Regular languages have no memory.
Context free languages have a stack, see pushdown automaton.

Automata & Formal Languages
Michaelmas Term 2023
Part II of the Mathematical Tripos
University of Cambridge
Prof. Dr. B. Löwe

Automata & Formal Languages

Contents

1 Formal Languages & Grammars

1.1 Notation & preliminaries

We fix any set X. If n is a natural number (note that we include 0 in the natural numbers), then X^n is the set of n-tuples of elements of X; we call these objects *X-strings of length n* (usually denoted by letters such as α, β, γ, σ, and τ). In the usual set-theoretic representation, $n = \{0, 1, ..., n-1\}$ and a string of length n is a function from the set n into X. Note that X^0 only contains the empty sequence which we shall denote by ε. We write

$$X^* := \bigcup_{n \in \mathbb{N}} X^n$$

for the set of all X-strings[1] and write $|\alpha| = n = \{0, ..., n-1\} = \mathrm{dom}(\alpha)$ if $\alpha \in X^n$; the number $|\alpha|$ is called the *length of* α. The set $X^+ := X^* \backslash \{\varepsilon\}$ is the set of non-empty X-strings. Since strings are functions, we can use the usual notation for function restriction to denote their initial segments, i.e., if $\alpha \in X^n$ and $k \leq n$, then $\alpha{\restriction}k$ is the unique initial segment of α of length k.

If $\alpha, \beta \in X^*$, we can concatenate them in the usual way and write $\alpha\beta$ for the concatenated string. If α has length n and β has length m, then $\alpha\beta$ has length $n + m$:

$$\alpha\beta(k) := \begin{cases} \alpha(k) & \text{if } k < n \text{ and} \\ \beta(\ell) & \text{if } k = n + \ell \text{ and } \ell < m. \end{cases}$$

If $x \in X$, we use the notation x^n for the string of length n consisting only of the symbol x. Similarly, if $\alpha \in X^*$, we write α^n for the concatenation of n copies of the string α (formally, we can define this by recursion as $\alpha^0 := \varepsilon$, $\alpha^{n+1} := \alpha^n\alpha$). We often (slightly incorrectly) confuse $x \in X$ with the string of length 1 consisting of the element x. So, if we write αx, we mean the string α with an extra element x appended at the end; if we write $x\alpha$, we mean the string α prefixed by an element x. If $Y, Z \subseteq X^*$, we write $YZ := \{\alpha\beta\,;\, \alpha \in Y \text{ and } \beta \in Z\}$; if $Y = \{\alpha\}$, we abbreviate this to αZ and if $Z = \{\beta\}$, we write $Y\beta$.

Given any function $f : X \to Y$, we can use it to define a function $\widehat{f} : X^* \to Y^*$ pointwise by $\widehat{f}(\alpha)(k) := f(\alpha(k))$ (i.e., a sequence of length n will be mapped to a sequence of length n). We often re-use the notation f for the extended function if no confusion is possible, i.e., we write $f : X^* \to Y^*$ instead of $\widehat{f}$.

[1] The notation X^* is sometimes called the *Kleene star* after the American logician Stephen Cole Kleene (1909-1994); more about this in § 2.6.

As usual, we say that a set X is *countable* if it is empty or there is a surjection from $\mathbb{N}$ onto X,[2] it is *infinite* if there is an injection from $\mathbb{N}$ into X. A set is called *uncountable* if it is not countable.[3] Clearly, if $X \subseteq Y$, X is countable and Y is uncountable, they are not the same, so $Y \backslash X \neq \varnothing$. Moreover, from the Part IA course *Numbers & Sets*, we know that the union of two countable sets is countable, therefore $Y \backslash X$ cannot even be countable. The following results are simple applications of techniques learned in *Numbers & Sets*:

Lemma 1.1. If X and Y are countable, then so is $X \times Y$.

Proof. First of all, if either X or Y is empty, then $X \times Y$ is empty, so we can assume that they are both non-empty and pick surjections $\pi_X \colon \mathbb{N} \to X$ and $\pi_Y \colon \mathbb{N} \to Y$. Remember from *Numbers & Sets* that there is a bijection $z \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, e.g., Cantor's *zigzag bijection*

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

which will feature prominently in § 4.5. Given any $n \in \mathbb{N}$, find i and j such that $n = z(i, j)$ and define $f(n) := (\pi_X(i), \pi_Y(j))$. It is easy to check that this is a surjection onto $X \times Y$. Q.E.D.

Proposition 1.2. If $X \neq \varnothing$ is countable, then X^* is infinite and countable.

Proof. Let $x \in X$ (this exists since X is non-empty). The map $n \mapsto x^n$ is an injection from $\mathbb{N}$ into X^*, so X^* is infinite.

Clearly, X^0 is countable (since it has only one element). By induction, using Lemma 1.1, we deduce that X^n is countable for every n. But then $X^* = \bigcup_{n \in \mathbb{N}} X^n$ is countable as a countable union of countable sets (by *Numbers & Sets*). Q.E.D.

Proposition 1.3 (Cantor's Theorem)**.** If X is infinite, then the power set of X, i.e., the set of all subsets of X, denoted by $\wp(X)$, is uncountable.

Proof. Let $i : \mathbb{N} \to X$ be an injection. Suppose that $\pi : \mathbb{N} \to \wp(X)$ is a function. We shall show that it is not a surjection. Define $D := \{i(n) \,;\, i(n) \notin \pi(n)\}$ and claim that D is not in the range of π. Assume otherwise, then there is some $d \in \mathbb{N}$ such that $D = \pi(d)$. But then $i(d) \in D = \pi(d)$ if and only if $i(d) \notin \pi(d)$. Contradiction! Q.E.D.

Proposition 1.4. If X is countable, then the set of finite subsets of X is countable.

Proof. Without loss of generality, $X \neq \varnothing$. Let $\pi : \mathbb{N} \to X$ be the surjection witnessing countability of X. By Proposition 1.2 we know that X^* is countable, so it's enough to show that there is a surjection from X^* to the set of finite subsets of X. If $\alpha \in X^*$, let $f(\alpha) := \operatorname{ran}(\alpha)$

[2] If $X \neq \varnothing$, there is an injection from X into $\mathbb{N}$ if and only if there is a surjection from $\mathbb{N}$ onto X.

[3] This definition is not the standard definition of "infinite" that will be presented in Part II *Logic & Set Theory*, but it is equivalent to it under the assumption of the axiom of choice.

be the set of all elements of X occurring in α. If $F \subseteq X$ is any finite set and $x \in F$, find the minimal n_x such that $\pi(n_x) = x$. Then $\{n_x\,;\, x \in F\}$ is a finite set of natural numbers of the same size as F. Order these by size, e.g., $n_0 < n_1 < ... < n_k$. Define α by $\alpha(i) := \pi(n_i)$. Then $f(\alpha) = F$. Q.E.D.

1.2 Rewrite systems

Suppose that Ω is a non-empty finite set whose elements are called *symbols*. Consider the set Ω^* of Ω-strings and the set $\Omega^+ := \Omega^* \backslash \{\varepsilon\}$ of non-empty Ω-strings. An element of $\Omega^+ \times \Omega^*$ is called a *production rule* or *rewrite rule* over Ω. To improve readability, we write $\alpha \to \beta$ for (α, β). The informal interpretation of such a production rule is: "whenever a string contains α as a substring, it can be rewritten by β".

Definition 1.5. A pair (Ω, P) is called a *rewrite system* if Ω is a non-empty finite set and P is a finite set of rewrite rules over Ω.

Proposition 1.6. If Ω is a non-empty finite set, then there are countably many rewrite systems on Ω.

Proof. By Proposition 1.2, Ω^* is countable and so is $\Omega^+ \times \Omega^* \subseteq \Omega^* \times \Omega^*$ by Lemma 1.1. The set P is a finite subset of $\Omega^+ \times \Omega^*$; thus, by Proposition 1.4, there are only countably many choices for P. Q.E.D.

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

$$\sigma \xrightarrow{R}_1 \tau$$

if there are $\alpha, \beta, \gamma, \delta \in \Omega^*$ such that $\sigma = \alpha\beta\gamma$, $\tau = \alpha\delta\gamma$, and $\beta \to \delta \in P$ and say that R *produces* τ *from* σ *in one step* or R *rewrites* σ *into* τ *in one step*. The relation $\xrightarrow{R}$ is defined as the transitive and reflexive closure of $\xrightarrow{R}_1$, i.e., $\sigma \xrightarrow{R} \tau$ if and only if either $\sigma = \tau$ or there are σ_0, ..., σ_n such that $\sigma_0 = \sigma$, $\sigma_n = \tau$, and for each $0 < k < n-1$, we have $\sigma_k \xrightarrow{R}_1 \sigma_{k+1}$. We say that R *produces* τ *from* σ or R *rewrites* σ *into* τ.

