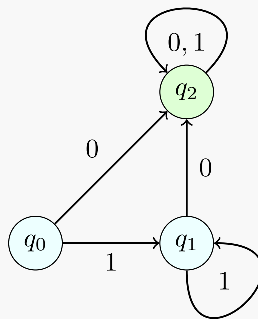
We can represent these graphically! Suppose we have the following automaton:

$$\Sigma = \{\mathtt{0}, \mathtt{1}\}$$
$$Q = \{q_0, q_1, q_2\}$$
$$F = \{q_2\}$$

with the transition function δ being

$$\delta(q_i, 0) = q_2 \qquad \delta(q_i, 1) = \begin{cases} q_1 & i = 0 \\ q_2 & \text{otherwise} \end{cases}$$

then we can represent the automaton as follows:



How do we interpret this? Well, we look at it as a computational machine, where:

1. The computer receives a word $w \in \mathbb{W}$ as its input.
2. The computer keeps track of its state alone. It starts in state q_0.
3. It reads the word letter by letter: if it reads a while in state q, it moves to state $\delta(q, a)$.
4. After reading all of w, the computational machine is in some state q_{end}. If $q_{\text{end}} \in F$, then D `accepts` w, otherwise it `rejects` w.

We define $\mathcal{L}(D)$ as the language generated by D, equal to $\{w : D \text{ accepts } w\}$. For example, the automaton above accepts binary strings if and only if they contain a `0`.

Note: No automaton D can ever accept the empty word, since $q_0 \notin F$.

Definition 3.5 (Automaton Homomorphism)

Now, we take two languages $D = (\Sigma, Q, \delta, q_0, F)$ and $D' = (\Sigma, Q', \delta', q_0', F')$. We say a function $f : Q \to Q'$ is a *homomorphism* if:

(a) For all $q \in Q$ and $a \in \Sigma$, we have $\delta'(f(q), a) = f(\delta(q), a)$.

(b) $f(q_0) = q_0'$.

(c) For all $q \in Q$, $q \in F \iff f(q) \in F'$.

We say a homomorphism f is an *isomorphism* if it is also bijective.

Proposition 3.6 (Homomorphisms and Equivalence)

If f is a homomorphism from D to D', then $\mathcal{L}(D) = \mathcal{L}(D')$.

Proof: w is in $\mathcal{L}(D)$ if and only if $\delta(q_0, w)$ is in F. This is equivalent to $f(\delta(q_0, w))$ being in F', which is equivalent to $\delta'(f(q_0), w)$ being in F'. $f(q_0) = q_0'$, so we are done. $\square$

Without loss of generality, we can assume that $q_0 \notin \text{range}(\delta)$. For every automaton D, we can construct an automaton D' which satisfies the condition such that $\mathcal{L}(D) = \mathcal{L}(D')$.

We do this by defining $Q' = Q \cup \{q^*\}$, where $q^* \notin Q$. hen, take $D' = (\Sigma, Q', \delta', q_0, F)$ with

$$\delta'(q, a) = \begin{cases} \delta(q, a) & q \in Q, \delta(q, a) \neq q_0 \\ \delta(q_0, a) & q = q^*, \delta(q_0, a) \neq q_0 \\ q^* & \text{otherwise} \end{cases}$$

Now, notice that $f : Q' \to Q$, where $f(q^*) = f(q_0) = q_0$ and f is otherwise the identity function, is a homomorphism. Therefore the languages induced by the two automata are equivalent.

3.3 Closure Properties

Regular languages are a useful class of languages because they behave nicely under set operations. In particular, they are closed under concatenation, union, intersection, complement, and difference!

We can also look at union and intersection automata in a different way. Given $Q, Q' \neq \varnothing$, with $F \subseteq Q$ and $F' \subseteq Q'$, define

$$F \wedge F' = \{(q, q') \in Q \times Q' : q \in F \text{ and } q' \in F'\} = F \times F'$$
$$F \vee F' = \{(q, q') \in Q \times Q' : q \in F \text{ or } q' \in F'\}$$

Then, define the product of two transition functions as

$$\delta \times \delta' : \Sigma \times (Q \times Q') \to Q \times Q', (a, (q, q')) \mapsto (\delta(a, q), \delta'(a, q'))$$

allowing us to define the product automata for intersection and union as

$$D \wedge D' = (\Sigma, Q \times Q', \delta \times \delta', (q_0, q_0'), F \wedge F')$$
$$D \vee D' = (\Sigma, Q \times Q', \delta \times \delta', (q_0, q_0'), F \vee F')$$

Proposition 3.7 (Unions and Intersections)

For automata $D = (\Sigma, Q, \delta, q_0, F)$ and $D' = (\Sigma, Q', \delta', q_0', F')$, we have

$$\mathcal{L}(D \wedge D') = \mathcal{L}(D) \cap \mathcal{L}(D') \quad \text{and} \quad \mathcal{L}(D \vee D') = \mathcal{L}(D) \cup \mathcal{L}(D')$$

Proof: Easily verified using the definitions. $\square$

We want to unite the notions of regular languages and automata, precisely in the sense that a language L is regular if and only if there is some automaton D which generates the language L. That is, regular languages are precisely those which are accepted by some computational procedure.

For an automaton $D = (\Sigma, Q, \delta, q_0, F)$, define a grammar $G = (\Sigma, Q, q_0, P)$. This has the same alphabet, and the "variables" it uses are the states of the automaton. Obviously, the grammar must have start symbol q_0, corresponding to the start state of D. The production rules are:

$$P = \underbrace{\{p \to aq : a \in \Sigma, p \in Q, \delta(a, p) = q\}}_{\text{nonterminal rules}} \cup \underbrace{\{p \to a : a \in \Sigma, p \in Q, \delta(a, p) \in F\}}_{\text{terminal rules}}$$

For any word $w \in \mathcal{L}(D)$, the state sequence of the accepting computation in D is the same as the variable sequence of the derivation in Q. The same holds for the converse, with the addition that the final state must be in F (as the last step of the derivation must be a terminal rule).

This gives us the theorem we sought.

> **Theorem 3.8** (Automata are Regular Languages)
>
> Every language which is accepted by an automaton is regular.

Recall that any class of languages $\mathcal{C}$ which is closed under union and complementation is also closed under intersection, while one closed under intersection and complementation is closed under union and difference. So we need only show that the regular languages are closed under complementation.

> **Proposition 3.9** (Closure Under Complementation)
>
> The class of regular languages is closed under the complementation operation.

Proof: Suppose L is a regular language, and is equal to $\mathcal{L}(D)$ for an automaton $D = (\Sigma, Q, \delta, q_0, F)$. Without loss of generality, suppose q_0 is not in the range of δ: if it were, then we could append a new state q_0' to Q, and set $\delta'(q, a) = q_0'$ whenever $\delta(q, a) = q_0$.

Define the new automaton $D' = (\Sigma, Q, \delta, q_0, Q \setminus (F \cup \{q_0\}))$. We claim that $\mathcal{L}(D') = \mathbb{W} \setminus \mathcal{L}(D)$.

Suppose $w \in \mathcal{L}(D')$. Then $w \neq \varepsilon$ and $\hat{\delta}(q_0, w) \notin F$. Then $w \notin \mathcal{L}(D)$. Conversely, suppose $w \in \mathcal{L}(D)$. Then $\hat{\delta}(q_0, w) \in F$, so $\hat{\delta}(q_0, w) \notin Q \setminus (F \cup \{q_0\})$, and thus $w \notin \mathcal{L}(D')$. □

3.4 Non-Deterministic Automata

Recall the definition of a deterministic automaton from 3.3. Let's tweak this definition slightly.

> **Definition 3.10** (Non-Deterministic Automaton)
>
> A tuple $N = (\Sigma, Q, \delta, q_0, F)$ is called a non-deterministic automaton if
>
> 1. Q is a finite set, with its elements called *states* and $q_0 \in Q$ the *start state*.
> 2. $F \subseteq Q \setminus \{q_0\}$ is the set of *accept states*.
> 3. $\delta : Q \times \Sigma \to \wp(Q)$ is the *transition function*.
>
> The only change from deterministic automata is that the domain of the transition function δ is now the *powerset* of Q. We interpret this as the set of next *possible* states, rather than there being a deterministic next state at any point.

We can define the extended transition function by

$$\hat{\delta}(q, \varepsilon) = \{q\} \qquad \text{and} \qquad \hat{\delta}(q, wa) = \bigcup \left\{ \delta(p, a) : p \in \hat{\delta}(q, w) \right\}$$

which makes the language generated by N the set $\mathcal{L}(N) = \left\{ w : \hat{\delta}(q_0, w) \cap F \neq \varnothing \right\}$.

Note: Heuristically, this is the set of words for which you can take *some* allowed path according to the normal transition function, and end up in an allowed state.

One might guess that non-deterministic automata are "more powerful" than their deterministic counterparts, in the same way that quantum computers can run algorithms classical computers cannot. However, in fact this is not true!

Theorem 3.11 (Non-Determinism Doesn't Help)

Given a language L, the following statements are equivalent:

1. L is regular.
2. L is generated by a deterministic automaton D.
3. L is generated by a non-deterministic automaton N.

Proof: We have proven ($2 \Rightarrow 1$) already, in Theorem 3.8.

We can get ($1 \Rightarrow 3$) by constructing an automaton from the regular grammar. The states of the automaton correspond to the non-terminal symbols in the grammar which generates L.

Then, there are transitions between any two non-terminal symbols where a non-terminal rule allows the production of one to the other, with the relevant letter from the alphabet of L. Add a new state (the only accepting state): every terminal rule corresponds to an arrow into this new state.

Finally, to show ($3 \Rightarrow 2$), we can use the *powerset construction*: create a deterministic automaton, with each state corresponding to some element of the powerset of states in the nondeterministic automaton, and draw arrows accordingly. □

3.5 The Pumping Lemma

For $L \subseteq \mathbb{W}$ a language, we say that L satisfies the pumping lemma with pumping number n if for every word $w \in L$ with $|w| \geqslant n$, we have:

1. $w = xyz$ with $|y| > 0$, $|xy| \leqslant n$, and
2. $xy^k z \in L$ for every $k \in \mathbb{N}$.

Note: If L satisfies the pumping lemma for some n, we just say it "satisfies the pumping lemma".

Theorem 3.12 (The Pumping Lemma)

Every regular language L satisfies the pumping lemma with some pumping number n.

Proof: By Theorem 3.11, $L = \mathcal{L}(D)$ for a deterministic automaton $D = (\Sigma, Q, \delta, q_0, F)$. We claim that L satisfies the pumping lemma with pumping number $n = |Q|$.

Suppose $w \in L$ with $|w| \geqslant n$. Then we can write

$$w = a_0 a_1 \dots a_{n-1} v \qquad \text{with } a_i \in \Sigma, v \in \mathbb{W}$$

The state sequence corresponding to the derivation of w must be

$$\underbrace{q_0 \to q_1 \to q_2 \to \cdots \to q_n}_{n+1 \text{ states}}$$

where each step from q_i to q_{i+1} goes through a_i. But by the pigeonhole principle, there must be two identical states reached: some $q_i = q_j$ with $i < j$. Then, define

$$x = a_0 \dots a_{i-1} \qquad y = a_i \dots a_{j-1} \qquad z = a_j \dots a_{n-1} v$$

This obviously satisfies $w = xyz$ and $|y| > 0$. Also, $|xy| = j \leqslant n$, so the pumping lemma conditions are satisfied with this setup.

But then we can repeat y any number of times (possibly none), since starting from state q_i and reading in y brings us back to $q_j = q_i$. Thus, since regular languages only depend on state and input, repeating y any number of times does not affect whether a particular word is in L. □

Corollary: There are context free languages which are not regular.

Proof: Consider the language $L = \{\mathbf{0}^n\mathbf{1}^n : n > 0\}$. This can be generated by the rules $S \to \mathbf{0}S\mathbf{1}$ and $S \to \mathbf{01}$, so it is context-free.

Suppose it is regular. Then it satisfies the pumping lemma, say with pumping number N. Consider the word $w = \mathbf{0}^N\mathbf{1}^N$, with $|w| \geqslant N$. We can find $w = xyz$ with $|xy| \leqslant N$ and $|y| > 0$. But then $x = \mathbf{0}^k$, $y = \mathbf{0}^\ell$, with $\ell > 0$. The pumping lemma implies that $\mathbf{0}^{N+l}\mathbf{1}^N \in L$, but this is clearly contradicts the definition of L (since $\ell > 0$). □

Example 3.13 (Zero-Prefixed Language)

For a fixed n, the language $L = \{0^n w : w \in \mathbb{W}\}$ is regular, and the smallest automaton D such that $L = \mathcal{L}(D)$ has at least n states.

To show this by contradiction, assume there is such an automaton. By the proof of the regular pumping lemma (3.12), L must satisfy the pumping lemma with pumping number n. But $w = 0^n \in L$ with $|w| = n$, and this word can be pumped down.

The words x, y, z are entirely zeroes: by pumping y down to nothing, we get a strictly shorter word xz. But this is a sequence of $n - |y| < n$ zeroes, and so cannot possibly be in L.

Corollary: The proof we have used here also implies that for any automaton D with n states with a path from q to q', there is a path from q to q' of length at most n.

One might wonder if the pumping lemma in fact *characterises* the regular languages, rather than simply being a property of them. That is, if L is a language which satisfies the regular pumping lemma with some pumping number, must it be regular?

Proposition 3.14 (Pumping Lemma Not Exclusive)

If L is a language satisfying the regular pumping lemma, it is *not* necessarily regular.

Proof: (Not constructive.) Consider the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. We write $\text{tail}(w)$ for the number of `1`s after the last occurring `0` in w, so that $\text{tail}(\mathtt{0110010111}) = 3$. For a (possibly infinite) set $X \subseteq \mathbb{N}$, define the language $L_X \subseteq \{\mathtt{0}, \mathtt{1}\}^*$ as containing all the words w such that either $\text{tail}(w) \in X$ or w does not contain a `0`.

Suppose $X \neq Y$. Then without loss of generality, there is some $n \in X \setminus Y$, which means $\mathtt{01}^n \in L_X \setminus L_Y$. Thus $L_X \neq L_Y$. Thus the function $X \mapsto L_X$ is an injection from the powerset of $\mathbb{N}$ into the set of languages of the form L_X.

All of these languages satisfy the pumping lemma with PN 2, so consider $w \in L_X$ with $|w| \geqslant 2$.

1. If w starts with a `0`, then take $x = \varepsilon$, $y = \mathtt{0}$, and z accordingly. Pumping up produces $\mathtt{0}^k z \in L_X$, and pumping down produces $z \in L_X$. If z contains a `0`, then prepending $\mathtt{0}^{k-1}$ does nothing, otherwise it is in L_X regardless.
2. If w starts with a `1`, then take $x = \varepsilon$, $y = \mathtt{1}$, and z accordingly. Pumping up produces $\mathtt{1}^k z \in L_X$, and pumping down produces $z \in L_X$. If z contains a `0`, then prepending $\mathtt{1}^{k-1}$ does nothing, otherwise it is in L_X regardless.

Separately, every regular language is generated by a grammar. By Proposition 2.12, there are only countably many regular grammars over a fixed language: only the size of the variable set matters, and for each $n \in \mathbb{N}$, the set of regular grammars with n variables is countable.

By Theorem 2.7, the powerset of $\mathbb{N}$ is uncountable. By the existence of an injection, there are uncountably many languages L_X, which we have shown satisfy the regular pumping lemma. But there are only countably many regular languages. Therefore satisfying the regular pumping lemma cannot be a sufficient condition to be a regular language. □

3.6 The Equivalence Problem

Recall from Proposition 3.6 that if we have a homomorphism f between two automata, then they are *equivalent*: they accept the same language.

Definition 3.15 (Accessible, Indistinguishable)

For a regular automaton $D = (\Sigma, Q, \delta, q_0, F)$ we say that a state q is *accessible* if there is some word w such that $\hat{\delta}(q_0, w) = q$.

Consider another automaton, with states Q'. If $q' \in Q'$ is accessible, it must be in the range of any homomorphism $f : Q \to Q'$. For such a homomorphism, then if $f(q) = f(q')$, then we say q and q' are *indistinguishable*.

We claim that indistinguishability is an *equivalence relation*. We can define the quotient automaton as the automaton $(D/\sim) = (\Sigma, Q/\sim, [\delta], [q_0], [F])$.

Proposition 3.16 (Quotient Automaton)

The quotient automaton is well-defined, and no two of its states are indistinguishable.

Proof: Suppose $q \sim q' \in Q$ and consider $\delta(q, a)$ and $\delta(q', a)$. If these are distinguished by a word, then so are q and q'. Therefore $\delta(q, a) \sim \delta(q', a)$.

Also, $[q] \sim [q']$ if and only if $q \sim q'$, so $[q] = [q']$. □

Corollary: For every deterministic automaton D, $\mathcal{L}(D) = \mathcal{L}(D/\sim)$.

Proof: The quotient map $q \mapsto [q]$ is a homomorphism. □

Definition 3.17 (Irreducible)

An automaton D is called *irreducible* if there are no inaccessible states and no two states are indistinguishable.

Proposition 3.18 (Homomorphisms on Irreducible Automata)

If f is a homomorphism between automata $D \to D'$, then

1. If D is irreducible, then f is an injection.
2. If D' is irreducible, then f is a surjection.

Proof: If $f(p) = f(q)$, then p and q must be indistinguishable. Further, if q' is not in the range of f, then it must be inaccessible. □

Corollary: If both D and D' are irreducible, then f is a bijection.

From now on, we write $D \cong D'$ to indicate that two deterministic automata are equivalent: that is, they generate the same language $\mathcal{L}(D) = \mathcal{L}(D')$.

Why are irreducible automata so important? We now show that the irreducible automata can be identified directly with languages: there is a one-to-one correspondence.

Theorem 3.19 (Irreducible Automata General)

For every deterministic automaton D, there is an equivalent irreducible automaton D'.

Proof: If q is accessible, then all states of the form $\delta(q, a)$ with $a \in \Sigma$ are also accessible. So if $A \subseteq Q$ is the set of accessible states in D, then we can define the restriction δ^* of δ on $A \times \Sigma$ (instead of $Q \times \Sigma$).

Then, if $w \in \mathcal{L}(D)$, then $\delta(q_0, w) \in F \cap A$. Thus $w \in \mathcal{L}(D) \iff w \in \mathcal{L}(D^*)$.

Now consider $D' = D^*/\sim$, the quotient automaton of D^*. This preserves the property of having no inaccessible states, and preserves the language generated. □

In fact, we can also prove a stronger claim. Irreducible automata are unique up to isomorphism for a given language!

Theorem 3.20 (Uniqueness of Irreducible Automata)

If $I \cong I'$ are two irreducible automata which generate the same language, then there must be an isomorphism between them.

Proof: Any two irreducible automata which are equivalent must have a homomorphism between them. By the corollary to Proposition 3.18, we know that this homomorphism must be a bijection, and thus an isomorphism. □

Proposition 3.21 (Minimal Automata)

For each deterministic automaton D, there is an irreducible automaton $I \cong D$, unique up to isomorphism, called the minimal automaton for $\mathcal{L}(D)$, with at most as many states as D.

Proof: Start with D. Remove inaccessible states to get D'. Then let $I = D'/\sim$. □

This gets us to the important part of this subsection, which is the equivalence problem for regular grammars. Let's finish building up to it.

Proposition 3.22 (Finitely Many Words to Check)

If $L \neq \varnothing$ satisfies the regular pumping lemma with pumping number n, then there is a word $w \in L$ with $|w| < n$.

Proof: L must have a shortest word. If $|w| \geqslant n$, then it can be pumped down, so it cannot be the shortest word. Thus the shortest word must be of length less than n. □

Proposition 3.23 (Regular Emptiness Determinable)

There is an algorithm which takes in regular grammars G as input and determines whether $\mathcal{L}(G) = \varnothing$. That is, the emptiness problem for regular grammars is solvable.

Proof: There is a deterministic automaton D such that $\mathcal{L}(D) = \mathcal{L}(G)$, with at most $2^{|V|+1}$ symbols. Thus $\mathcal{L}(G)$ satisfies the regular pumping lemma with pumping number at most $2^{|V|+1}$. Check all the finitely many possible words up to this length to see if they are in $\mathcal{L}(D)$. If none of them are, then $\mathcal{L}(G) = \varnothing$. □

Proposition 3.24 (Inaccessibility Determinable)

There is an algorithm which takes in deterministic automaton and determines which states, if any, are inaccessible.

Proof: A state q is accessible if and only if there is some w with $|w| \leqslant |Q|$ such that $\delta(q_0, w) = q$. There are finitely many such words, which we may check individually. □

Proposition 3.25 (Indistinguishability Determinable)

There is an algorithm which takes in two states of a deterministic automaton and determines whether they are indistinguishable.

Proof: The process we will use is called the *table filling algorithm.* Write $Q \times Q$ as an $|Q| \times |Q|$ table. We can ignore the major diagonal, by reflexivity and the lower-left triangle by symmetry.

First, we check all pairs (q, q') and mark them as distinguished if one and only one is a member of F. These are distinguished by the word ε, which we will call the witness.

In subsequent steps, we can check every unmarked square. For each $a \in \Sigma$, write $q_* = \delta(q, a)$ and $q'_* = \delta(q', a)$. If (q_*, q'_*) is marked with w as the witness, then (q, q') are distinguished by aw.

At the end of each step of the algorithm, check whether a new pair has been marked. If so, then keep going. If not, then we can terminate early. Regardless, the algorithm must terminate: there are finitely many entries to be filled in the table.

Then q and q' are indistinguishable if and only if (q, q') is marked in the table. If the pair is marked by w, then either one is in F and the other is not, or w distinguishes q and q'.

If there was a pair distinguished by a word not marked by the end, then it must be distinguished by a word w of minimal length. Find such a pair with the overall shortest minimal distinguishing word: $|w| > 0$. Then let a be the first letter of w, so $w = av$.

Consider $q_* = \delta(q, a)$ and $q'_* = \delta(q', a)$. They are clearly distinguished by v. But they cannot be marked (as (q, q') would be marked in the subsequent step), contradicting minimality.

Thus we can fill in the entire table, halting in at most as many steps as the table has cells, and at the end have determined whether any possible pair of states q and q' has any witness distinguishing them, that is a word w such that precisely one of $\hat{\delta}(q, w)$ and $\hat{\delta}(q', w)$ is in F.

But this is just the definition of two states being distinguishable, as we have seen. Therefore this construction gives us a way to check if two states are indistinguishable: if their cell in the grid is not marked with a witness when the algorithm terminates. □

This brings us to the conclusion of this chapter: a positive solution to the equivalence problem for regular grammars!

Theorem 3.26 (The Equivalence Problem)

Given two deterministic automata D and D', there is an algorithm to determine whether $D \cong D'$, or equivalently check that they have the same language $\mathcal{L}(D) = \mathcal{L}(D')$.

Proof: Construct irreducible automata I and I' using the construction given in the proof of Proposition 3.21, which we can do in finite time.

These are unique up to isomorphism by Theorem 3.20: we then need only verify that I and I' are isomorphic. But we can do this systematically too.

Firstly, notice that if I and I' have different numbers of states, they cannot be isomorphic.

If they both have n states, there are only $n!$ possible bijections. These can be listed systematically, then checked through independently to verify whether they are isomorphisms. □

Note: This is really very special! Given two computers, this result means that we are able to look at them and systematically determine whether they accept the same inputs.

3.7 Regular Expressions

We now look at regular expressions, which are powerful tools for analysing regular languages. They are used frequently in computer programming to parse text.

Definition 3.27 (Regular Expression)

Given an alphabet Σ, we define the augmented alphabet

$$\overline{\Sigma} = \Sigma \cup \left\{\varnothing, \epsilon, (,), +, ^+, ^*\right\}$$

and define the regular expressions over Σ to include *exactly* the expressions given by the rules:

1. The symbols $\varnothing$ and ε are regular expressions.
2. Every $a \in \Sigma$ is a regular expression.
3. If R is a regular expression, then so are R^+ and R^*.
4. If R and S are regular expressions, then so are (RS) and $(R + S)$.

In fact, the parentheses are usually unnecessary, since the operations we are dealing with are typically associative. We drop them as often as possible for convenience, and take the concatenation operation $(R, S) \mapsto RS$ as having a higher priority than the union operation $(R, S) \mapsto R + S$.

Definition 3.28 (Kleene Plus/Star)

If L is a language, the Kleene Plus is defined by

$$L^+ = \{w : \exists w_0, w_1, \dots w_n \in L \text{ s.t. } w = w_0 w_1 \dots w_n\}$$

(that is, finite concatenations of elements of L). The Kleene Star is then defined to be this set plus the empty word: $L^* = L^+ \cup \{\varepsilon\}$.

Now, we move on to our motivation for studying regular expressions: associating them with regular languages. In fact, as we shall soon see, we associate them with *essentially regular* languages: that is, languages which are either regular or would be but for the empty word ε.

Definition 3.29 (Language Associated with Regular Expression)

We can assign languages $\mathcal{L}(E)$ to regular expressions E, again recursively.

1. If $E = \varnothing$, then $\mathcal{L}(E) = \varnothing$. Similarly, if $E = \varepsilon$, then $\mathcal{L}(E) = \varepsilon$.
2. If $E = a$ for $a \in \Sigma$, then $\mathcal{L}(E) = \{a\}$.
3. If R is a regular expression, then $\mathcal{L}(R^*) = \mathcal{L}(R)^*$ and $\mathcal{L}(R^+) = \mathcal{L}(R)^+$.
4. If R and S are regular expressions, then $\mathcal{L}(R + S) = \mathcal{L}(R) \cup \mathcal{L}(S)$.
5. If R and S are regular expressions, then $\mathcal{L}(RS) = \mathcal{L}(R)\mathcal{L}(S)$.

Theorem 3.30 (Regular Expressions are Essentially Regular Languages)

L is essentially regular if and only if there is a regular expression R over Σ with $L = \mathcal{L}(R)$.

Proof: Omitted. □

Corollary: The class of regular languages is closed under the Kleene Plus operation.

4 Context-Free Languages

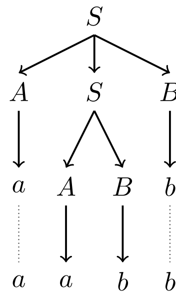
Context-free rules are generally of the form $A \to \alpha$, where $A \in V$ and $\alpha \in \Omega^+$. In general, the English language does not have this property: it is context-sensitive. For example, whether you can turn the variable V (for *verb*) into “go” or “goes” depends on the context of whether the preceding noun is singular or plural.

Firstly, note that these are not just the regular languages. The simplest example is the language $\{0^n 1^n : n \in \mathbb{N}\} \subseteq \{0,1\}^*$. This is not regular, by the pumping lemma. However, it is context-free: it is generated by the rules $S \to 0S1$ and $S \to 01$.

4.1 Parse Trees

Derivations of words in a context-free language give rise to a sort of tree structure. The order of such derivations is mostly irrelevant: in fact, it forms a partial order.