If $\sigma \xrightarrow{R} \tau$, we call a sequence $(\sigma_0, ..., \sigma_n)$ as in the definition an R-*derivation* of τ from σ using R of length n.[4] Because of this, we also say τ *can be derived from* σ *in* R. Note that this sequence need not be uniquely determined (cf. Example Sheet # 1).

If R is a rewrite system and α is a string, we write $\mathcal{D}(R, \alpha) := \{\beta\,;\, \alpha \xrightarrow{R} \beta\}$ for the set of strings that can be derived from α in R.

1.3 Relation to actual languages

If we think of Ω as a set of basic linguistic construction units, i.e., letters or words or sentences, then we can think of larger, composite linguistic entities as elements of Ω^*: words are finite

[4] Note that even though the derivation is a sequence of length $n+1$, we call it a *derivation of length* n. This is because we are counting the number of rewrite rules applied in the derivation. The sequence consisting of σ_0 is a derivation of length zero since zero rewrite rules have been applied.

sequences of letters, sentences are finite sequences of words, and texts are finite sequences of sentences. Not every finite sequence of letters is a word, not every finite sequence of words is a grammatical sentence, and not every sequence of sentences form an intelligible text. Thus, the task is to describe which subset $L \subseteq \Omega^*$ consists of the *wellformed* sequences that we shall consider acceptable in our language.

In practice, an actual human vernacular language is a *finite* subset of Ω^*. E.g., in the case of words constructed from letters, the language is usually defined by a *dictionary*: a finite list of all words existing in this language. However, Noam Chomsky observed that one of the most fundamental features of language is what he called *(linguistic) recursion*:

> [The] arbitrary decree that there is a finite upper limit to sentence length in English ... would serve no useful purpose. ... The point is that there are processes of sentence formation that this elementary model for language is intrinsically incapable of handling. ... In general, the assumption that languages are infinite is made for the purpose of simplifying the description. If a grammar has no recursive steps, ... it will be prohibitively complex. If it does have recursive devices, it will produce infinitely many sentences.[5]

E.g., the process of taking subordinate clauses is a *productive* feature of language. In principle, no matter how complex a sentence is, one can increase its complexity by prefixing it with "X observes that". So, we form an infinite sequence of grammatical sentences

> B likes A.
> C believes that B likes A.
> D reports that C believes that B likes A.
> E observes that D reports that C believes that B likes A.
> etc.

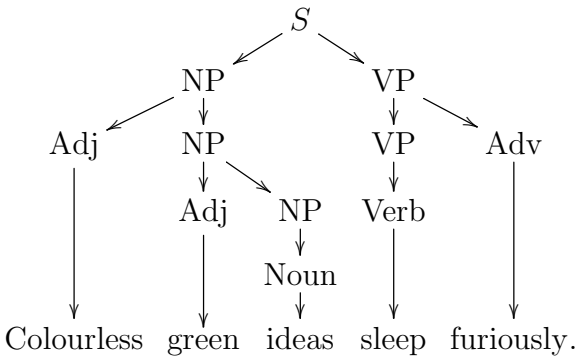
Clearly, at some point, these sentences become too complex to be used in practice as a means of human communication, but there is no non-arbitrary maximum depth: if you can understand a sentence with n such nestings, you will be able to understand a sentence with $n+1$ such nestings. Chomsky's proposal is therefore to embrace that the structure of languages is governed by recursive rules and they are therefore best represented by *infinite* sets of sequences. Recursive rules of such a *generative grammar* could be something like

$$\begin{aligned} S &\to \mathrm{NP}\,\mathrm{VP}, \\ \mathrm{NP} &\to \mathrm{Adj}\,\mathrm{NP}, \\ \mathrm{NP} &\to \mathrm{Noun}, \\ \mathrm{VP} &\to \mathrm{VP}\,\mathrm{Adv}, \\ \mathrm{VP} &\to \mathrm{Verb}, \end{aligned} \tag{*}$$

where NP stands for "noun phrase" and VP for "verb phrase" together with a dictionary list that allows to exchange Noun, Verb, Adj, or Adv with every noun, verb, adjective, or adverb

[5] Chomsky, N. (1956). Three models for the description of language. IRE Transactions of Information Theory, 2(3), 113–124; pp. 115f.

in the dictionary, respectively. This allows the *derivation* of Chomsky's famous *grammatical, but nonsensical* sentence "Colourless green ideas sleep furiously".



In contrast, Chomsky's second example of a string of words "Furiously sleep ideas green colourless" is not only nonsensical, but cannot be produced from the grammar described above. It is therefore *ungrammatical*. Using our definitions from the next section, we shall be able to prove its ungrammaticality mathematically (Example 1.15).

1.4 Grammars

We shall look at specific rewrite systems that we shall call *grammars*. Our symbols in Ω come in two types: *terminal symbols*, also called *letters*, and *nonterminal symbols*, also called *variables*. We write $\Sigma \subseteq \Omega$ for the set of letters, also called the *alphabet*, and $V \subseteq \Omega$ for the set of variables. We assume that Σ and V are both non-empty and disjoint. By convention, we use a, b, c for terminals and A, B, C for nonterminals.

A Σ-string, i.e., an element of Σ^* is called a *word over* Σ. We usually use letters such as u, v, and w to refer to words and use the symbol $\mathbb{W} := \Sigma^*$ for the set of words and $\mathbb{W}^+ := \mathbb{W} \backslash \{\varepsilon\}$ for the set of non-empty words. Any subset $L \subseteq \mathbb{W}$ is called a *language over* Σ. The set of languages is just the power set of $\mathbb{W}$, so, by Proposition 1.3, we know that there are uncountably many languages.

Definition 1.7. A *grammar over* Σ is a tuple (Σ, V, P, S) where Σ and V are non-empty and disjoint with $\Omega := \Sigma \cup V$ and we have that $S \in V$ and that (Ω, P) is a rewrite system. We call S the *start symbol*.

Since grammars are special rewrite systems, we can use the notation for rewrite systems for our grammars, i.e., if $G = (\Sigma, V, P, S)$ is a grammar and $R := (\Omega, P)$, then $\mathcal{D}(G, \alpha) := \mathcal{D}(R, \alpha)$, $\alpha \xrightarrow{G}_1 \beta$ if and only if $\alpha \xrightarrow{R}_1 \beta$, and $\alpha \xrightarrow{G} \beta$ if and only if $\alpha \xrightarrow{R} \beta$. We define

$$\mathcal{L}(G) := \{w \in \mathbb{W} \,;\, S \xrightarrow{G} w\} = \mathbb{W} \cap \mathcal{D}(G, S)$$

and call this the *language generated by* G. These are all the words that can be derived from the start symbol. The following are very basic properties whose proofs give us a general idea how to work with grammars. Let $G = (\Sigma, V, P, S)$ be a grammar.

Example 1.8. If there is no production of the form $S \to \alpha \in P$, then $\mathcal{D}(G, S) = \{S\}$ and $\mathcal{L}(G) = \varnothing$.

[Clearly, there is a unique derivation of length zero and it derives S, so $S \in \mathcal{D}(G, S)$ for any grammar G. By our assumption, there are no G-derivations from S of length one (and therefore not of any greater length), thus $\mathcal{D}(G, S) = \{S\}$ and hence $\mathcal{L}(G) = \mathcal{D}(G, S) \cap \mathbb{W} = \varnothing$.]

Example 1.9. If there is no production of the form $\alpha \to w \in P$ where $w \in \mathbb{W}$, then $\mathcal{L}(G) = \varnothing$.

[We prove by induction on the length of derivation that the final string is not a word. Clearly, the unique derivation of length zero produces S which is not a word. If $S \xrightarrow{G} \alpha$ by any derivation of greater length, the final step in the derivation is of the form $\beta \xrightarrow{G}_1 \alpha$, in particular, one of the rules or P is applied in the rewriting of β to α. But the right-hand side of that rule contains a variable, so α contains a variable and thus is not a word.]