This tree is associated with a label for each node. For example, suppose we have the rules $S \to ASB$, $S \to AB$, $A \to a$ and $B \to b$. Then, we can derive $aabb$ in the obvious way, but this derivation can be done in many different orders. This can be represented by the below *parse tree*.



A parse tree $\mathbb{T} = (T, \ell)$ starting from A is a tree T and a labelling function $\ell : T \to \Omega$ such that the root of T is labelled A, and $\ell(t) \in \Sigma$ if and only if t is a terminal node (leaf) in T.

In fact, this imposes a total order, first by the level hierarchy of the tree and then by the left-to-right ordering given by the structure of the derivations. We write $\sigma_{\mathbb{T}} \in \mathbb{W}$ for the left-to-right of the labels of the terminal nodes: the word that the tree depicts a derivation of.

Proposition 4.1 (Parse Trees Depict Everything)

There is a parse tree $\mathbb{T}$ starting from S using only rules in a given grammar G such that $\sigma_{\mathbb{T}} = w$ if and only if $w \in \mathcal{L}(G)$.

Proof: If $A \to w$ is a G-derivation, we can write it uniquely into a parse tree starting from A such that $\sigma_{\mathbb{T}} = w$. Also, if any $\mathbb{T}$ is a parse tree starting from A using only rules from G, then there must be a derivation $A \to \sigma_{\mathbb{T}}$. □

We say that a parse tree has a *height* equal to the longest path from the root to a leaf. For example, the tree given above has height 3.

For a parse tree $\mathbb{T}$ with a nonterminal node t, then $\mathbb{T}_t$ is the subtree starting at t.

If $\ell(t) = B$, then this is a parse tree in its own right starting from B. We can graft trees into each other using this concept. If $\mathbb{T}'$ is a G-parse tree starting from B, then we can cut out $\mathbb{T}_t$ and replace it by $\mathbb{T}'$ to graft the new tree into the gap.

By context-freeness, $\mathbb{T}^* = \text{graft}(\mathbb{T}, t, \mathbb{T}')$ is a G-parse, and if $\sigma_{\mathbb{T}} = v\sigma_{\mathbb{T}_t}v'$, then $\sigma_{\mathbb{T}^*} = v\sigma_{\mathbb{T}'}v'$.

4.2 Chomsky Normal Form

For regular languages, we had the very nice property that a derivation of any word w had length $|w|$. We want to determine a similar property for context-free languages which guarantee a derivation length. Towards that end, we standardise context-free grammars.

Definition 4.2 (Chomsky Normal Form)

We say that a grammar $G = (\Sigma, V, P, S)$ is in *Chomsky normal form* if all its production rules are in one of the two forms:

1. $A \to BC$ for variables $A, B, C \in V$
2. $A \to A$ for a variable $A \in V$ and a letter $a \in \Sigma$.

Grammars in this form have several nice properties. All of them are context-free, by definition. Additionally, the parse trees they generate are all standardised: they their nodes are at most binary branching. That is, each node is either a leaf or has at most two descendents.

Proposition 4.3 (CNF Derivations)

If w is a word derived in G a grammar in Chomsky normal form, then any G-derivation of w has length $2\,|w| - 1$.

Proof: Call rules of the form $A \to BC$ binary, and rules of the form $A \to a$ unary. A binary rule increases the number of variables in the string and the string's length by 1, while a unary variable preserves length and decreases variable count by 1.

We start with S, which has a single variable and is of length 1. We must reach a length of $|w|$, which requires precisely $|w| - 1$ applications of a binary rule. Then, variable count is $|w|$, and we must bring it to 0, which requires $|w|$ unary rule applications, for a total of $2\,|w| - 1$. □

Proposition 4.4 (CNF Parse Tree Height)

If G is a grammar in CNF and $\mathbb{T}$ is a height $h+1$ G-parse tree, then if $\sigma_{\mathbb{T}} = w \in \mathbb{W}$, $w \leqslant 2^h$.

Proof: $|w|$ is the number of leaves in $\mathbb{T}$. Parse trees of CNF grammars are at most binary branching, so there are at most 2^{h+1} leaves. Any derivation must have at least $|w|$ unary rule applications, which each decrease the number of leaves by at least one. So $|w| \leqslant 2^{h+1} - |w|$, thus $|w| \leqslant 2^h$ as required. □

So Chomsky normal form is very useful for analysing context-free grammars. It would be nice to assume that such grammars are in Chomsky normal form without loss of generality. It turns out that we can indeed do this!

Theorem 4.5 (Chomsky's Theorem)

For any context-free grammar $G = (\Sigma, V, P, S)$, there is an equivalent grammar G' in Chomsky normal form such that $\mathcal{L}(G) = \mathcal{L}(G')$.

Proof: We need to prove some intermediary results in order to obtain a complete proof of this theorem. First, we define a *problematic production* as a production rule $A \to \alpha$ if $|\alpha| > 1$ and α contains a variable, and a *unit production* if α is a single variable.

We show that we can assume away problematic productions. For each $a \in \Sigma$, we introduct a new variable X_a. For $\alpha \in \Omega^*$, we can denote by $X(\alpha)$ the string α with each letter a replaced by X_a. Then, we adjoin the X_a to V to give $V' = V \cup \{X_a : a \in \Sigma\}$.

Next, remove all problematic productions $A \to \alpha$, and replace them by $A \to X(\alpha)$. Also, add the rules $X_a \to a : a \in \Sigma$ to P'. The new grammar $G' = (\Sigma, V', P', S)$ is clearly equivalent.

We call a grammar *unit closed* if $(A \to B) \in P$ and $(B \to \alpha) \in P$ implies $(A \to \alpha) \in P$. We can assume without loss of generality that this holds, forming the unit closure by iteratively adding extra rules until G is unit closed. The number of steps is bounded by $|V|\,|P|$, and no step changes the language.

Now, we have a unit closed grammar G which is free of problematic productions. We can freely remove all unit productions from it to obtain G'. We must show that $\mathcal{L}(G) \subseteq \mathcal{L}(G')$ to prove that we have not changed the language.

The shortest derivation of any word cannot use unit productions. If it did, we would have to have used $A \to B$ and $B \to b$, but then by unit closure the rule $A \to b$ is already in B and can have been used instead. Therefore, it cannot have been the shortest, and so removing these productions does not change the language.

Now, we assume that G is a context-free grammar which is free of problematic productions, unit closed, and does not have unit productions. If it is not already in Chomsky normal form, there must be some rule in P which violates the conditions, which is necessarily of the form $A \to \alpha = A_0 \dots A_n$.

Now, we define $V' = V \cup \{X_0 \dots X_{n-2}\}$ by adding $n-1$ new variables not already in V. We remove this rule from P and adjoin the rules

$$P' = P \setminus \{A \to \alpha\} \cup \{A \to A_0X_0,\, X_0 \to A_1X_1,\, \dots,\, X_{n-3} \to A_{n-2}X_{n-2},\, X_{n-2} \to A_{n-1}A_n\}\,.$$

We consider $G' = (\Sigma, V', P', S)$. This new grammar is clearly equivalent: any derivation in G' must use the new X_i only in derivations of $A \to \alpha$, and by context-freeness we can assume they are used in this order and consecutively.

Now, we can finally prove any context-free grammar G is equivalent to a grammar in Chomsky normal form. Assume without loss of generality that G is free of problematic productions, that it is unit closed, that it does not have unit productions, and that it does not have any rules of the form $A \to \alpha$ where $|\alpha| > 1$.

By the intermediary results we have shown, none of the processes we used to convert G change the language, and G now satisfies all the conditions required for Chomsky normal form. □

4.3 The Context-Free Pumping Lemma

Continuing our pattern of extending nice results from regular languages into slightly weaker versions which hold for context-free languages, we return to the pumping lemma.

Definition 4.6 (The Context-Free Pumping Lemma)

For $L \subseteq \mathbb{W}$ a language, we say L satisfies the context-free pumping lemma with pumping number n if for every word $w \in L$ with $|w| \geqslant n$, there are words $u, v, x, y, z \in \mathbb{W}$ such that $w = xuyvz$, $|uv| > 0$, $|uyv| \leqslant n$, and for all $k \in \mathbb{N}$ we have $xu^kyv^kz \in L$.

As before, we say that L satisfies the context-free pumping lemma if it satisfies the context-free pumping lemma for some pumping number n.

Proposition 4.7 (Context-Free Weaker than Regular)

Every language L which satisfies the regular pumping lemma also satisfies the context-free pumping lemma.

Proof: Take $y = v = \varepsilon$. Clearly, $|uv| = |u| > 0$, and $|uyv| = |u| \leqslant |xu| \leqslant n$. Also, we have $xu^k yv^k z = xu^k z$, which is simply the statement of the regular pumping lemma. □

Corollary: As there are only countably many context-free languages over any given alphabet Σ, this lemma cannot characterise any of our classes of languages (there are uncountably many languages satisfying the regular pumping lemma, and thus the context-free pumping lemma).

The proof that every context-free language satisfies this lemma is usually attributed to Yehoshua Bar-Hillel, and is sometimes named after him.

Theorem 4.8 (Context-Free Pumping Lemma)

(Also known as the *Bar-Hillel Lemma*). For every context-free language L, there is a natural number n such that L satisfies the context-free pumping lemma with pumping number n.

Proof: By Chomsky's Theorem (4.5), there is a grammar $G = (\Sigma, V, P, S)$ in Chomsky normal form such that $L = \mathcal{L}(G)$. Let $m = |V|$ and $n = 2^m + 1$. Then we show that L satisfies the context-free pumping lemma with pumping number n.

Take $w \in L$ with $|w| \geqslant n$, and consider a G-parse tree $\mathbf{T}$ starting at S with $\sigma_{\mathbf{T}} = ww$. Then the height of $\mathbf{T}$ is at least $m+1$ by Proposition 4.4. So there is a terminal node $t \in \mathbf{T}$ which is at a height of at least $m+1$, and we can choose $s \in \mathbf{T}$ such that the height of $\mathbf{T}_s$ is exactly $m+1$.

The sequence leading from s to t has $m+2$ nodes: all but the last are labelled with variables, and the last is the derivation $\cdot \to \ell(t)$ with $\ell(t) \in \Sigma$.

By the pigeonhole principle, there are $t_0 \neq t_1$ in this sequence such that $\ell(t_0) = \ell(t_1) = A \in V$, since there are only $|V| = m$ possible non-letter variables for the $m+1$ nodes.

This means $\mathbf{T}_{t_0}$ and $\mathbf{T}_{t_1}$ are both G-parse trees starting at A. Then, we can define

$$\begin{aligned} \sigma_{\mathbf{T}} &= x_0 \sigma_{\mathbf{T}_s} z_1 \\ \sigma_{\mathbf{T}_s} &= x_1 \sigma_{\mathbf{T}_s} z_0 \\ \sigma_{\mathbf{T}_{t_0}} &= u \sigma_{\mathbf{T}_{t_1}} v \quad \text{and} \quad \sigma_{\mathbf{T}_{t_1}} = y \end{aligned}$$

which makes $\sigma_{\mathbf{T}}$ equal to $x_0 x_1 uyvz_0 z_1 = xuyvz = w$ as required (taking $x = x_0 x_1$ and $z = z_0 z_1$).

Now, w then satisfies all the length bounds. Grafting in the subtrees recursively gives us $xu^k yv^k z$ for all k as desired. Thus G satisfies the context-free pumping lemma. □

Let's apply this lemma to prove a language is not context-free. Specifically, take $L = \{a^n b^n c^n : n > 0\}$ on the alphabet $\Sigma = \{a, b, c\}$.

Proposition 4.9 (Not Context-Free)

The language $L = \{a^n b^n c^n : n > 0\}$ on the alphabet $\Sigma = \{a, b, c\}$ is not context-free.

Proof: Suppose it were context-free and satisfied the context-free pumping lemma with pumping lemma n. Then consider $w = a^n b^n c^n$, which satisfies $|w| = 3n \geqslant n$. □

If we write $w = xuyvz$, then if $|uv| > 0$ but $|uyv| \leqslant n$, we cannot have the subword uyv contain both an a and a c (for it would have to contain all n occurrences of b in between).

If it does not contain a c, then pumping down changes the number of a or b without altering c, and otherwise pumping down changes the number of b or c without altering a.

Thus L cannot satisfy the context-free pumping lemma, and is thus not context-free.

Remark 4.10 (Memory)

We have showed that $L_2 = \{a^n b^n : n > 0\}$ is not a regular language. A useful heuristic to apply here is that regular languages do not have arbitrarily large memory. In fact, they can only store exactly as much information as their state count allows.

What is the analogy here for context-free languages? Well, $L_2 = \{a^n b^n : n > 0\}$ *is* context-free, so it seems like context-free languages are more powerful and possess memory of sorts. But $L_3 = \{a^n b^n c^n : n > 0\}$ is not context-free, which implies that in the process of reading off b^n to verify the count is right, the information was lost or destroyed.

We won't look at the actual construction here, but context-free languages are equivalent to a special kind of computer called a *pushdown automaton*. This is like a deterministic automaton which has a *stack*: a storage unit in which one can push letters onto the stack or pop them off, using the last-in-first-out rule. δ then also specifies any stack operations and can depend on the stack.

4.4 Closure Properties and Decision Problems

We know that context-free languages are closed under union and concatenation. Remember that by set algebra, a class of language closed under union and complementation must be closed under intersection, since $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$. This means that if the class of context-free languages is not closed under intersection, it cannot be closed under complementation. We will now show this to be the case.

Proposition 4.11 (No Closure Under Intersection)

The class of context-free languages is not closed under intersection.