Example 1.10. Let $\Sigma = \{a\}$, $V = \{S\}$, $P_0 := \{S \to aaS, S \to a\}$, and $G_0 := (\Sigma, V, P_0, S)$. Then $\mathcal{L}(G_0)$ is the set of all odd-length words consisting of the letter a.

[Let's prove this in detail: first of all, we notice that each rewrite step either keeps the number of symbols the same or increases it by two. We prove by induction on the length of the derivation, that every string produced by G from S has odd length: the unique string with a derivation of length zero is S which has odd length; of all strings produced by derivations of length n have odd length, say, length $2k+1$, then a string with a derivation of length $n+1$ has either length $2k+1$ or $(2k+1)+2$, thus odd length.

In order to see that the unique word a^{2n+1} of length $2n+1$ can be produced, we give provide a concrete derivation: we apply the production rule $S \to aaS$ to the start symbol n times to obtain $a^{2n}S$ and finally apply $S \to a$ to remove the nonterminal and acquire the desired word a^{2n+1}.]

A proof that $\alpha \notin \mathcal{D}(G, S)$ is always an induction proof of an invariant property shared by all elts of $\mathcal{D}(G,S)$ but not α.

An analysis of the argument in Example 1.10 shows that if in a grammar all production rules preserve oddness of length and we can provide a derivation of a^{2n+1}, then the grammar will produce the same language. E.g., $G_i = (\{a\}, \{S\}, P_i, S)$ with

$$
\begin{aligned}
P_1 &:= \{S \to aSa, S \to a\},\\
P_2 &:= \{S \to Saa, S \to a\},\\
P_3 &:= \{S \to aaS, S \to aaSaa, S \to a\},\\
P_4 &:= \{S \to aaS, S \to Saa, S \to aSa, S \to a\},\\
P_5 &:= \{S \to aaS, aSa \to aaa, S \to a\},\\
P_6 &:= \{S \to aaS, aaS \to aSa, S \to a\}, \text{ or}\\
P_7 &:= \{S \to aaS, aaS \to a, S \to a\}, \text{ etc.}
\end{aligned}
$$

Thus, different grammars can produce the same language. We say that two grammars G and G' are called *equivalent* if $\mathcal{L}(G) = \mathcal{L}(G')$. But they are not entirely interchangeable as $\mathcal{D}(G, S) \neq \mathcal{D}(G', S)$ can hold.

Definition 1.11. Let Σ be an alphabet and let $G = (\Sigma, V, P, S)$ and $G' = (\Sigma, V', P', S')$ be two grammars over Σ. Let $f : \Omega \to \Omega'$ be any function and extend it by recursion to Ω^*. We say that f is an *isomorphism between G and G'* if

(i) it is the identity on Σ, i.e., $f(a) = a$ for all $a \in \Sigma$; By recursion identity on Σ^*

(ii) $f(S) = S'$;

(iii) the restriction $f|_V$ is a bijection between V and V'; and

(iv) for each $\alpha, \beta \in \Omega^*$, we have $\alpha \to \beta \in P$ if and only if $f(\alpha) \to f(\beta) \in P'$.

If there is an isomorphism between G and G', we also say that the two grammars are *isomorphic*.

Proposition 1.12. Isomorphic grammars are equivalent.

Proof. If f is an isomorphism between G and G', then f^{-1} is an isomorphism between G' and G. Thus, by symmetry, it's enough to show that if f is such an isomorphism, then $\mathcal{L}(G) \subseteq \mathcal{L}(G')$. We can consider the f-image of any G-derivation of w (i.e., apply f to each Ω-string in the derivation to obtain a new sequence of Ω'-strings). By property (ii), it starts with S'; by property (iv), it is a G'-derivation; by property (i), it derives $f(w) = w$. Q.E.D.

Say $\sigma_1, \dots, \sigma_n$ a derivation of w from S, then $f(\sigma_1), \dots, f(\sigma_n)$ a derivation of $f(w)$ from $f(S)$.

Proposition 1.13. If $G = (\Sigma, V, P, S)$ is any grammar and $|V'| = |V|$, then there is a grammar $G' = (\Sigma, V', P', S')$ that is isomorphic to G.

Proof. We first extend the bijection $f : V \to V'$ to a bijection from Ω to Ω' by letting it be the identity on Σ. Then we define $S' := f(S)$ and P' by property (iv). This means that the extension of f to Ω is an isomorphism between G and G'. Q.E.D.

Proposition 1.14. Fix an alphabet Σ. Up to equivalence, there are only countably many grammars over Σ.

Proof. If we fix a finite set V, then by Proposition 1.6, there are only countably many rewrite systems (Ω, P). Thus, the set of grammars over Ω is a finite union of countable sets, hence countable.

Propositions 1.12 & 1.13 imply that the set of all languages produced by a grammar with n variables is independent of the choice of the set of variables. Thus, if we fix a countable set of variables $\{V_i\,;\, i \in \mathbb{N}\}$, every grammar is equivalent to a grammar with $V = \{V_i\,;\, i \leq n\}$ for some n. Therefore the set of all languages produced by a grammar is a countable union of countable sets, thus countable (by *Numbers & Sets*). Q.E.D.

We remark that the proof of Proposition 1.14 gives us a very important tool: whenever we have two grammars and we only care about the languages they produce, we may w.l.o.g. assume that their sets of variables are disjoint. If not, we just pick a disjoint set of variables of the same size and use the isomorphic grammar with that set of variables instead.

Proposition 1.14 also shows that there are many languages that cannot be produced by any grammar: By Proposition 1.3, there are uncountably many languages, but only countably many of them are generated by a grammar. Thus, most languages are not generated by a grammar.

Example 1.15. As an illustration, we show that Chomsky's example "Furiously sleep ideas green colourless" is not derivable in the grammar given in § 1.3. We first need to specify the grammar G formally: Σ is the finite set of all words in some English dictionary,

$$V := \{S, \mathrm{NP}, \mathrm{VP}, \mathrm{Adj}, \mathrm{Adv}, \mathrm{Verb}, \mathrm{Noun}\},$$

and P is the list of production rules given in (*) together with the dictionary rules that transform the nonterminals into the corresponding terminals, i.e., the relevant production rules are

$$\begin{aligned} S &\to \mathrm{NP}\,\mathrm{VP}, & \mathrm{Noun} &\to \text{ideas},\\ \mathrm{NP} &\to \mathrm{Adj}\,\mathrm{NP}, & \mathrm{Verb} &\to \text{sleep},\\ \mathrm{NP} &\to \mathrm{Noun}, & \mathrm{Adj} &\to \text{colourless},\\ \mathrm{VP} &\to \mathrm{Verb}, & \mathrm{Adj} &\to \text{green},\\ \mathrm{VP} &\to \mathrm{VP}\,\mathrm{Adv}, & \mathrm{Adv} &\to \text{furiously}. \end{aligned}$$

We claim that no derivable string ends in either Adj or colourless (let's call a string that does not end in either of these two *good*) and show this by induction on the length of the derivation. Clearly, any derivation of length zero produces S which is a good string. Suppose all derivations of length n produce only good strings and assume that α is produced by a derivation of length $n+1$. Let's assume that the last step of that derivation is $\beta \xrightarrow{G}_1 \alpha$. Clearly, β has a derivation of length n, so by induction hypothesis, β is a good string.

An inspection of our grammar rules show that since β does not end in Adj and $\beta \xrightarrow{G}_1 \alpha$, then α does not end in Adj. So, if α is not good, it must end in colourless. Furthermore, the only rule that could produce colourless is the rule Adj $\to$ colourless. Thus, if α ends in colourless, then β must end in Adj. Contradiction!

1.5 The Chomsky hierarchy

Fix Σ, V, and $S \in V$.

(1) A production rule $\alpha \to \beta$ is called *noncontracting* if $|\alpha| \leq |\beta|$.

(2) A production rule $\alpha \to \beta$ is called *context-sensitive* if there are $\gamma, \delta, \eta \in \Omega^*$ and $A \in V$ with $\alpha = \gamma A \delta$, $\beta = \gamma\eta\delta$, and $\eta \neq \varepsilon$.

(3) A production rule $A \to \beta$ is called *context-free* if $A \in V$ and $|\beta| \geq 1$.

(4) Production rules $A \to a$ and $A \to aB$ are called *regular* if $A, B \in V$ and $a \in \Sigma$.