Proof: We have seen that $\{a^n b^n : n > 0\}$ and $\{c^n : n > 0\}$ are context-free. Then while their concatenation $\{a^n b^n c^m : n, m > 0\}$ is too, and by the same principle so is $\{a^m b^n c^n : n, m > 0\}$, the intersection of these two is $\{a^n b^n c^n : n > 0\}$, which we have seen is not context-free. □

Now, we move on to studying the word problem, the emptiness problem, and the equivalence problem for context-free languages. We have previously seen that the word problem is solved for noncontracting (and therefore context-free) languages. Let's look at the emptiness and equivalence problems.

Proposition 4.12 (Context-Free Emptiness is Determinable)

There is an algorithm which takes in context-free grammars G as input and determines whether $\mathcal{L}(G) = \varnothing$. Compare this with the proof that the emptiness problem is solvable (Proposition 3.23).

Proof: By a similar proof to the version for regular languages, we can show that if L satisfies the context-free pumping lemma with pumping number n, it must have a word of length less than n. (Otherwise, a "minimal" word could be pumped down to get an even shorter word, and would thus not be minimal).

Given a grammar G, we can just check every word up to $n = 2^{|V|} + 1$, which we showed in the proof of the context-free pumping lemma (4.8) was the pumping number. □

By contrast, the equivalence problem for context-free grammars is in fact undecidable, though we will not prove this in this course.

5 Computability Theory Part 1: Hardware

5.1 Register Machines

Fix an alphabet Σ and a non-empty finite set Q, which we will call the set of *states*. We will use these sets to define a specific type of computer called a *register machine*. We can give the machine instructions: for $k \in \mathbb{N}$, $a \in \Sigma$, and $q, q' \in Q$, we say that

$$\begin{array}{rll} (0,k,a,q) & \text{aka.} & +(k,a,q) & (\text{"add"}) \\ (1,k,a,q,q') & \text{aka.} & ?(k,a,q,q') & (\text{"check"}) \\ (2,k,q,q) & \text{aka.} & ?(k,\varepsilon,q,q') & (\text{"check"}) \\ (3,k,q,q') & \text{aka.} & -(k,q,q') & (\text{"remove"}) \end{array}$$

are (Σ, Q)-instructions. We interpret these as follows:

1. "add": append the letter a to the content of register k and go to state q.
2. "check": check whether the last letter in register k is an a (alternatively if $a = \varepsilon$, if it is empty) and go to q if so or q' if not.
3. "remove": check whether register k is empty. if it is, go to state q, otherwise remove its last letter and go to state q'.

Definition 5.1 (Register Machine)

A tuple $M = (\Sigma, Q, P)$ is called a Σ-register machine if

1. Q is a finite set with two special elements $q_S \neq q_H$ called the *start* and *halt* states.
2. P is a function on Q where the range of P consists of (Σ, Q)-instructions.

P is then called the *program*, and for each state $q \in Q$ we refer to $P(q)$ as a program line.

Since $\mathbb{N}$ is infinite, we might be tempted to conclude that these machines contain infinitely many registers. But in fact, since q is finite, there is a maximal n which appears in any program line. We call the register count $n+1$ the *upper register index* of the machine, in which case M is a device with $n+1$ storage units which can contain words in $\mathbb{W}$.

At any given time, the situation of the register machine is determined by its state and what is in all the registers. We call $C = (q, w_0, \ldots, w_n) \in Q \times \mathbb{W}^{n+1}$ a *configuration* or *snapshot* of M, made up of the state q and the register content. We then say that M *transforms* C to C' if any of the following statements are true:

1. $P(q) = +(k, a, q')$ and $C' = (q', w_0, \ldots w_{k-1}, w_k a, w_{k+1}, \ldots, w_n)$.
2. $P(q) =?(k, a, q', q'')$ for $a \in \Sigma$ and either $w_k = wa$ for some $w \in \mathbb{W}$ and $C' = (q', w_0, \ldots, w_n)$, or alternatively $w_k \neq wa$ for any word w and $C' = (q'', w_0, \ldots, w_n)$.
3. $P(q) =?(k, \varepsilon, q', q'')$ and either $w_k = \varepsilon$ and $C' = (q', w_0, \ldots, w_n)$ or alternatively $w_k \neq \varepsilon$ and $C' = (q'', w_0, \ldots, w_n)$.
4. $P(q) = -(k, q', q'')$ and either $w_k = \varepsilon$ and $C' = (q', w_0, \ldots, w_n)$ or alternatively $w_k = wa$ for some $w \in \mathbb{W}, a \in \Sigma$ and $C' = (q', w_0, \ldots w_{k-1}, w, w_{k+1}, \ldots, w_n)$.

This model of a register machine is really quite abstract. The machine has a state, and two of these states are special. There are also registers, which store words as stacks.

We can think of M as modelling computation. The machine starts in the start state q_S. We give it the input $\overline{w}$ of $n+1$ words in its registers, and define the sequence of computational snapshots.

Specifically, if M has upper register number n, and $\overline{w} = (w_0 \dots w_n) \in \mathbb{W}^{n+1}$, then the computation sequence of M with input 1 is defined as

$$C(0, M, \overline{w}) = (q_S, \overline{w})$$
$$C(k+1, M, \overline{w}) = C' \text{ where } M \text{ transforms } C(0, M, \overline{w}) \to C'$$

For convenience, we often talk about "input w" for $w \in \mathbb{W}$, with the understanding that this is given to the first register, with the input to the other n registers being the empty word ε.

> **Remark 5.2** (Turing Machines)
>
> There is a long history of computation, most notably characterised by Alan Turing's work on defining machines similar to these. *Turing machines* are similar to these register machines, but instead have a single infinite tape which serves as the input, workpad, and the output all in one place.
>
> The theory of computation was further pioneered by Joachim Lamber, Zdzislaw Melzak, Marvin Minsky, John C. Shepherdson, Howard E. Storgis, and of course, John von Neumann.
>
> In fact, these register machines use von Neumann architecture, which is a simplified version of what real-world computers are based on! There is a finite set of storage cells which can be independently accessed and modified.

Recall our definition of a stack, in the aside 4.10. Registers are also examples of last-in-first-out (LIFO) storage methods, and in fact are equivalent. In general, accessing any information which is not at the top of the stack destroys it. However, since we have multiple registers, we can always copy the information elsewhere.

We then say M halts on input $\overline{w}$ if there is some integer k such that $C(k, M, \overline{w})$ has the halt state q_H. If such a k exists, then we say M converges on $\overline{w}$, otherwise it diverges. The smallest such k is then known as the *halting time*, with the register content at this time being the *output* of the machine M when ran on input $\overline{w}$.

> **Definition 5.3** (Strong Equivalence)
>
> We say that two machines M and M' are *strongly equivalent* if for every input $\overline{w}$ and every integer k, the register content of the two machines is equal at time k and the state of M at time k is halting if and only if the state of M' at time k is halting.

Heuristically, the machines behave the exact same way on the same inputs.

> **Proposition 5.4** (Countably Many Machines)
>
> Up to strong equivalence, there are only countably many register machines.

Proof: For each upper register index n and each $|Q| = m$, there are $2(n+1)m$ "add" instructions, $3(n+1)m^2$ "check" instructions, and $(n+1)m^2$ "delete" instructions. Together, this gives a total of $(n+1)m(4m+2)$, and so we can have $((n+1)m(4m+2))^m$ possible register machines.

Every register machine is strongly equivalent to one of these machines for some n and m. But the number of possible pairs of m and n has cardinality $\mathbb{N} \times \mathbb{N}$, which is countable.

Since the countable union of finite sets is countable, the set of all equivalence classes of register machines is countable, proving the proposition. □

In some sense, strong equivalence really is quite restrictive!

Proposition 5.5 (The Padding Lemma)

For each register machine M, there are infinitely many strongly equivalent register machines.

Proof: We prove the following statement, which is sufficient to prove the proposition. For any register machine M with state set of size $|Q| = n$, there is a strongly equivalent register machine with state set of size $n + 1$.

Let $M = (\Sigma, Q, P)$ be a register machine with upper register number n. Consider a new state $\hat{q} \notin Q$. Define a new program P^+, which is the same as P when acting on Q, but when acting on $\hat{q}$ gives $P(\hat{q}) = ?(0, \varepsilon, \hat{q}, \hat{q})$. Then $M^+ = (\Sigma, Q \cup \{\hat{q}\}, P^+)$ has one extra state.

But M^+ is strongly equivalent, since if C is a configuration with state in Q, M^+ and M will transform it identically. Repeating this gives unlimited strongly equivalent register machines. □

This is called the *padding lemma* for padding the state set of the machine with useless states.

5.2 Performing Operations and Answering Questions

It's time for our computers to actually do something. First, we consider partial functions, which are like functions but are not necessarily defined everywhere.

Definition 5.6 (Partial Function)

We say f is a *partial function* from X to Y if the domain of f is a subset of X and the range of f is a subset of Y, and write $f : X \dashrightarrow Y$. Additionally, we say $f(x)\downarrow$ if $x \in \text{dom}(f)$ and $f(x)\uparrow$ otherwise, or f converges/diverges on x.

How is this related to computation?

Definition 5.7 (Performing Operation)

Fix an upper register index n. For a partial function $F : \mathbb{W}^{n+1} \dashrightarrow \mathbb{W}^{n+1}$, we say that register machine M *performs the operation* F if, given input $\overline{w} \in \mathbb{W}^{n+1}$:

1. If $F(\overline{w})\uparrow$, then M diverges on $\overline{w}$.
2. If $F(\overline{w})\downarrow$, then M converges on $\overline{w}$ with register content $F(\overline{w})$ at time of halting.

This is useful! We finally have a model of a computer taking in input and producing output. What sort of operations can a computer perform?

Example 5.8 (Example Operations)

Suppose F has an empty domain. Then M performing F will never converge, which means it performs the operation "keep running forever without halting". Not very useful!

For the opposite example, suppose F is the total identity function with $F(\overline{w}) = \overline{w}$ for all $\overline{w} \in \mathbb{W}^{n+1}$. We can write $P(q_S) = ?(0, \varepsilon, q_H, q_H)$ to define a machine achieving this task.

How powerful are these computers, really? A natural question to ask here is which functions are computed, if not all? Is there a class of partial functions which can be computed, and if so what are some properties of this class?

Here, we see that they are closed under concatenation (equivalently, function composition).

Theorem 5.9 (Subroutine Lemma)

If there are machines M and M' performing operations F and F', then there is a machine $\hat{M}$ performing $F' \circ F$.

Proof: Without loss of generality, suppose that $Q \cap Q' = \{q_H\}$ and that $q_H = q'_S$. Let $\hat{Q} = Q \cup Q'$ and $\hat{P} = P^* \cup P'$, where P^* is P with $(q_H, P(q_H))$ removed.

Then $\hat{M} = (\Sigma, \hat{Q}, \hat{P})$, where q_S is the start state and q'_H is the halt state, performs $F' \circ F$. □

Now, we turn to the idea of *questions*. Rather than simply getting our computers to perform operations, we want them to perform *useful* operations! Since computers run on binary, from this point on, we will take the alphabet to be $\Sigma = \{\mathbf{0}, \mathbf{1}\}$ accordingly, so that $\mathbb{W} = \mathbb{B}$. We will see later that this does not affect the strength of our computing power.

Definition 5.10 (Questions and Answers)

A *question* about $(n+1)$-tuples with $k+1$ *answers* is a partition $W = \{A_0 \ldots A_k\}$ of $\mathbb{B}^{n+1}$. with $k+1$ parts. A register machine answers the question W if it has $k+1$ answer states $\hat{q}_0 \ldots \hat{q}_k$, and for every $\overline{w} \in \mathbb{B}^{n+1}$:

1. M takes input $\overline{w}$ and in a finite number of steps reaches one of the states $\hat{q}_i$.
2. This $\hat{q}_i$ corresponds exactly to the part A_i which contains $\overline{w}$.

There are machines which answer all sorts of questions. For example, "Does register i end with a $\mathbf{0}$?" is decidable by a register machine: one can define $A_0 = \{\overline{w} : w_i = v\mathbf{0}\}$ and $A_1 = \mathbb{B}^{n+1} \setminus A_0$, then take the only program instruction to be $q_S \mapsto ?(i, \mathbf{0}, \hat{q}_0, \hat{q}_1)$.

Proposition 5.11 (Case Distinction Lemma)

Let $W = \{A_i : 0 \leqslant i \leqslant k\}$ be a question with $k+1$ answers, and $f_i : \mathbb{B}^{n+1} \dashrightarrow \mathbb{B}^{n+1}$ be a sequence of $k+1$ operations.

If W is answered by a register machine $M = (Q, P)$, and f_i is performed by $M_i = (Q_i, P_i)$ for all i, then we can construct a register machine that performs the operation $g(\overline{w}) = f_i(\overline{w})$ for $\overline{w} \in A_i$.

This is called the *case distinction lemma* because it involves performing different functions based on cases of which part w belongs to.

Proof: Suppose $M = (Q, P)$ has start symbol q_S and answers W with and states $\hat{q}_i$. Suppose further that $M_i = (Q_i, P_i)$ performs f_i with start symbols q^i_S and halt symbols q^i_H.

Now suppose without loss of generality that $Q \cap \left(\bigcap_{i \leqslant k} Q_i\right) = \left\{q^i_S : i \leqslant k\right\}$. Then, we take $\hat{q}_i = q^i_S$ and combine the machines accordingly. □

If $F : \mathbb{B}^{n+1} \dashrightarrow \mathbb{B}^{n+1}$, define the iteration of F on $\overline{w}$ recursively: $F^0(\overline{w}) = \overline{w}$, $F^{k+1}(\overline{w}) = F(F^k(\overline{w}))$.