We observe that context-sensitive, context-free, and regular production rules are noncontracting. Every context-free rule is context-sensitive (just let $\gamma = \delta = \varepsilon$) and every regular rule is context-free. regular ⇒ context free ⇒ context sensitive ⇒ non contracting

We call a grammar *noncontracting*, *context-sensitive*, *context-free*, or *regular* if all of its production rules are noncontracting, context-sensitive, context-free, or regular, respectively. If G is a noncontracting grammar, we know that any string in $\mathcal{D}(G, S)$ must have length at

least $|S| = 1$ [proof by induction on the length of the derivation]. Thus, a noncontracting grammar can never derive the empty word ε.

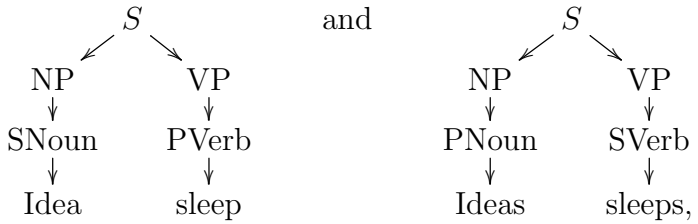
We call a language *noncontracting*, *context-sensitive*, *context-free*, or *regular* if it is produced by a noncontracting, context-sensitive, context-free, or regular grammar, respectively.

By the above remark, noncontracting, context-sensitive, context-free, and regular languages cannot contain ε. As a consequence, we shall focus for most of this course on languages $L \subseteq \mathbb{W}^+$. In § 1.8, we discuss the possibility of generalising the notions to allow our grammars to derive the empty word.

Example 1.16. The rules in the generative grammar for Chomsky's example sentence "Colourless green ideas sleep furiously" from Example 1.15 is context-free, since all production rules have a single variable on the left-hand side. This is not in general true for production rules of natural language. Suppose we have variables SNoun for "singular noun", SVerb for "singular verb", PNoun for "plural noun", and PNoun for "plural verb". Then the context-free rules

$$\mathrm{NP} \to \mathrm{SNoun},\ \mathrm{NP} \to \mathrm{PNoun},\ \mathrm{VP} \to \mathrm{SVerb}, \text{ and } \mathrm{VP} \to \mathrm{PVerb}$$

would allow us to derive the ungrammatical sentences "Idea sleeps" and "Ideas sleep" by the derivations



so we need to replace them with the context-sensitive production rules

$$\mathrm{NP\,VP} \to \mathrm{SNoun\,VP}, \qquad \mathrm{NP\,VP} \to \mathrm{PNoun\,VP},$$
$$\mathrm{SNoun\,VP} \to \mathrm{SNoun\,SVerb}, \text{ and} \qquad \mathrm{PNoun\,VP} \to \mathrm{PNoun\,PVerb}.$$

Chomsky called languages generated by any grammar *type 0 languages*, context-sensitive languages *type 1 languages*, context-free languages *type 2 languages*, and regular languages *type 3 languages*. The noncontracting languages are missing in this list of types since Chomsky proved that they are the same as the type 1 languages, i.e., a language is noncontracting if and only if it is context-sensitive (this is discussed on Example Sheet # 1).

By the above remarks, we know that the four Chomsky types form a hierarchy, i.e., that the set of languages of a type is a subset of the set of languages of any lower type. Furthermore, by Proposition 1.14, we know that each of the classes of languages is countable. We call this hierarchy the *Chomsky hierarchy* (Figure 1). A hierarchy like this is called *proper* if all of the classes are distinct, i.e., if each type has languages that are not of any higher type (e.g., that there is a context-free language that is not regular).

On the level of grammars, it is easy to see that there are context-free grammars that are not regular, context-sensitive grammars that are not context-free, and grammars that are not context-sensitive. But a grammar that is not regular can still be equivalent to a grammar that is regular as the following example shows:

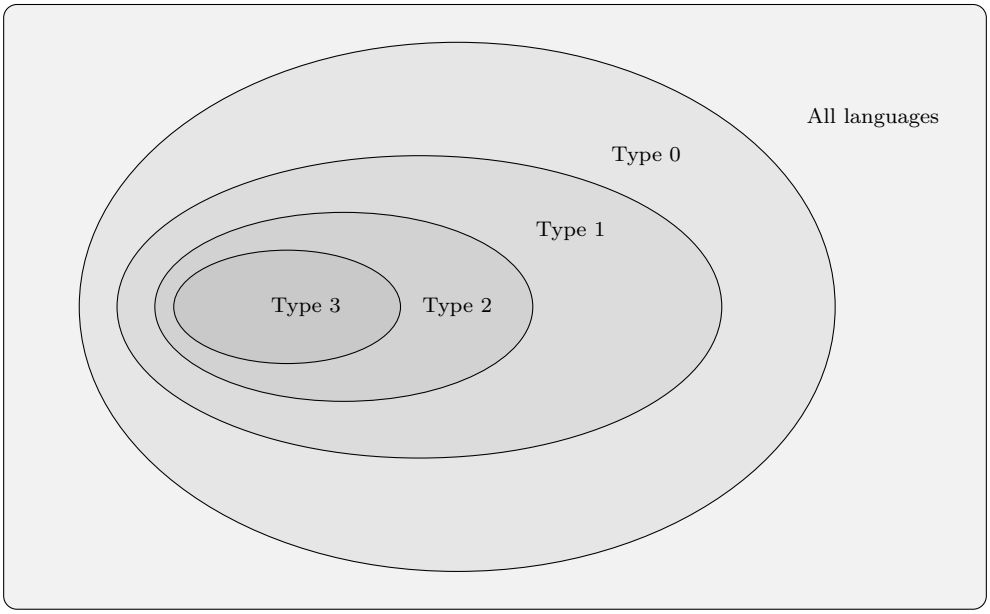


Figure 1: The Chomsky hierarchy

Example 1.17. Consider $\Sigma = \{a\}$, $V := \{A, S\}$, $P_8 := \{S \to aA, S \to a, A \to aS\}$, and $G_8 := (\Sigma, V, P_8, S)$. This is a regular grammar and $\mathcal{L}(G_8) = \{a^{2n+1}\,;\, n \in \mathbb{N}\}$. Thus, the set of odd-length words with the letter a is a regular grammar.

By Example 1.17, $\{a^{2n+1}\,;\, n \in \mathbb{N}\}$ is a regular language. However, the grammars with production rules P_0, P_1, P_2, P_3, and P_4 given in or after Example 1.10 are not regular; the grammar with production rules P_5 is not even context-free; the grammar with production rules P_6 is not even context-sensitive; and the grammar with production rules P_7 is not even noncontracting. And yet, they are all equivalent and produce a regular language.

This example highlights that proving that the Chomsky hierarchy is proper is more complicated: we need techniques to prove that languages are not in a given Chomsky type.

1.6 Decision problems

Historically, computability theory and the theory of models of computation was driven by *decision problems*. David Hilbert (1862–1943) gave an address at the *International Congress of Mathematicians* in Paris in the year 1900 in which he formulated mathematical problems for the 20th century.[6] One of them was *Hilbert's Tenth Problem*:

> Given a diophantine equation with any number unknown quantities and with rational integral numerical coefficients: *To devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers.*

[6] D. Hilbert (1900). Mathematische Probleme. Vortrag, gehalten auf dem internationalen Mathematiker-Kongreß zu Paris 1900. Nachrichten von der Königlichen Gesellschaft der Wissenschaften zu Göttingen. Mathematisch-Physikalische Klasse, 3, 253–297.

Several decades later Hilbert and Wilhelm Ackermann (1896–1962) formulated the so-called *Entscheidungsproblem* ("decision problem") in their monograph on mathematical logic:[7]

> From the considerations of the last section, we conclude the fundamental importance of the problem, to determine for any given formula of predicate calculus whether it is [logically valid] or not.

Both of these questions ask for a procedure to determine the answer to a question, in the usual terminology, for an *algorithm*. If the answer to the two mentioned problems is positive (and that was presumably Hilbert's expectation), it can be given by producing such an algorithm.

Note that we did not define the word "algorithm". As long as we are giving positive answers, this is not an issue: if a procedure is obviously algorithmic, we do not need a formal definition of the word "algorithm". In other words, it is enough to have *sufficient* criteria for being an algorithm that allow us to see a proposed algorithm and determine whether it actually is one. However, negative answers will require a definition of what an "algorithm" is, or more specifically, *necessary* criteria for being an algorithm. This means that there is a strong asymmetry between positive and negative answers to decision problems. Positive answers are usually considerably easier to give. We shall come back to the question of formalising the notion of an algorithm in § 4.5.[8]

In this section, we'll formulate the typical decision problems for grammars. Let G and G' be formal grammars and $w \in \mathbb{W}$ be a word.

The word problem. Is there an algorithm to determine whether $w \in \mathcal{L}(G)$?

The emptiness problem. Is there an algorithm to determine whether $\mathcal{L}(G) = \varnothing$?

The equivalence problem. Is there an algorithm to determine whether $\mathcal{L}(G) = \mathcal{L}(G')$?

We say that a decision problem is *solvable* if there is such an algorithm and that it is *unsolvable* if there is not. These three decision problems will be a guiding motivation throughout this lecture course. It will turn out that all three (general) decision problems are unsolvable. We are therefore particularly interested in restricting the decision problems to the classes of grammars given by the Chomsky hierarchy. E.g., the *word problem for regular grammars* is the question whether there is an algorithm that determines for any regular grammar G and word $w \in \mathbb{W}$ whether $w \in \mathcal{L}(G)$.

In this section, we shall give a positive solution for the word problem for type 1, type 2, and type 3 languages. We shall return to the word problem for type 0 languages in § 4.8 (Corollary 4.45).

Lemma 1.18. If G is noncontracting and $w \in \mathbb{W}$, then there is a bound N depending only on $|w|$ and $|\Omega|$ such that $w \in \mathcal{L}(G)$ if and only if w has a derivation of length at most N.

[7] D. Hilbert, W. Ackermann (1928). *Grundzüge der theoretischen Logik*. Springer-Verlag, p. 90.

[8] We mention briefly that both of the mentioned Hilbert problems have negative answers. The negative solution to the *Entscheidungsproblem* will be discussed in § 4.8; the negative solution of Hilbert's Tenth Problem was provided by Davis, Matiyasevich, Putnam, and Robinson. Cf. Y. V. Matiyasevich (1993). Hilbert's Tenth Problem. MIT Press & M. Davis (1973). Hilbert's Tenth Problem is Unsolvable. *American Mathematical Monthly* 80, 233–269.

Proof. Suppose $w \in \mathcal{L}(G)$, then there is a derivation $(\sigma_0, ..., \sigma_n)$ such that $\sigma_0 = S$ and $\sigma_n = w$. Let's assume that n is minimal with the property that there is a derivation of length n. Since G is noncontracting, we know that for $k \leq \ell$, we have $|\sigma_k| \leq |\sigma_\ell|$, so the sequence of lengths of the strings in the derivation is nondecreasing. However, for a fixed length m, the number of strings of length m is fixed: it is $|\Omega|^m$. This means by the pigeonhole principle that if there are more than $|\Omega|^m$ consecutive strings of length m in the derivation, then one of them must repeat. But then the derivation can be shortened by eliminating the loop: that's a contradiction to the assumption that n is the minimal length of a derivation of w. So, the derivation contains at most $|\Omega|^m$ strings of length m. But this allows us to give an upper bound on the length of the entire derivation, viz.

$$N := \sum_{i=1}^{|w|} |\Omega|^m.$$

Q.E.D.

Theorem 1.19. The word problem for noncontracting grammars is solvable.

Proof. Given w and G, compute $N := \sum_{m=1}^{|w|} |\Omega|^m$ and systematically check all derivations of length N. If one of them produces w, output "Yes"; if not, output "No". By Lemma 1.18, this algorithm produces the correct result. Q.E.D.

Since type 1, type 2, and type 3 languages are all noncontracting, this solves the word problem for all of these classes. We shall return to the emptiness and the equivalence problems in §§ 2.8, 3.5, & 4.13.

1.7 Closure properties

There are a number of algebraic operations on languages that allow us to combine languages to new languages. Let $L, M \subseteq \mathbb{W}^+$ be any languages over an alphabet Σ.

(a) *Concatenation.* The language LM consists of words vw such that $v \in L$ and $w \in M$.

(b) *Union.* The language $L \cup M$ consists of words either in L or in M.

(c) *Intersection.* The language $L \cap M$ consists of words that are both in L and M.

(d) *Complement.* The language $\overline{L} := \mathbb{W}^+ \backslash L$ consists of nonempty words that are not in L.

(e) *Difference.* The language $L \backslash M$ consists of words in L that are not in M.

We are particularly interested in which classes of languages are closed under which operations. Basic set theoretic relationships between the operations show that there are various implications between the closure properties:

Lemma 1.20. Let $\mathcal{C}$ be a class of languages. Then the following implications hold:

(a) If $\mathcal{C}$ is closed under union and complementation, then it is closed under intersection.

(b) If $\mathcal{C}$ is closed under intersection and complementation, then it is closed under union.

(c) If $\mathcal{C}$ is closed under intersection and complementation, then it is closed under difference.

(d) If $\mathbb{W}^+ \in \mathcal{C}$ and $\mathcal{C}$ is closed under difference, then it is closed under complementation.

Proof. These are all set algebra consequences of the definitions and de Morgan's Laws $\mathbb{W}^+\backslash(A \cap B) = \mathbb{W}^+\backslash A \cup \mathbb{W}^+\backslash B$ and $\mathbb{W}^+\backslash(A \cup B) = \mathbb{W}^+\backslash A \cap \mathbb{W}^+\backslash B$. Q.E.D.

It is useful to realise that some of the operations correspond to simple transformations of grammars, but they work only if we remove the possibility of undesirable interactions between the grammars. By Proposition 1.13, we already know that we can assume w.l.o.g. that two grammars have disjoint sets of variables.

Definition 1.21. A production rule is called *variable-based* if its left-hand side does not contain any letters. A grammar is called *variable-based* if all of its rules are variable-based.

Lemma 1.22. For every grammar, there is a variable-based grammar that is equivalent to it.

Proof. Add new variables X_a for every letter $a \in \Sigma$; let $V' := V \cup \{X_a\,;\, a \in \Sigma\}$. For each production rule $\alpha \to \beta \in P$, replace every occurrence of a letter a occurring in α by the corresponding new variable X_a; we write $X(\alpha)$ for this string. Clearly, $X(\alpha)$ does not contain any letters anymore, and so $X(\alpha) \to X(\beta)$ is a variable-based rule. Now define $P' := \{X(\alpha) \to X(\beta)\,;\, \alpha \to \beta \in P\} \cup \{X_a \to a\,;\, a \in \Sigma\}$ and $G' := (\Sigma, V', P', S)$.

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

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

Conversely, assume that $w \in \mathcal{L}(G')$ and let $S = \sigma_0 \xrightarrow{G'}_1 \ldots \xrightarrow{G'}_1 \sigma_m = w$ be a G'-derivation of w. If we apply the operation X to this derivation, we obtain a sequence $(\tau_0, ..., \tau_m)$ with $\tau_0 = S = \sigma_0 = X(\sigma_0)$ and $\tau_i = X(\sigma_i)$. This sequence is not necessarily a G'-derivation. If $\sigma_i \xrightarrow{G'}_1 \sigma_{i+1}$ was an application of a rule of the form $X(\alpha) \to X(\beta)$, then the same rule will warrant that $X(\sigma_i) \xrightarrow{G'}_1 X(\sigma_{i+1})$; if $\sigma_i \xrightarrow{G'}_1 \sigma_{i+1}$ was an application of one of the rules $X_a \to a$, then applying X will result in $X(\sigma_i) = X(\sigma_{i+1})$. Since for each letter a there is only one production rule that produces a, we know that $|w|$ many steps of the derivation must be of this form. Thus, removing these $|w|$ many steps will make the remainder of the sequence $(\tau_0, ..., \tau_m)$ a G'-derivation of length $m - |v|$ of $X(w)$. But then $w \in \mathcal{L}(G)$ by our earlier observation. Q.E.D.

Note that the transformation $P \mapsto P'$ in this proof preserves being noncontracting, being context-sensitive, and being context-free, but not being regular. However, regular grammars are variable-based anyway, so there is no need to apply Lemma 1.22 to a regular grammar.

Let $G = (\Sigma, V, P, S)$ and $G' = (\Sigma, V', P', S')$ be two grammars over the same alphabet Σ.

(a) *Concatenation.* The *concatenation grammar of G and G'* is $(\Sigma, V \cup V' \cup \{T\}, P^*, T)$ with a new variable T and $P^* := \{T \to SS'\} \cup P \cup P'$.