If W is a question with only two answers A_0 and A_1, then we define the repetition $R_{F,W}(\overline{w}) = F^m(\overline{w})$ if m is the least integer such that $F^m(\overline{w}) \in A_0$, and undefined otherwise.

Proposition 5.12 (Repeat Lemma)

If F is performed by a machine, and W is answered by a machine, then $R_{F,W}$ is performed by a machine.

Proof: If $M = (Q, P)$ performs F and $M' = (Q', P')$ answers W with $\hat{q}_0$ or $\hat{q}_1$, construct $\hat{M}$ with start state q'_S, identifying $\hat{q}_1$ with q_S and q_H with q'_S, and letting $\hat{q}_0$ be the halt state. □

Example 5.13 (More Possible Computations)

Here are some more things we can do with register machines:

1. Replace the content of register i with w.
2. Copy/move the final letter from register i to register j, if it exists.
3. Move register i to register j in the reverse/correct order.
4. Copy register i to register j.

We have seen the idea of computers performing operations and answering questions, and we have extended our idea of computation through the three useful lemmas proved in this subsection.

Remark 5.14 (The Three Register Machine Lemmas)

Consider the Subroutine Lemma (5.9), the Case Distinction Lemma (5.11), and the Repeat Lemma (5.12). These are constructive! They give an explicit method of contructing a register machine which has the properties we desire.

This implies that descriptions of how to build a register machine are in fact precise enough to identify the machine.

We often have unused registers, known as *scratch space*. Building register machines with extra registers increases their size, but is required to preserve important computational properties. These are sometimes genuinely necessary. For example, we cannot swap the values in two registers without using an intermediary third register to store temporary values.

5.3 Computable Functions and Sets

If M is a register machine and $k > 0$, define the partial function $f_{M,k} : \mathbb{B}^k \dashrightarrow \mathbb{B}$ by

$$f_{M,k}(\overline{w}) \uparrow \implies M \text{ does not halt on } \overline{w}.$$
$$f_{M,k}(\overline{w}) = v_0 \implies M \text{ halts on } \overline{w} \text{ with output } \overline{v}.$$

Importantly, here we only care about the content of the first register at halting time, if the program halts! This is why the range of $f_{M,k}$ is an element of $\mathbb{B}$ rather than $\mathbb{B}^k$.

This matches our intuition of *scratch space*: it isn't relevant to the overall computation. All other registers apart from the first are deemed scratch space.

However, this calls into question our overly strict definition of strong equivalence! It is, of course, perfectly possible that there are multiple register machines which perform the same important computation in slightly different ways, utilising scratch space differently to always arrive at the same output. Intuitively, this shouldn't matter. We will explore this further soon.

Definition 5.15 (Computable)

A partial function $f : \mathbb{B}^k \dashrightarrow \mathbb{B}$ is called *computable* if there is a register machine M and natural number k such that $f = f_{M,k}$. Note that:

1. If M and M' are strongly equivalent, then $f_{M,k} = f_{M',k}$.
2. The converse of this does *not* hold.

By Proposition 5.4, there are only countably many computable functions. However, by the Padding Lemma (5.5), each computable function has infinitely many M with $f = f_{M,k}$.

Which functions are computable? We know a few already: the identity is computable, and so are constant functions and projections.

Definition 5.16 (Characteristic Function, Computable)

For a subset $A \subseteq \mathbb{B}^k$, define the *characteristic function* $\chi_A(\overline{w})$ of A as

$$\chi_A(\overline{w}) = \begin{cases} \mathbf{1} & \text{if } \overline{w} \in A \\ \mathbf{0} & \text{otherwise} \end{cases}$$

and the *pseudo-characteristic* function $\psi_A(\overline{w})$ as

$$\psi_A(\overline{w}) = \begin{cases} \mathbf{1} & \text{if } \overline{w} \in A \\ \uparrow & \text{otherwise} . \end{cases}$$

We then call the set A *computable* if χ_A is computable. Alternatively, we say A is *computably enumerable* if ψ_A is computable.

Note: Historically, these sets were instead referred to as *recursive* and *recursively enumerable*. Intuitively, these two notions might seem like they are equivalent, but in fact they are not! Not every computably enumerable set is computable, as we shall see later.

We've taken a long detour into computability theory, so now we take a minute to link back to the study of languages.

Theorem 5.17 (Regular Languages are Computable)

Every regular language $L \subseteq \mathbb{B}$ is computable.

Proof: Fix a deterministic automaton $D = (\Sigma_{\mathbf{01}}, Q, \delta, q_0, F)$ with $\mathcal{L}(D) = L$. We will construct the register machine $\hat{M} = (\hat{Q}, \hat{P})$ to mimic the behaviour of D.

For each state $q \in Q$, we construct a subset $Q_q \subseteq \hat{Q}$ of mimicking states. While we are in a state from Q_q, we are replicating the behaviour of q, and we only leave the subset when we are done.

Now, how does this replication work? First, in order to set up the machine, we reverse the order of w into register 1 (since automata and register machines read in the opposite order). We then move into the subset of Q_{q_0} in order to replicate the start state.

When we enter a state in Q_q, we read and remove the final letter in register 1 (say b) and then move into the subset $Q_{q'}$, where $q' = \delta(q, b)$. If there are no letter remaining in register 1, we either empty register 0 and halt (if $q \notin F$) or we empty register 0 and write a into it (if $q \in F$). □

The algorithms we defined in the proof of Theorem 2.21 can be performed by a register machine. This means that if G is noncontracting, then $\mathcal{L}(G)$ is computable. Thus:

$$\text{regular} \implies \text{context-free} \implies \text{noncontracting} \implies \text{computable} \implies \text{computably enumerable.}$$

Proposition 5.18 (Computability)

Let $X \subseteq \mathbb{B}^k$. Then if X is computable, so is $\mathbb{B}^k \setminus X$ (closure under complementation) and ψ_X is computably enumerable if and only if there is a computable pseudo-characteristic function. Specifically, X is computably enumerable if and only if it is the domain of a computable partial function.

Proof: Consider the clearly computable $g : \mathbb{B} \to \mathbb{B}$ defined by $g(\varepsilon) = a$ and $g(w) = \varepsilon$ otherwise. Then $\chi_{X^c} = g \circ \chi_X$. The second part is shown by composing ψ with a constant function. □

5.4 Coding Numbers

Coding numbers are our way of encoding every possible sequence in $\mathbb{B}$. We can enumerate these as if they were binary sequences, going:

$$\varepsilon, \mathbf{0}, \mathbf{1}, \mathbf{00}, \mathbf{01}, \mathbf{10}, \mathbf{11}, \mathbf{000}, \mathbf{001}, \mathbf{010}, \mathbf{011}, \mathbf{100}, \mathbf{101} \dots$$

This really is a sensible ordering: eventually, every finite sequence will be reached. We have the encoding function $\# : \mathbb{B} \to \mathbb{N}$, and its inverse $\#^{-1} : \mathbb{N} \to \mathbb{B}$, which define the bijection given by $\#(w) = 2^{|w|} + b(w) + 1$ (where $b(w)$ involves reading w as if it were a binary integer and converting it to decimal).

Note: This ordering is usually called the *shortlex* ordering, in reference to it preferring shorter words and breaking ties lexicographically.

Note: Prepending a **1** to each word (listed in shortlex order) and reading each of them as binary representations of integers yields the natural numbers $\mathbb{N}$ in order.

This means if we have a partial *numerical* function $f : \dashrightarrow \mathbb{N}^k \to \mathbb{N}$, we can define the composition:

$$\mathbb{B}^k \xrightarrow{\#} \mathbb{N}^k \xrightarrow{f} \mathbb{N} \xrightarrow{\#^{-1}} \mathbb{B}$$

which gives us $f^{\#}(\overline{w}) = \#^{-1}(f(\#(w)))$. This means we can encode any familiar numerical function as a function in the binary world! We say that such a function f is computable if its encoding $f^{\#}$ is, and we say that $A \subseteq \mathbb{N}^k$ is computable or computably enumerable if $\{\overline{w} : \#(\overline{w}) \in A\}$ is.

Corollary: By this definition, constant functions and projections $\mathbb{N}^k \dashrightarrow \mathbb{N}$ are computable. As a special case of projection, the identity function is computable.

Proposition 5.19 (Successor Computable)

The *successor* function $x \mapsto x + 1$ is computable.

Proof: Consider $s : \mathbb{B} \to \mathbb{B}$, with $s(w)$ being the immediate successor of w.

Take the unused register k, and empty it. Reverse the content of register 0, and repeat the following procedure indefinitely: check if the final letter of register 0 is a **1**, if so remove it and write **0** in register k. If register 0 ever ends in a 0, replace it by a 1 and finish repeating, and if register 0 is ever empty, write a 0 into it and finish repeating.

When finished repeating, copy the content of register k into the end of register 0. □

What do we do with the numerical information contained in register machines? The ability to refer to numbers using their codes allows us to represent operations which require this information.

Proposition 5.20 (Indexing Computable)

The function "given the content v of register i, what is the letter of register j at index v?" is computable.

Proof: We can do this by taking two registers k and l and emptying them. Copy the content of j into k in reverse order. Repeatedly remove the last letter of k and apply the successor function to l. When l contains v, return the last letter of k. □

That's not all we can do! We can also refer to the count of computation steps, often defined as analogous to time. We may want a machine to run for a fixed number of steps, which we call *truncation*. It is possible to make a machine which simulates another one for only a certain length of time, which we construct using a *count-through argument*.

Definition 5.21 (Truncation Sets)

For M a register machine and $k, n \in \mathbb{N}$, define the three truncation sets

$$T_{M,k,n} = \{\overline{w} \in \mathbb{B}^k : M \text{ halts on input } \overline{w} \text{ in at most } n \text{ steps.}\}$$
$$T_{M,k} = \{(\overline{w}, u) \in \mathbb{B}^{k+1} : M \text{ halts on input } \overline{w} \text{ in at most } \#u \text{ steps.}\}$$
$$\hat{T}_{M,k} = \{(\overline{w}, u, v) \in \mathbb{B}^{k+2} : (\overline{w}, u) \in T_{M,k} \text{ and } M \text{ outputs } v \text{ in register } 0.\}$$

Proposition 5.22 (Truncation Sets are Computable)

The three sets $T_{M,k,n}$, $T_{M,k}$, and $\hat{T}_{M,k}$ are all computable.

Proof: We do this for $\hat{T}_{m,k}$: the others are easier.

Describe the machine which computes χ for this set. Given input $(\overline{w}, u, v)$, we empty the unused register l. After each step of the M-computation, we do the following subroutine:

1. Check whether the content of l is equal to u.
2. If so, write **0** in register 0 and halt.
3. Otherwise, apply the successor function to l.

If M ever reaches its halting state at any point, check whether register 0 is equal to v. If so, replace the content of register 0 by a **1**, otherwise replace it by a **0**. □

Corollary: The famous *halting problem* shows that we cannot in general define a machine which simulates other machines and determines whether they will halt eventually. However, we *can* do it when the problem is restricted to halting within finite time, as this construction demonstrates.

5.5 Primitive Recursive Functions

In 1931, Kurt Gödel published a paper proving his famous *incompleteness theorem.* Alonzo Church (of the Church-Turing thesis) defined a more important and slightly larger class of functions he called the *recursive* functions. Now, we call Gödel's functions the *primitive recursive* functions.

Definition 5.23 (Composition, Recursion)

Suppose we have the partial numerical functions $f : \mathbb{N}^k \dashrightarrow \mathbb{N}$, $g : \mathbb{N}^{k+2} \dashrightarrow \mathbb{N}$, and $g_1 \ldots g_k : \mathbb{N}^l \to \mathbb{N}$. Then, we define the partial numerical *composition* function c by:

$$c : \mathbb{N}^l \dashrightarrow \mathbb{N},\ \overline{n} \mapsto f(g_1(n) \ldots g_k(n))$$

We also define the partial numerical *recursion* function r by:

$$r(\overline{n}, 0) = f(\overline{n}) \qquad r(\overline{n}, m+1) = g(\overline{n}, m, r(\overline{n}, m)).$$

This allows us to define a class of functions closed under these operations.

Definition 5.24 (Primitive Recursive Functions)

The class of *primitive recursive functions* is the smallest class of partial functions containing the identity function, constant functions, projection functions, and the successor function, which is closed under both composition and recursion.

Finally, we can build addition using the basic functions and operations!

First, $\pi_{1,0}(w) = w$ is a basic function, and thus recursive. Next, so is the function $\pi_{3,2}(w, v, u) = u$. The successor function is basic, thus $s \circ \pi_{3,2}(w) = s(u)$ is too. Finally, apply recursion to the first and last functions to get

$$h(w, \varepsilon) = \pi_{1,0}(w)$$
$$h(w, s(v)) = s(\pi_{3,2}(w, v, h(w, v))) = s(h(w, v))$$

The function h defined here is primitive recursive. This is the *Grassmann equation for addition*. Written more simply, we have $h(n, 0) = n$ and $h(n, m + 1) = s(h(n, m))$, which indeed gives $h(n, m) = n + m$.

We can apply more recursions to get multiplication and exponentiation.

In general, the *count-through argument* involves creating a machine which initialises a counter and runs some procedure repeatedly, incrementing the counter using the successor function each step. Then, we can use the counter to halt based on a condition.

It is, of course, much harder to show that functions are primitive recursive than that they are possible to accomplish via register machines. For example, we do not even know if primitive recursive functions have arbitrary case distinction yet!