(b) *Union.* The *union grammar of G and G'* is $(\Sigma, V \cup V' \cup \{T\}, P^*, T)$ with a new variable T and $P^* := \{T \to S, T \to S'\} \cup P \cup P'$.

Remark 1.23. Note that if G and G' are context-free or context-sensitive, then so are their concatenation and union grammars (since all three new productions $T \to SS'$, $T \to S$, and $T \to S'$ are context-free). Even if G and G' are regular, then their union and concatenation grammars are not regular, since the new productions $T \to S$, $T \to S'$, and $T \to SS'$ are not regular rules. (We'll discuss this in § 2.1.)

Proposition 1.24. Let G and G' be grammars that do not share any variables and are variable-based. Let H be their concatenation grammar. Then $\mathcal{L}(H) = \mathcal{L}(G)\mathcal{L}(G')$.

Proof. For the forward direction, let $vw \in LM$, i.e., $v \in L$ and $w \in M$. By definition, we have a G-derivation $(\sigma_0, ..., \sigma_n)$ of v and a G'-derivation $(\tau_0, ..., \tau_m)$ of w. Then

$$T \xrightarrow{H}_1 SS' = \sigma_0 S' \xrightarrow{G}_1 \sigma_1 S' \xrightarrow{G}_1 ... \xrightarrow{G}_1 \sigma_n S' = vS'$$
$$= v\tau_0 \xrightarrow{G'}_1 v\tau_1 \xrightarrow{G'}_1 ... \xrightarrow{G'}_1 v\tau_m = vw$$

is an H-derivation of vw.

For the converse, let $T = \sigma_0 \xrightarrow{H}_1 \sigma_1 \xrightarrow{H}_1 ... \xrightarrow{H}_1 \sigma_n = u$ be any H-derivation. Since there is only one rule involving T, we know that $\sigma_1 = SS'$. For $i \geq 1$, if $\sigma_i = x_0...x_\ell$, we define by recursion what it means that x_j *belongs to the first half in* σ_i. Our definition will be done in such a way that all variables occurring in the first half are in V and all variables occurring in the other half are in V'. If $i = 1$, we say that $S \in V$ belongs to the first half in σ_1 and $S' \in V'$ doesn't. Suppose $\sigma_i = \alpha\gamma\beta$ and $\sigma_{i+1} = \alpha\delta\beta$, i.e., σ_{i+1} is produced by an application of the rule $\gamma \to \delta$. We assumed that our grammars were variable-based, and hence γ consists only of variables. By definition of H, these must either all be from V or all from V'. In the first case, γ lies entirely in the first half; in the second case, γ lies entirely in the other half. Any symbol instance occurring in σ_{i+1} lies either in α, δ, or β. If the symbol instance is in α or β then it already occurred in σ_i, and we say that it *belongs to the first half in* σ_{i+1} if and only if it belonged to the first half in σ_i. If the symbol instance is in δ, then it *belongs to the first half in* σ_{i+1} if and only γ was entirely in the first half in σ_i (that's equivalent to $\gamma \in V^* \backslash \{\varepsilon\}$). If that's the case, we also say that the production step from i to $i + 1$ belongs to the first half.

An induction shows that the symbols belonging to the first half form an initial segment of each σ_i. So, we have $u = vw$ where v is the subword of letters belonging to the first half. We now collect all production steps that belong to the first half and observe that they form a G-derivation of v from S; similarly, all production steps that do not belong to the first half form a G'-derivation of w from S'. This shows that $u = vw \in LM$. Q.E.D.

Proposition 1.25. Let G and G' be grammars that do not share any variables and are variable-based. Let H be their union grammar. Then $\mathcal{L}(H) = \mathcal{L}(G) \cup \mathcal{L}(G')$.

Proof. Clearly, if $S \xrightarrow{G} v$, then $T \xrightarrow{H} v$ by using the rule $T \to S$; similarly, if $S' \xrightarrow{G'} v$, then $T \xrightarrow{H} v$. Thus, $\mathcal{L}(G) \cup \mathcal{L}(G') \subseteq \mathcal{L}(H)$.

Since $V \cap V' = \varnothing$ and the grammars are variable-based, no rule from P can apply to a string that contains no variables from V and no rule from P' can apply to a string that contains no variables from V'. As a consequence, we see (by induction) that any H-derivation starting from S will only use rules from P and any H-derivation starting from S' will only use rules from P'. Thus, if $S \xrightarrow{H} v$, then $S \xrightarrow{G} v$ and if $S' \xrightarrow{H} v$, then $S' \xrightarrow{G'} v$. But since there are only two rules involving T, any H-derivation $(\sigma_0, \sigma_1, ..., \sigma_n)$ with $\sigma_0 = T$ will have $\sigma_1 = S$ or $\sigma_1 = S'$, so $T \xrightarrow{H} v$ implies either $S \xrightarrow{G} v$ or $S' \xrightarrow{G'} v$. Q.E.D.

Corollary 1.26. The classes of type 0, type 1, and type 2 languages are closed under concatenation and union.

Proof. As mentioned, context-free grammars are variable-based; note, furthermore, that the construction in the proof of Lemma 1.22 preserves being context-sensitive and being noncontracting. Thus, by the proofs of Proposition 1.14 and Lemma 1.22, we may assume w.l.o.g. that $V \cap V' = \varnothing$ and that P and P' are variable-based, thus we can apply Propositions 1.24 & 1.25 in combination with Remark 1.23. Q.E.D.

It is not obvious how to produce grammars for the other closure properties (intersection, complementation, difference). The question whether the relevant classes are closed under these operations will play a major role in our discussions of the Chomsky classes.

1.8 A comment on the empty word

As mentioned, noncontracting grammars cannot produce the empty word ε (cf. the proof of Lemma 1.18). What if we wish to talk about languages that may contain the empty word, e.g., the language of even-length words or the set of words that do not contain the letter a?

We can fix this easily by additionally allowing rules that produce the empty word in our production rules. Let us call any production rule $\alpha \to \varepsilon$ an *ε-production* and the rule $S \to \varepsilon$ the *basic ε-production*. Of course, these rules are not noncontracting, so adding these rules to any grammar will catapult it out of the Chomsky hierarchy.

Even with just the basic ε-production, we can easily mimic arbitrary production rules in a noncontracting way: if $\alpha \to \beta$ is a rule with $|\beta| < |\alpha|$, say, $|\alpha| = n + k > n = |\beta|$, then consider the (noncontracting) rule $\alpha \to \beta S^k$. In the presence of $S \to \varepsilon$, this rule can be used to produce the effect of the original contracting rule $\alpha \to \beta$.

In order to avoid this, we call a production rule *ε-adequate* if the symbol S only appears on the left-hand side. A grammar (G, V, P, S) is called *ε-adequate* if all of its production rules are. In order to avoid very lengthy theorem statements, we use the letter Q to stand for one of the four properties of being regular, context-free, context-sensitive, or noncontracting.

Proposition 1.27. Any grammar is equivalent to an ε-adequate grammar. Moreover, any grammar with property Q is equivalent to an ε-adequate grammar with property Q.

Proof. Suppose $G = (\Sigma, V, P, S)$ is a grammar. Take a new variable $T \notin V$ and let

$$\begin{aligned} V' &:= V \cup \{T\}, \\ P' &:= P \cup \{T \to \alpha \,;\, S \to \alpha \in P\}, \text{ and} \\ G' &:= (\Sigma, V', P', T). \end{aligned}$$

Clearly, G' is ε-adequate and obviously $\mathcal{L}(G) = \mathcal{L}(G')$. Observe that the transformation $P \mapsto P'$ preserves all four properties that Q can stand for. Q.E.D.

A grammar is called *essentially* Q if it is ε-adequate and all of its production rules are either the basic ε-rule or have property Q. A language is called called *essentially* Q if it is produced by an essentially Q grammar.

Proposition 1.28. A language L is essentially Q if and only if $L\backslash\{\varepsilon\}$ is Q.

Proof. "$\Rightarrow$": Let L be a language that is essentially Q as witnessed by a grammar G that is essentially Q. First of all, let us observe that if all rules are ε-adequate, then no derivation will contain S except at the very beginning: no rule can ever introduce an instance of S, and the only instance of S will need to be rewritten by a string without S in the first step of the derivation. That means that if $S \to \varepsilon$ is a rule of G, it must be used in the first step of the derivation. All other derivation steps will only use productions that have property Q. Thus, the grammar obtained by removing the rule $S \to \varepsilon$ from G will produce a language L' that is Q and $L' = L\backslash\{\varepsilon\}$.

"$\Leftarrow$": Let G be a Q grammar producing $L\backslash\{\varepsilon\}$. By Proposition 1.27, we can assume w.l.o.g. that G is ε-adequate; so G is essentially Q. As in the proof of the other direction, any G-derivations can only contain S at the very beginning. If $\varepsilon \notin L$, then $\mathcal{L}(G) = L$ and we are done. If $\varepsilon \in L$, then the grammar G' that adds $S \to \varepsilon$ to G will satisfy $\mathcal{L}(G') = L$ (using the fact that no G-derivations contain S except at the very beginning). Q.E.D.

2 Regular languages

2.1 Understanding regular derivations

A regular grammar only has two types of production rules, $A \to aB$, called *nonterminal rules*, and $A \to a$, called *terminal rules*. The syntactic form of regular grammars seriously restricts what we can do with them.

Lemma 2.1. Let $G = (\Sigma, V, P, S)$ be a regular grammar.

(a) If $\alpha \in \Omega^*$ and $S \xrightarrow{G} \alpha$, then α is either a word or of the form wA where $w \in \mathbb{W}$ and $A \in V$.

(b) If $v \in \mathbb{W}$ and $S \xrightarrow{G} v$, then any derivation of v has length $|v|$ and only the final step of the derivation is an application of a terminal rule, the others are application of nonterminal rules. The terminal rule creates the final letter in the word v.

Proof. (a) is easily proved by induction: it is true for S and if any of the rules of G is applied to wA, it either produces a word or some $w'B$ where $w' \in \mathbb{W}$ and $B \in V$.

For (b), we observe that nonterminal rules keep the number of variables the same; terminal rules reduce the number of variables by one. Claim (a) shows that the derivation of a word w consists of strings of the form wA for all steps except the last. Therefore, all production steps except for the last must have been nonterminal and the last one is terminal. Note that nonterminal rules increase the length by one and terminal rules keep the length the same. Since the length of the string consisting only of the start symbol is 1, we have applied $|v|-1$ nonterminal rules before reaching a string of length $|v|$. Since the penultimate string is of the form wA and the final production step is a terminal rule, it's clear that this rewrites A into the final letter of v. Q.E.D.

Note that even in this severely restricted case, we do not have uniqueness of derivations: the following is a regular grammar for the language $\{01\}$: $S \to 0A, S \to 0B, A \to 1, B \to 1$; both $S \to 0A \to 01$ and $S \to 0B \to 01$ are derivations of 01.

Now that we understand what regular derivations look like, we can re-visit the question of closure under union and concatenation. The union and concatenation grammars we used in § 1.7 were not regular, so we need to give alternative constructions. Let $G = (\Sigma, V, P, S)$ and $G' = (\Sigma, V', P', S')$ be regular grammars.

(a) The *regular concatenation grammar of G and G'* is $(\Sigma, V \cup V', P^*, S)$ where $P^* := P' \cup (P \backslash \{A \to a\,;\, A \to a \in P\}) \cup \{A \to aS'\,;\, A \to a \in P\}$.

(b) The *regular union grammar of G and G'* is $(\Sigma, V \cup V' \cup \{T\}, P^*, T)$ with a new variable T and $P^* := P \cup P' \cup \{T \to \alpha\,;\, S \to \alpha \in P\} \cup \{T \to \alpha\,;\, S' \to \alpha \in P'\}$.

Clearly, if G and G' are regular, then so are the regular concatenation and regular union grammars.

Proposition 2.2. Let G and G' be regular grammars with disjoint sets of variables. If H is their regular concatenation grammar, then $\mathcal{L}(H) = \mathcal{L}(G)\mathcal{L}(G')$; if H is their regular union grammar, then $\mathcal{L}(H') = \mathcal{L}(G) \cup \mathcal{L}(G')$.

Proof. Concatenation. Suppose $vw \in \mathcal{L}(G)\mathcal{L}(G')$ and let $S \xrightarrow{G} v$ and $S' \xrightarrow{G'} w$. By Lemma 2.1 (b), we know that the final production in the derivation of v is a terminal rule, say $A \to a$ and therefore $A \to aS'$ is a rule in H. Thus, we have that $S \xrightarrow{H} vS'$. We also have that $S' \xrightarrow{G'} w$ and hence $S' \xrightarrow{H} w$, so together $S \xrightarrow{H} vw$.

For the other direction, suppose $S \xrightarrow{H} u$ and let $S = \sigma_0 \xrightarrow{H}_1 \sigma_1 \xrightarrow{H}_1 ... \xrightarrow{H}_1 \sigma_n = u$ be an H-derivation, By Lemma 2.1, we know that all strings σ_i (for $0 \leq i < n$) contain exactly one variable. If σ_i contains a variable from V', then this remains true for all σ_j with $j \geq i$. Therefore, the strings with variables from V form an initial segment of the derivation. Let m be the unique number such that σ_m has a variable from V and σ_{m+1} has a variable from V'. The only H-rule that allows to do that is of the form $A \to aS'$, so $\sigma_m = xA$ and $\sigma_{m+1} = xaS'$ and all rules applied before m are G-rules and all rules applied after $m+1$ are G'-rules. Thus $S \xrightarrow{G} xA$ and therefore (using the original rule $A \to a$ in G), $S \xrightarrow{G} xa$. Write $v := xa$ and $u = vw$. Then $S' \xrightarrow{G'} w$.

Union. This proof does not need that G and G' are regular; we shall only use that they are context-sensitive. The direction "$\supseteq$" is obvious since any derivation $S \xrightarrow{G} w$ or $S' \xrightarrow{G'} w$ can be made into a derivation $T \xrightarrow{H} w$ by exchanging the first rule application rewriting either S or S' by the corresponding rule in P^* rewriting T.

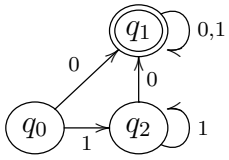
For the other direction, suppose $T \xrightarrow{H} w$. Since H is context-sensitive, all strings occurring in this derivation except for the last one must contain variables (once a string is a word, nothing can be rewritten anymore as every production rule needs a variable to be rewritten). If $T \xrightarrow{H}_1 w$, i.e., the derivation has length one, then it is the result of a rule application of $T \to w$. By definition, either $S \to w \in P$ or $S' \to w \in P'$, so $w \in \mathcal{L}(G) \cup \mathcal{L}(G')$. Otherwise, we have $T \xrightarrow{H}_1 \alpha \xrightarrow{H} w$ with α containing variables. That $T \xrightarrow{H}_1 \alpha$ is either witnessed by some rule $S \to \alpha \in P$ or some rule $S' \to \alpha \in P'$. In the former case, all variables in α are in V; in the latter case, all variables in α are in V'. W.l.o.g., let's assume that we are in the first situation, i.e., all variables in α are in V and $S \to \alpha \in P$. By induction (and the fact that H is context-sensitive, i.e., only variables get rewritten), all variables occurring in the rest of the derivation $\alpha \xrightarrow{H} w$ will also be in V, so all rules applied in the derivation come from P and thus we have $\alpha \xrightarrow{G} w$. But now $S \xrightarrow{G}_1 \alpha \xrightarrow{G} w$, thus $S \xrightarrow{G} w$, and therefore $w \in \mathcal{L}(G) \subseteq \mathcal{L}(G) \cup \mathcal{L}(G')$. Q.E.D.

Corollary 2.3. The class of regular languages is closed under concatenation and union.

2.2 Deterministic automata

Let Σ be an alphabet. Then a tuple $D = (\Sigma, Q, \delta, q_0, F)$ is called a *(deterministic) automaton* if Q is a finite set such that $q_0 \in Q$, $F \subseteq Q \backslash \{q_0\}$, and $\delta : Q \times \Sigma \to Q$. The elements of Q are called *states*; the function δ is called the *transition function*.

We graphically represent automata by directed labelled graphs where the vertices are labelled by the elements of Q, each vertex has precisely $|\Sigma|$ immediate successors marked by directed edges labelled with the letters of the alphabet Σ. The vertices labelled with non-elements of F get a single circle, and the vertices labelled with elements of F get a double circle. The following is an example for $\Sigma = \{0, 1\}$, $Q = \{q_0, q_1, q_2\}$, and $F = \{q_1\}$:



Note that we simplify our graphical representations by writing a single arrow with multiple labels if they all have the same source and target (e.g., the arrow from q_1 to itself in the above diagram).