Let us show that a few basic functions are primitive recursive, using only composition and recursion.

1. $f_0(n) = \text{sgn}(n) = 0$ for $n = 0$ and 1 otherwise, called the signum. This is trivial, since recursion has a built-in basic version of case distinction. We can simply write $f_0(0) = 0$ and $f_0(n + 1) = 1$.
2. $f_1(n) = p(n) = 0$ for $n = 0$ and $n - 1$ otherwise, called the predecessor. This is again easy: we have $f_1(0) = 0$ and $f_1(n + 1) = n$.
3. $f_2(n, m) = n -^* m = \max\{n - m, 0\}$, called the cut-off subtraction. We can define this by $n -^* 0 = n$, and $n -^* (m + 1 = p(n -^* m))$.
4. $f_3(n, m) = |n - m|$ the absolute difference. We can define $f_3(n, m) = f_2(n, m) + f_2(m, n)$.
5. $f_4(n, m) = 1$ if $n \neq m$ and 0 if $n = m$. We can use $f_4(n, m) = f_0(|n - m|)$.

These functions let us build up to the modulus function, given by $r(n, m) = k$ with $0 \leqslant k < m$ and $n \equiv k \bmod m$. We can define this recursively by

$$r(0, m) = 0 \qquad r(n + 1, m) = (r(n, m) + 1) \cdot f_4(r(n, m), f_1(n)).$$

Theorem 5.25 (Primitive Recursives Computable)

Every primitive recursive function is computable.

Proof: By induction, basic functions are computable. By the Subroutine Lemma (Theorem 5.9), compositions of computable functions are computable.

If f and g are computable, then so is the recursion. On input $(\overline{w}, n)$, a machine can check if $n = 0$ and return $f(\overline{w})$ if so. Otherwise, the machine can compute $g(\overline{w}, n)$ and the predecessor of n then run the same process. □

We can now do something interesting: splitting and merging words. Consider Cantor's famous *zigzag bijection* (used in the diagonal argument):

$$z : \mathbb{N}^2 \to \mathbb{N} : (i, j) \mapsto j + \frac{(i + j)(i + j + 1)}{2}.$$

This is the composition of primitive recursive functions, and thus primitive recursive itself. We can consider the coding $z^{\#}$ of this function, called the *merging function*: if $z : \mathbb{N}^2 \to \mathbb{N}$, then $z^{\#} : \mathbb{B}^2 \to \mathbb{B}$ is a computable function!

We can also consider its inverse $z^{\#^{-1}} : \mathbb{B} \to \mathbb{B}^2$, the *splitting function.* Of course, this is strictly not a computable function, since its range is not a subset of $\mathbb{B}$, but it can be performed by a register machine. Clearly, the bijection can be extended inductively to $\mathbb{B}^n$ for any n.

Remark 5.26 (Bounded Search)

We have seen that any *bounded search* problem is doable by iterated checking. For total computable functions $b : \mathbb{B}^k \to \{0, 1\}$ and sets $R \subseteq \mathbb{B}^{k+1}$, bounded search problems are problems of the form "is there a solution $\overline{w}$ which is less than $b(v)$ to $(v, \overline{w}) \in R$?"

Here, b is the *bounding function.* A corollary of this definition and Theorem 5.25 is that for any problem, if one can give a bound expressed in standard arithmetical functions for the length of search, then the process of searching for witnesses is computable.

By the proof of Theorem 2.21, this means that searching for witnesses in the noncontracting word problem is computable! We can do this using a count-through argument once again.

Consider the minimisation problem. Denote Church's recursive functions by $\mathcal{R}$, and the closure of the primitive recursive functions under minimisation by $\mathcal{P}$. Then $\mathcal{P} \subseteq \mathcal{R}$ (the converse does not hold, due to recursive functions possibly being partial).

5.6 Coding Languages and Machines

So far, we have only considered machines over the language $\Sigma = \{\mathbf{0}, \mathbf{1}\}$. However, this is not actually a restriction! For an arbitrary Σ, suppose that m is such that $|\Sigma| < 2^m$. Then we can code the letters of Σ using elements of the set $\{\mathbf{0}, \mathbf{1}\}^m$, to which there is an injection.

Let $i : \{\mathbf{0}, \mathbf{1}\}^n \to \Sigma$ be the encoding function sending these binary representations to the actual letters. Then for functions $\mathbb{W}^k \dashrightarrow \mathbb{W}$, we can define $f^i : \mathbb{B}^k \to B$ by $f^i = i \circ f \circ i^{-1}$. We say f is computable if f_i is, and $A \subseteq \mathbb{W}^k$ is computable/computably enumerable if $\{i(\overline{w}) : \overline{w} \in A\}$ is.

So we can now talk about functions on the natural numbers, the English language, or any other finite set being computable without worrying about the choice of alphabet.

The definition of a register machine (the machine itself, without input or register content) looks like a sequence of state-instruction pairs. Without loss of generality, we can assume that q_S is never referred to by instructions. Then we can label the states by binary numbers: q_H is $\mathbf{0}$, and subsequent states q_k are simply k in binary.

We can simplify this to list the instructions in order, separated by commas. For example,

$$q_S \mapsto -(2, q_2, q_H) \quad q_H \mapsto ?_\varepsilon(1, q_H, q_H) \quad q_1 \mapsto +_\mathbf{0}(2, q_H)$$

can then be written as

$$-(\mathbf{1}, \mathbf{1}, \mathbf{0}), ?_\varepsilon(\mathbf{0}, \mathbf{0}, \mathbf{0}), +_\mathbf{0}(\mathbf{1}, \mathbf{0}).$$

Replacing the commas by slashes and the brackets with square brackets, we can thus encode the register machine as a string in the alphabet

$$\Sigma = \{\mathbf{0}, \mathbf{1}, +_\mathbf{0}, +_\mathbf{1}, -, ?_\varepsilon, ?_\mathbf{0}, ?_\mathbf{1}, [,], /\}.$$

This is a big step! We now have a language which codes machines themselves. We can encode this language in $\{\mathbf{0}, \mathbf{1}\}$ as before, which means we have unique binary strings for each character and thus each register machine, allowing us to define computable functions *on register machines*!

Note: In particular, $|\Sigma| = 11 < 2^4$, so we can encode each character with 4 bits.

A register code can be represented by a regular expression. For example,

$$R(+_{\mathbf{0}}) = +_{\mathbf{0}}[(\mathbf{0}+\mathbf{1})^*/(\mathbf{0}+\mathbf{1})^*].$$

Note: This interpretation makes it clear why we avoided the use of brackets: indeed, they have a particular interpretation within the world of regular expressions!

We can then take the union of all of these groups of codes:

$$\begin{aligned} R &= R(+_{\mathbf{0}}) + R(+_{\mathbf{1}}) + R(-) + R(?_{\varepsilon}) + R(?_{\mathbf{0}}) + R(?_{\mathbf{1}}) \\ R_{\text{RM}} &= R(/R)^* \end{aligned}$$

That is, R generates the language of possible program instructions, and the regular expression generating the language of possible register machines is just a non-empty slash-separated list of program codes, here R_{RM}.

Note: In fact, there is a slight caveat, that some of these will point to nonexistent states. We can take these to implicitly point to the halt state q_H.

A configuration given by $(q, \overline{w})$ can be encoded as a set of finite sequences of binary numbers:

$$R_C = (\mathbf{0}+\mathbf{1})^* \left(/(\mathbf{0}+\mathbf{1})^*\right)^*$$

so the set of configuration codes is regular and thus computable. We now observe that:

1. the sets of codes of register machines and configurations are both computable.
2. the *transformation function* $f_T : \mathbb{B}^2 \to B, (\text{code}(M), \text{code}(C)) \to \text{code}(C')$, where M is a machine transforming C to C' is computable.
3. the *computation sequence* $f_{CS} : \mathbb{B}^3 \to B, (\text{code}(M), \text{code}(C), v) \to \text{code}(C')$, where M is a machine transforming C to C' in $\#v$ steps is computable.

The first point is by regularity, the second is by the subroutine lemma, and the third is by the closure of computability under recursion.

This is a powerful result! Not only is there a register machine that can read in codes for register machines and determines if they are valid, there is a register machine that can take in a register machine and *simulate it running on an arbitrary input*! This is truly incredible.

6 Computability Theory Part 2: Software

6.1 The Software Principle

Let's consider some history first.

Remark 6.1 (Historical Context to Computers)

In the 1920s, a *computer* referred to a person who performed computation. Friedrich Leibniz famously believed that this was tedious and needed to be automated. He wrote:

> *"It is beneath the dignity of excellent men to waste their time in calculation when any peasant could do the work just as accurately with the aid of a machine."*

When Alan Turing was working on the Bombe, a machine to crack the Enigma code used by the Germans in World War II, he realised that the only hurdle was speed. The cipher could trivially be broken by brute force, trying every computation, but humans were simply far too slow to break the ciphers. This was a huge forward step, even though it was a machine which only performed one operation. In fact, the German Navy used a slight variation on the Enigma machine, and the Bombe was useless in attempting to crack it!

Stepping back even earlier, Charles Babbage's *difference engine* was the first mechanical calculator, which used divided differences. Each machine could do just one thing: if you wanted a machine to fulfil a new purpose, you needed a new machine.

This isn't the world we live in today! We have *highly general* machines, which can run almost anything. How is this possible?

This massive qualitative step up in what today's computers are able to accomplish relies on the following theorem, which would have been shocking and surprising to anyone who grew up before the advent of personal computing.

Theorem 6.2 (The Software Principle)

There is a register machine U, called the *universal register machine*, such that

$$f_{U,2}(v,u) = \begin{cases} f_{M,k+1}(\overline{w}) & \text{if } v \text{ and } u \text{ are proper} \\ \uparrow & \text{otherwise} \end{cases}$$

Here, we require that v encodes a register machine M with upper register index k, and u encodes a configuration C with register content $\overline{w}$ of length $k+1$ (properness).

Proof: We input u and v, and must compute $f_{M,k+1}(\overline{w})$ where $v = \text{code}(M)$ and u is a code for $\overline{w}$. Take scratch registers n, m, and ℓ and empty them. Register l will be our counter in a count-through argument.

Repeat the following procedure until register n contains the code of q_H:

1. Let t be the content of register ℓ.
2. By our observations in the previous section, we can calculate $C(\#t, M, \overline{w}) = (q, \overline{s})$.
3. Write the code of q into register n, and write s_0 into register m.

When register n contains the code of q_H, output the content of register m and halt. □

So far, we have assumed that machines with more registers have been more powerful. Indeed, this is true to a point: a register machine with only one register cannot do all that much.

But by this theorem, we have seen that there is a *single* ultimately powerful machine U, which has a finite upper register index, but can do the work of *any* machine (including those with many more registers). This is really quite a surprising result!

In some sense, this is a shift of the complexity and computing power of a register machine from the *hardware* (the states and register count of the machine) to the *software* (specified as the code of the machines). Here, u is the hardware and v is the software.

For binary words $v \in \mathbb{B}$, we write

$$\begin{aligned} f_{v,k}(\overline{w}) &= f_{U,2}(v, \text{code}(q_S, \overline{w})) \\ W_v &= \text{dom}(f_{v,1}) \\ T_v &= T_M \text{ where } v = \text{code}(M) \end{aligned}$$

Specifically, W_v is the list of all computably enumerable subsets, indexed by elements of $\mathbb{B}$!

Theorem 6.3 (The *s*-*m*-*n* Theorem)

If $g : \mathbb{B}^{k+1} \dashrightarrow \mathbb{B}$ is a computable function, then there is a total computable function $h : \mathbb{B} \to \mathbb{B}$ such that for all u and $\overline{w}$, we have

$$f_{h(v),k}(\overline{w}) = g(\overline{w}, v)$$

Proof: For every v, the function $g_v(\overline{w}) = g(\overline{w}, v)$ is computable. This is not in itself sufficient: for example $\overline{w} \mapsto (\overline{w}, v)$ is clearly computable for all v, but we need some way to do this uniformly.

If M is the register machine for g, such that $f_{M,k+1} = g$, then the proof of the Subroutine Lemma yields a concrete machine N_v with $f_{N_v,k}(\overline{w}) = f_M(\overline{w}, v) = g(\overline{w}, v)$.

Then $h(v) = \text{code}\, N_v$ gives us the theorem! □

Note: Originally, h was denoted S^m_n, and the name for the theorem stuck. The process of moving a parameter into the index is called *currying*, named for Haskell Curry.

Corollary: We have machines encoded as binary strings: $M \mapsto \text{code}(M) \in \mathbb{B}$, where $f_{w,k} : \mathbb{B}^k \dashrightarrow \mathbb{B}$ is performed by M such that $\text{code}(M) = w$. Also, $W_w = \text{dom}(f_{w,1})$. This means that:

$$\{W_w : w \in \mathbb{B}\} = \{A \subseteq \mathbb{B} : A \text{ is computably enumerable}\}.$$

6.2 Computably Enumerable Sets

We define the sets $\mathbf{K} = \{w : f_{w,1}(w)\downarrow\}$ and $\mathbf{K}_0 = \{(w, v) : f_{w,1}(v)\downarrow\}$. These are the halting sets, related to Alan Turing's famous Halting Problem.

Theorem 6.4 (Computably Enumerable Sets)

$\mathbf{K}$ and $\mathbf{K}_0$ are both computably enumerable but not computable.

Proof: Firstly, $\mathbf{K}_0 = \text{dom}(f_{v,2})$. The operation $w \mapsto (w, w)$ can be performed by a register machine, thus $f(w) = f_{U,2}(w, w)$ is computable (and $\mathbf{K} = \text{dom}(f)$, proving the first part).

Define the function f by

$$f(w) = \begin{cases} \uparrow & w \in \mathbf{K} \\ \mathbf{0} & \text{otherwise} \end{cases}$$

and notice that if $\mathbf{K}$ were computable, then f would be partial computable. Taking d such that $f_{d,1} = f$, we have $f_{d,1}(d)\downarrow \iff d \in \mathbf{K} \iff f(d)\uparrow \iff f_{d,1}(d)\uparrow$, which is a contradiction! □

Note: This proof is an example of Cantor's *diagonal argument*.

We now consider Hofstadter's Limitative Theorems.

Definition 6.5 (Σ_1, Π_1, and Δ_1 Sets)

We say that $X \subseteq \mathbb{B}^n$ is Σ_1 if there is a computable set $Y \subseteq \mathbb{B}^{k+1}$ such that $\overline{w} \in X$ if and only if there is some v with $(\overline{w}, v) \in Y$. We say that X is Π_1 if it is the complement of a set in Σ_1, and we say that X is Δ_1 if it is both Σ_1 and Π_1.

Proposition 6.6 (Computable Sets Δ_1)

Every computable set is Δ_1.

Proof: In fact, since computability is closed under complementation, we may show simply that any such set is Σ_1: if this holds, then its complement is also in Σ_1, and so it is in Π_1 too.

For a fixed X, define $Y = \{(\overline{w}, v) : \overline{w} \in X, v \in \mathbb{B}\}$. Clearly, this is computable, and we trivially meet the Σ_1 condition by choosing any $v \in \mathbb{B}$. □

In fact, we can show a stronger result.

Theorem 6.7 (Computably Enumerable is Σ_1)

X is computably enumerable if and only if it is Σ_1.

Proof: If X is computably enumerable, take $X = \text{dom}(f)$ and let M compute f. Then consider $T_{M,k} = \{(\overline{w}, v) : f_{M,k}(\overline{w}) \text{ halts after } \#v \text{ steps}\}$. We have seen this to be computable, and clearly $\overline{w} \in \text{dom}(f) \iff$ there is a v with $(\overline{w}, u) \in T_{M,k}$. Thus X is Σ_1.

Conversely, take the set Y with $\overline{w} \in X \iff \exists v : (\overline{w}, v) \in Y$. Then Y is computable. Take $h : \mathbb{B}^k \dashrightarrow \mathbb{B}$ to be the minimisation of χ_Y, which is computable:

$$h(\overline{w}) = \begin{cases} \text{the least } v \text{ with } (\overline{w}, v) \in Y & \text{if one exists} \\ \uparrow & \text{otherwise} \end{cases}$$

Clearly, $\text{dom}(h) = \{\overline{w} : \exists v \text{ s.t. } (\overline{w}, v) \in Y\} = X$, so X is the domain of a computable function and thus computably enumerable. □

Proposition 6.8 (Computably Enumerable is Range)

A set X is computably enumerable if there is a computable function g such that $X = \text{range}(g)$.

Proof: If $X = \text{dom}(f)$, let $g(w) = w$ if $f(w) \downarrow$ and $\uparrow$ otherwise. Then $\text{range}(g) = \text{dom}(f) = X$.

Conversely, if $X = \text{range}(g)$, then let M compute g. Observe that $w \in X \iff$ there exist v and u such that $(v, u, w) \in \hat{T}_{M,1}$. This is Σ_1, so X is computably enumerable. □

A lot of results are still open. We do not know exactly what it means for a problem to be solvable, we have unsolved decision problems remaining, and we have not finished categorising our sets.

In general, for non-empty sets $X \subseteq \mathbb{B}$, X is computably enumerable if and only if it is Σ_1, which is equivalent to it being the domain of a computable function, which is in turn equivalent to it being the range of a computable function!

This is a powerful base to build on. We have the set of Σ_1 and Π_1 sets, which are computably enumerable. We also have their overlap (the Δ_1 sets), and we have the set of computable sets, which we know is a subset of the Δ_1 sets.

Now, we find the relationship between Δ_1 sets and computable sets. We know that the latter is a subset of the former, but is there an uncomputable Δ_1 set?

Proposition 6.9 (Δ_1 is Computable)

In fact, X is computable if and only if it is Δ_1.

Proof: We now need only show the reverse direction. If $X \subseteq \mathbb{B}$ is Δ_1, then both X and $\mathbb{B} \setminus X$ are Σ_1. Let M and M' be register machines such that

$$\begin{aligned} \overline{w} \in X &\iff \exists v \text{ s.t. } (\overline{w}, v) \in T_M \\ \overline{w} \notin X &\iff \exists v \text{ s.t. } (\overline{w}, v) \in T_{M'} \end{aligned}$$

Define $Y = \{(\overline{w}, v) : \#v_{(0)} \text{ is even and } (\overline{w}, v_{(1)}) \in T_M \text{ or } \#v_{(0)} \text{ is odd and } (\overline{w}, v_{(1)}) \in T_{M'}\}$.

We use minimisation to get that

$$h(\overline{w}) = \begin{cases} \text{the least } v \text{ such that } (\overline{w}, v) \in Y & \text{if one exists} \\ \uparrow & \text{otherwise} \end{cases}$$

So h is computable. But in fact h is a total function, since either $\overline{w} \in X$ or $\overline{w} \in \mathbb{B} \setminus X$ (so there is a witness for M or M'). We can write X using this total function: it is precisely the set of $\overline{w} \in Y$ such that $\#h(\overline{w})_{(0)}$ is even! This is clearly computable, so we are done. □

Corollary: **K** is computably enumerable (so Σ_1) but not Δ_1, so it is not computable.

Corollary: The computably enumerable sets are not closed under complementation (for example, $\mathbb{B} \setminus \mathbf{K}$ is not computably enumerable).

Corollary: Every type 0 language is computably enumerable.

Proof: If $G = (\{\mathbf{0}, \mathbf{1}\}, V, P, S)$ is a grammar, consider $\Omega = \{\mathbf{0}, \mathbf{1}\} \cup V$ and $\Upsilon = \Omega \cup \{/\}$. Then $T = (\Omega^*/)^*\Omega^*$ is the set of putative derivations, which necessarily lists all the derivations (and more). But this is checkable by a register machine: the set

$$D = \{(\overline{w}, v) : \overline{w} \in T \text{ and } w \text{ codes a } G\text{-derivation starting from } S \text{ and ending in } v\}$$

is computable. The language $\mathcal{L}(G)$ is then equivalent to the set of words v such that there exists some w with $(w, v) \in D$. This is Σ_1 and so computably enumerable. □

Corollary: The converse also holds: every computably enumerable set is a type 0 language.

This seems to form a pattern: loosely, computably enumerable sets are those which can be written down using the existential quantifier $\exists$.

6.3 Closure Properties

We devote this section to proving the remaining results in the following summary table.

	concatenation	union	intersection	complement	difference
regular (type 3)	✓	✓	✓	✓	✓
context-free (type 2)	✓	✓	×	×	×
context-sensitive (type 1)	✓	✓	✓	✓	✓
computable	✓	✓	✓	✓	✓
computably enumerable (type 0)	✓	✓	✓	×	×

Proposition 6.10 (Computability Closure)

The computable sets are closed under union and intersection.

Proof: Consider the characteristic functions:

$$\chi_{A\cap B}(w) = \begin{cases} \mathbf{1} & \text{if } \chi_A(w) = \chi_B(w) = \mathbf{1} \\ \mathbf{0} & \text{otherwise} \end{cases} \qquad \chi_{A\cup B}(w) = \begin{cases} \mathbf{0} & \text{if } \chi_A(w) = \chi_B(w) = \mathbf{0} \\ \mathbf{1} & \text{otherwise} \end{cases}$$

which are clearly computable. □

Proposition 6.11 (Computably Enumerable Closure)

The computably enumerable sets are closed under union and intersection.

Proof: For intersection, we can use the pseudo-characteristic function

$$\psi_{A\cap B}(w) = \begin{cases} \mathbf{1} & \text{if } \psi_A(w) = \psi_B(w) = \mathbf{1} \\ \uparrow & \text{otherwise} \end{cases}$$

but for union, we must parallelise the computation using the zigzag method again, since one of these functions may not halt while the other one does! If

$$w \in A \iff \exists v : (w, v) \in C \qquad \text{and} \qquad w \in B \iff \exists v : (w, v) \in D$$

then $w \in A \cup B$ if and only if there is some $v \in \mathbb{B}$ satisfying $(\#v)_{(0)}$ even and $(w, v(1)) \in C$ or $(\#v)_{(0)}$ odd and $(w, v(1)) \in D$. □

We can show closure under concatenation by a similar argument, while the counterexample **K** shows a lack of closure under complementation (and hence difference).

6.4 The Church-Turing Thesis

So far, we have a fairly good notion of computation and computability, at least for positive results of the form "this is computable". We have shown that a lot of things can be achieved with register machines, which are a reasonable model, but simply showing that register machines cannot do something is not always sufficient to rule it as definitively *not* being computable.

Is it conceivable that there is an alternate formalisation of computation which gives a genuinely different classification of what is computable? If so, we have merely provided a theory of register machines, rather than any general theory of computation!

Remark 6.12 (History of the Church-Turing Thesis)

In 1936, Alan Turing submitted his paper *On Computable Numbers, with an Application to the Entscheidungsproblem*, with a proof of the negation of Hilbert's conjecture that there was a universal algorithm for correctness.

The referee originally declined it, claiming that Alonzo Church had just solved the problem! Instead, he sent Church the paper directly, who realised that Turing's proof was so different that it also needed publishing. He invited Turing to Princeton to compare proofs.

Turing had proved these results using Turing Machines (5.2), which are very similar to register machines. Church had used his recursive calculus, which was totally distinct. It was therefore surprising that they had found the same categorisation of computable sets and functions!

Theorem 6.13 (Church-Turing Thesis)

If $f : \mathbb{B}^n \dashrightarrow \mathbb{B}$, then the following are equivalent:

1. f is computable as defined by consideration of register machines
2. f is primitive recursive (in sense of the smallest closed class of functions)
3. f can be performed by a Turing machine.

Furthermore, these mentioned equivalent formal concepts of computability are entirely faithful to the informal notions. *Any* reasonable attempt to describe the concept of computability will lead to a formal notion that is equivalent to the ones we have described.

Of course, this is not a formal mathematical statement or a theorem in the truest sense: it is a statement about *us*, and what we consider to be reasonable. There are, of course, unconventional models of computation (such as quantum computation). But with any traditional set of basic computation steps, we in fact get the same notion of computability every time. No non-contrived counterexample has been provided thus far.

This is not to say the notions of *computation* are at all similar, simply *computability* (which in some way is a more elegant statement).

With the Church-Turing Thesis, we now know what it means for a problem to be "algorithmically solvable", an idea which was fuzzy until now!

Take some encoding of grammars $v \mapsto G_v$ such that $\{G_v : v \in \mathbb{B}\}$. We can write:

1. the word problem as the set $\mathcal{P}_W = \{(w, v) : w \in \mathcal{L}(G_v)\}$
2. the emptiness problem as the set $\mathcal{P}_E = \{v : \mathcal{L}(G_v) \text{ is empty}\}$
3. the equivalence problem as the set $\mathcal{P}_Q = \{(w, v) : \mathcal{L}(G_w) = \mathcal{L}(G_v)\}$

and say that these problems being solvable is equivalent to these sets being computable.

In fact, the word problem is not computable, since computably enumerable sets are type 0, and these are not all the computable sets. If $\mathcal{P}_W$ were computable, then

$$f(w) = \begin{cases} \uparrow & (w, w) \in \mathcal{P}_W \\ \mathbf{0} & \text{otherwise} \end{cases}$$

would be computable, and so we could find a d such that the domain of f was $\mathcal{L}(G_d)$. But then

$$d \in \mathcal{L}(G_d) \iff d \in \mathrm{dom}(f) \iff (d, d) \notin \mathcal{P}_W \iff d \notin \mathcal{L}(G_d)$$

which is clearly a contradiction.

This brings us to the following summary of results for the three main problems:

	word	emptiness	equivalence
regular (type 3)	✓	✓	✓
context-free (type 2)	✓	✓	×
context-sensitive (type 1)	✓	×	×
computably enumerable (type 0)	×	?	?

In fact, neither the emptiness nor equivalence problems are solvable for type 0 grammars. We will show these results over the following sections.

6.5 Reductions and Solvability

Definition 6.14 (Partial Order and Preorder)

We define a relation $\leqslant$ on a set X as being a *partial order* if it is reflexive, transitive, and antisymmetric. A partial *preorder* drops the antisymmetry condition. However, we can define $\equiv$ to be the relation where $x \equiv y$ if $x \leqslant y$ and $y \leqslant x$. Then $(X/\equiv, \leqslant)$ is a partial order! These equivalence classes are called $\equiv$-degrees.

A total computable function $f : \mathbb{B} \to \mathbb{B}$ is called a reduction of L to L' if we have

$$w \in L \iff f(w) \in L' \text{ for all } w.$$

We then say that L is many-one reducible to L', or $L \leqslant_m L'$. Note that we do not enforce f being injective or surjective. Clearly, this is a partial preorder! Its $\equiv_m$-degrees are called *degrees of unsolvability*.

Proposition 6.15 (Reductions preserve Computability)

If $L \leqslant_m L'$ and L' is computable, then so is L.

Similarly, if $L \leqslant_m L'$ and L' is computably enumerable, then so is L.

Proof: If f is the witness showing that $L \leqslant_m L'$, then

$$\chi_L = \chi_{L'} \circ f \qquad \text{and} \qquad \psi_L = \psi_{L'} \circ f$$

which shows the result. □

Corollary: We see that $L \leqslant_m L'$ implies that $(\mathbb{B} \setminus L) \leqslant_m (\mathbb{B} \setminus L')$.

Similarly, $\mathbf{K} \leqslant_m L$ is sufficient to show that L is not computable, and $(\mathbb{B} \setminus \mathbf{K}) \leqslant_m L$ is sufficient to show that L is not even computably enumerable.

As the coding functions are computable, the emptiness problem is equivalent to

$$\mathcal{P}_E = \{w : \mathcal{L}(G_w) = \varnothing\} \equiv_m \{w : W_w = \varnothing\}$$

and the equivalence problem is equivalent to

$$\mathcal{P}_Q = \{(w, v) : \mathcal{L}(G_w) = \mathcal{L}(G_v)\} \equiv_m \{(w, v) : W_w = W_v\}.$$

where $W_w = \mathrm{dom}(f_{w,1})$. We must show that these are not computable.

If $\mathcal{C}$ is a class of languages, we say that a problem X is $\mathcal{C}$-hard if

$$L \in \mathcal{C} \implies L \leqslant_m X \quad (\text{“at least as complicated as every element in } \mathcal{C}\text{”})$$

and we say that X is $\mathcal{C}$-complete if X is $\mathcal{C}$-hard and $X \in \mathcal{C}$ (“the most complicated element of $\mathcal{C}$”). The most common class for which this becomes a relevant problem is NP, the class of problems which can be solved by an algorithm not bounded by polynomial time.

Proposition 6.16 (Δ_1 Completeness)

If $L \neq \varnothing, \mathbb{B}$ is computable, then L is Δ_1-complete.

Proof: Note that Δ_1 is the class of computable sets (solvable problems). By assumption, take $v \in L$ and $u \notin L$, with

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

which is clearly total and computable, and witnesses $X \leqslant_m L$. □

We now use the *s-m-n* Theorem (6.3) for the first time to show that $\mathbf{K}$ is Σ_1-complete.

Theorem 6.17 ($\mathbf{K}$ is Σ_1-complete)

The "halting set" $\mathbf{K}$ is Σ_1-complete (with Σ_1 the class of computably enumerable problems).

Proof: Choose f with $X = \text{dom}(f)$, and define $g : \mathbb{B}^2 \to \mathbb{B}$ mapping $(w, v) \mapsto f(w)$. This is computable, so it extends to a total computable function h satisfying

$$f_{h(w),1}(v) = g(w, v) = f(w)$$

We now claim that h reduces X to $\mathbf{K}$. Notice that

$$w \in X \iff w \in \text{dom}(f) \iff f_{h(w),1} \text{ is defined everywhere.}$$

Since this is defined everywhere, in particular it is defined at $h(w)$ itself:

$$f_{h(w),1}(h_w) \downarrow \iff h(w) \in \mathbf{K}.$$

The same pattern shows that $w \notin X \iff h(w) \notin \mathbf{K}$. Therefore X is reducible to the halting problem, and is therefore not solvable. Thus $\mathbf{K}$ is Σ_1-complete. $\square$

The bottom class in the partial hierarchy of unsolvability is therefore the class of solvable problems Δ_1 (except the trivial "classes" of $\varnothing$ and $\mathbb{B}$). Above them in the hierarchy are Σ_1 and Π_1 classes, which include $\mathbf{K}$ and $\mathbb{B} \setminus \mathbf{K}$ respectively as complete members of the class.

We may ask if this is in fact the complete hierarchy. In fact, it cannot be!

Definition 6.18 (Turing Join)

If $X, Y \subseteq \mathbb{B}$, define the *Turing join* of X and Y by

$$X \oplus Y = \mathbf{0}X \cup \mathbf{1}Y.$$

Note that $X \leqslant_m X \oplus Y$ via $w \mapsto \mathbf{0}w$, and $Y \leqslant_m X \oplus Y$ via $w \mapsto \mathbf{1}w$.

Thus the sets $\mathbf{K}$ and $\mathbb{B} \setminus \mathbf{K}$ are both reducible to their Turing join. As they are not equivalent, they are strictly less hard than $\mathbf{K} \oplus (\mathbb{B} \setminus \mathbf{K})$, which must therefore sit above them on the hierarchy.

6.6 Index Sets and Rice's Theorem

We now consider a special type of set of words.

Definition 6.19 (Index Set)

A set $I \subseteq \mathbb{B}$ is called an *index set* if for all weakly equivalent w and v in $\mathbb{B}$ (those words with $W_w = W_v$), we have $w \in I \iff v \in I$. Equivalently, index sets are sets which are closed under weak equivalence.

Example 6.20 (Index Sets)

The following sets are index sets:

1. $\varnothing$ and $\mathbb{B}$ (the *trivial* index sets)
2. $\mathbf{Emp} = \{w : W_w = \varnothing\}$ is a nontrivial index set.
3. $\mathbf{Fin} = \{w : W_w \text{ is finite}\}$ and $\mathbf{Inf} = \{w : W_w \text{ is infinite}\}$ are nontrivial index sets.

This brings us to one of the most central theorems of theoretical computer science, which centres on these index sets.

Theorem 6.21 (Rice's Theorem)

No nontrivial index set is computable.

Proof: Fix $w \in \mathbb{B}$ and consider the function

$$g_w(u, v) = \begin{cases} f_{w,1}(v) & u \in \mathbf{K} \\ \uparrow & \text{otherwise} \end{cases}$$

This is not generally allowed by the Case Distinction Lemma (5.11), as the question $u \in \mathbf{K}$ is uncomputable (indeed, our canonical example of *the* uncomputable question!) which means that we cannot go to the second case. However, it is allowed in this specific case: we either get an affirmative answer, or we accidentally never halt, satisfying our alternate case anyway!

Then g_w is computable: check if $w \in \mathbf{K}$, and return $f_{w,1}(v)$ if so and never halt if not. The s-m-n Theorem (6.3) ensures that there is a total computable h_w with $f_{h_w(u),1}(v) = g_w(u, v)$.

1. If $u \in \mathbf{K}$, then $f_{h_w(u),1} = f_{w,1}$, so $W_{h_w(u)} = W_w$.
2. If $u \notin \mathbf{K}$, then $f_{h_w(u),1}$ is nowhere defined, so $W_{h_w(u)} = \varnothing$.

Let I be a nontrivial index set. Fix some e such that $W_e = \varnothing$. Then either $e \in I$ or $e \notin I$.

1. Suppose $e \in I$. Then take $w \notin I$, and consider g_w and h_w. We claim that h_w is a reduction witnessing that $\mathbb{B} \setminus \mathbf{K} \leqslant_m I$.
   - If $u \notin \mathbf{K}$, then $W_{h_w(u)} = \varnothing$, so $h_w(u)$ is weakly equivalent to e. Since I is closed under weak equivalence, $h_w(v) \in I$.
   - If $u \in \mathbf{K}$, then $W_{h_w(u)} = W_w$, so $h_w(u)$ is weakly equivalent to w. Since I is closed under weak equivalence, $h_w(v) \notin I$.
2. Now suppose that $e \notin I$. Take $w \in I$, and consider g_w and h_w again. Now, we claim that h_w witnesses $\mathbf{K} \leqslant_m I$, which we can show in essentially the same way.

So in the first case, $u \in (\mathbb{B} \setminus \mathbf{K}) \iff h_w(u) \in I$, so $\mathbb{B} \setminus \mathbf{K} \leqslant_m I$. In the second case, $u \in \mathbf{K} \iff h_w(u) \in I$, so $\mathbf{K} \leqslant_m I$. Either way, *some* uncomputable set reduces to I, which was an arbitrary nontrivial index set. Thus any nontrivial index set is uncomputable, as we desired! □

Corollary: **Emp** is uncomputable: the emptiness problem for type 0 grammars is unsolvable.

In fact, the proof shows quite a lot more!

We have specific cases: $e \in I \implies \mathbb{B} \setminus \mathbf{K} \leqslant_m I$, or in particular I is not computably enumerable. $e \in I \implies \mathbf{K} \leqslant_m I$. By closure under weak equivalence, we may check this for any e with $W_e = \varnothing$. In particular, $\mathbb{B} \setminus \mathbf{K} \leqslant_m \mathbf{Emp}, \mathbf{Fin}$. Similarly, $\mathbf{K} \leqslant_m \mathbf{Inf}$.

We can further show that **Emp** is Π_1-complete. However **Fin** and **Inf** are neither Σ_1 nor Π_1, since the Turing Join (6.18) of the two $\mathbf{K} \oplus (\mathbb{B} \setminus \mathbf{K})$ reduces to them.

This is quite a significant proof! Index sets are essentially semantic properties of programs, and we have shown that no such set is decidable. This means that there is no general algorithm to decide semantic properties of programs!

The final problem we come to is the equivalence problem for type 0 grammars.

Corollary: The set $\mathbf{Eq} = \{(u, v) : W_u = W_v\}$ is not computable.

Proof: Suppose e is such that $W_e = \varnothing$, and consider the map $w \mapsto (w, e)$. This is an operation that can be performed by a register machine, which means its pseudo-characteristic function $\chi'(w) = \chi(w, e)$ is the characteristic function of **Eq**. But then this is obviously the characteristic function of the emptiness problem, which we saw previously was uncomputable. □

7 A Recap of Computation

At the start of this course, we set out to define *computation*. Instead of defining a particular programming language, the goal was to gain a deep understanding of what problems were genuinely undecidable by algorithms in general. We began with the motivation of Hilbert's tenth problem, which was to find integer solutions to polynomials using an algorithm: soon shown to be unsolvable.

The first notion of computability came from languages. Languages like English are composed of syntax and semantics: logical rules to follow and coherence in meaning. We considered syntax, in the form of grammars. These are composed of alphabets: strings of symbols which we might call words, but could of course represent anything. Rewrite systems consist of rules which can turn strings into other strings, and we form words by rewriting strings containing variables into those with just the letters of the alphabet.

We considered the hierarchy of languages. The most basic set we considered was the set of all possible languages generated by some grammar (type 0). Then, we imposed progressively more restrictions on these languages: first every rule must turn a string into a longer string (type 1, or noncontracting), then rules can only turn variables into other variables or letters (type 2, or context-free), then finally rules can only append letters to the end of the word in predetermined ways (type 3, or regular).

We defined some problems relating to these systems. The word problem involved verifying whether a set of rules could eventually generate a given word. The emptiness problem involved verifying whether a set of rules could eventually generate any word at all. The equivalence problem involved verifying whether two sets of rules generate the same set of words.

Next, we found another notion of computability: automata. Automata have states, and they can transition between these states in predetermined ways as they read the letters of a word in order. Some states in automata are marked as accept states, and the words which make an automaton enter such a state when they are read are in the language of the automaton. We expanded this idea to non-deterministic automata, then showed that this didn't expand what we could do.

In fact, we showed that the idea of regular languages and the idea of automata were precisely equivalent: any regular language could be represented as an automaton, and any automaton defined a regular language. This allowed us to prove several important results, and solve all three of the problems we defined for the class of regular languages. We found neat ways of describing regular languages using regular expressions.

We then considered context-free languages: a more expansive class. We showed that Chomsky Normal Form, a far more restrictive way of describing context-free languages, was in fact sufficient to represent all of them, and used this fact to prove several results about the class of language, demonstrate their closure properties, and solve the first two of our three problems.

After this, we built up a new model of computation from scratch. We defined register machines, which are basic computers. They have a few registers, which hold binary strings. They also have a list of program lines, which perform basic instructions (checking/removing the last digit of a register, or adding a digit to a register, before going to a different instruction). We proved several results about these machines in order to simplify our notion of computation, before tying them in to a useful notion of computation which involved answering questions and performing operations.

We then showed that these machines were surprisingly capable, given their limitations. By proving that machines could perform a lot of specific functions, and that they could chain possible functions together in notable ways like subroutines and case distinction, we showed that register machines encoded a general idea of computation.

We continued to expand the power of computers: many mathematical operations were computable, as are all bounded search problems. Defining the class of primitive recursive functions, we found that these were also all computable. Most importantly, we showed that the process of simulating arbitrary other computers was itself computable.

This brought us to the idea of software, which was a powerful general principle. We defined a method which described register machines in perfect detail using a binary code. Rather than relying on a register machine with hardware configured to perform certain tasks, we showed that there exists a singular register machine which can perform every task, merely given the binary specification of another register machine which performed that task.

We then showed that our idea of computation was not universal. In fact, there are sets and functions which can never be computed, not even in principle. The canonical example is given by Alan Turing's Halting Problem, which is the problem of verifying if a given machine or piece of code will ever finish running when it is run on a particular input.

This brought us to the most famous result in computer science: the Church-Turing thesis. Not only are our notions of computability from grammars, register machines, and primitive recursive functions identical, in fact *every* notion of computability is the same.

We defined the classes of solvable and unsolvable problems, and defined the idea of reductions, which show definitively that certain problems are at least as hard as others. Solving just a few problems would yield solutions to countless others. We showed that there was in fact a hierarchy of unsolvability, and that certain known sets were above other sets.

Finally, we defined index sets, which are sets of programs which share semantic properties and halt on the exact same inputs. We showed that all of these sets were in fact uncomputable, which means that no program can ever determine the truth value of nontrivial semantic properties for an arbitrary piece of software it is given.

This gave us a solution to all three of our decision problems for our classes of language!

	word	emptiness	equivalence
regular (type 3)	✓	✓	✓
context-free (type 2)	✓	✓	×
context-sensitive (type 1)	✓	×	×
computably enumerable (type 0)	×	×	×