We interpret the automaton $D = (\Sigma, Q, \delta, q_0, F)$ as follows: q_0 is the *start state* and the automaton starts in this state. The automaton is given a word $w \in \mathbb{W}$ and reads it letter by letter from the beginning. The transition function δ tells the automaton what to do: if the automaton is in the state q and reads the letter a, it moves to the state $\delta(q, a)$. After reading a letter from the word, the automaton then proceeds to the next letter. Once it is done reading the word, it will be in a particular state q. The set F is the set of *accepting states*: the automaton accepts the word w if and only if that state is in F.

More formally, we define a function $\widehat{\delta} : Q \times \mathbb{W} \to Q$ by recursion on the length of the word:

$$\widehat{\delta}(q, \varepsilon) := q \text{ and}$$
$$\widehat{\delta}(q, wa) := \delta(\widehat{\delta}(q, w), a),$$

and define the *language accepted by* D via $\mathcal{L}(D) := \{w\,;\, \widehat{\delta}(q_0, w) \in F\}$. Here, we say that D *accepts* w if $\widehat{\delta}(q_0, w) \in F$ and that D *rejects* w if $\widehat{\delta}(q_0, w) \notin F$. Note that for every $w = a_0...a_n$, the function $\widehat{\delta}$ uniquely determines a sequence of states that the automaton passes through during a computation: $q_0 = \widehat{\delta}(q_0, \varepsilon)$, $q_1 := \widehat{\delta}(q_0, a_0)$, $q_2 := \widehat{\delta}(q_0, a_0a_1)$, ..., $q_{n+1} := \widehat{\delta}(q_0, a_0...a_n) = \widehat{\delta}(q_0, w)$. The sequence is also called the *state sequence of the computation*.

Example 2.4. The automaton graphically represented above accepts the language

$$L := \{w\,;\, w \text{ contains at least one } 0\}.$$

[Let us analyse which words will result in state q_0, q_1, and q_2, respectively. By this we mean words s such that $\widehat{\delta}(q_0, w) = q_i$. Since q_0 has no incoming edges, the only word that results in q_0 is ε. The state q_2 has two incoming edges, a 1-transition from q_0 which means that the word $\varepsilon 1 = 1$ results in q_2, and then a 1-transition from q_2. So, by induction, any finite sequence consisting entirely of 1s will result in q_2; thus, the words that result in q_2 are precisely the words in $\{1\}^+$. Finally, all 0-transitions lead to q_1, so any word that contains a 0 will always be in state q_1 immediately after reading that 0. However, since both transitions from q_1 lead to q_1, you cannot ever leave that state. In summary, the empty word ends in q_0, any word consisting of 1s results in q_2, and any word that contains a 0 results in q_1. This description covers all possible words. We note that q_1 is the only accepting state, which proves the claim.]

If $D = (\Sigma, Q, \delta, q_0, F)$ and $D' = (\Sigma, Q', \delta', q_0', F')$ are deterministic automata over the same alphabet Σ, we say that a map $f : Q \to Q'$ is a *homomorphism from D to D'* if

(i) for all $q \in Q$ and $a \in \Sigma$, we have that $\delta'(f(q), a) = f(\delta(q, a))$,

(ii) we have $f(q_0) = q_0'$, and

(iii) for all $q \in Q$, $q \in F$ if and only if $f(q) \in F'$.

As usual, bijective homomorphisms are called *isomorphisms* and automata that have an isomorphism between them are called *isomorphic*. Note that if f is a bijection, then f^{-1} satisfies (i) to (iii) and thus is a homomorphism.

If f is a homomorphism, property (i) extends by induction to $\widehat{\delta'}(f(q), w) = f(\widehat{\delta}(q, w))$ for $w \in \mathbb{W}$.

This means that while homomorphisms are not in general surjective, they hit every state in Q' that is reachable from q_0' (i.e., a state of the form $\widehat{\delta'}(q_0', w)$ for some word $w \in \mathbb{W}$) by property (i); these states are the only states that matter for $\mathcal{L}(D')$. Similarly, homomorphisms are not in general injective, but if $f(p) = f(q)$, then p and q have to agree on everything that is relevant for determining the accepted language: e.g., $p \in F$ if and only if $q \in F$ by property (iii). The two states need to be what we shall later (§ 2.7) call *indistinguishable*.

Proposition 2.5. If there is a homomorphism from D to D', then $\mathcal{L}(D) = \mathcal{L}(D')$.

Proof. Let f be a homomorphism from D to D'; then for any word w, we have

$$\begin{aligned} w \in \mathcal{L}(D) &\iff \widehat{\delta}(q_0, w) \in F \\ &\overset{\text{(iii)}}{\iff} f(\widehat{\delta}(q_0, w)) \in F' \\ &\iff \widehat{\delta'}(f(q_0), w) \in F' \\ &\overset{\text{(ii)}}{\iff} \widehat{\delta'}(q_0', w) \in F' \iff w \in \mathcal{L}(D'). \end{aligned}$$

Q.E.D.

Theorem 2.6. Any language accepted by a deterministic automaton is regular.

Proof. Let $D = (\Sigma, Q, \delta, q_0, F)$ and define $G = (\Sigma, Q, P, q_0)$ with the following production rules (for $p, q \in Q$ and $a \in \Sigma$): $p \to aq$ is in P if and only if $\delta(p, a) = q$ and $p \to a$ is in P if and only if $\delta(p, a) \in F$.

Suppose $a_0 ... a_n = w \in \mathcal{L}(D)$, i.e., $\widehat{\delta}(q_0, w) \in F$. This means the state sequence of the computation is given by $q_{i+1} := \delta(q_i, a_i)$ such that $\widehat{\delta}(q_0, w) = \delta(q_n, a_n) = q_{n+1}$. By definition, $q_i \to a_i q_{i+1} \in P$ and $q_n \to a_n \in P$ (since $q_{n+1} \in F$). Thus, the state sequence yields a G-derivation

$$q_0 \xrightarrow{G}_1 a_0 q_1 \xrightarrow{G}_1 a_0 a_1 q_2 \xrightarrow{G}_1 \ldots \xrightarrow{G}_1 a_0 a_1 ... a_{n-1} q_n \xrightarrow{G}_1 a_0 a_1 ... a_n = w, \qquad (\dagger)$$

so $\mathcal{L}(D) \subseteq \mathcal{L}(G)$.

Conversely, suppose $a_0a_1...a_n = w \in \mathcal{L}(G)$ and apply Lemma 2.1 to see that any G-derivation of w is the form (†) for letters a_i and variables q_i and furthermore that this means that for all $i < n$, we have $q_i \to a_iq_{i+1} \in P$, as well as $q_n \to a_n \in P$. By definition of P, this in turn means that $\delta(q_i, a_i) = q_{i+1}$ and $\delta(q_n, a_n) \in F$. We obtain immediately that $\widehat{\delta}(q_0, w) = \delta(q_n, a_n) \in F$, so $w \in \mathcal{L}(D)$. Q.E.D.

Accepting the empty word. To match § 1.8, we should briefly comment on the role of the empty word for automata. By the stipulation that $q_0 \notin F$, we make sure that the empty word can never be accepted by an automaton. This matches with our definition of regular grammars and we need this in the proof of Theorem 2.6: an automaton that accepts the empty word would require a derivation $q_0 \to \varepsilon$ in the grammar. If we modify our definitions of grammars as discussed in § 1.8, we could remove the stipulation that $q_0 \notin F$ from our definition of automata and retain the equivalence.

2.3 Nondeterministic automata

We would like to prove the converse of Theorem 2.6. However, the transformation of an automaton into a grammar from the proof of Theorem 2.6 is not invertible since a regular grammar could contain production rules $A \to aB$ and $A \to aC$ for $B \neq C$, but transformation functions in deterministic automata have to assign a unique value $\delta(q, a)$. This suggests a more liberal notion of automaton:

Let Σ be an alphabet. Then a tuple $N = (\Sigma, Q, \delta, q_0, F)$ is called a *nondeterministic automaton* if Q is a finite set such that $q_0 \in Q$, $F \subseteq Q\backslash\{q_0\}$, and $\delta : Q \times \Sigma \to \wp(Q)$. We think of $\delta(q, a)$ as the set of possible states that the automaton can reach from q upon reading a. The graphical representation of nondeterministic automata is the same as for deterministic automata, except that a given vertex may have multiple or no outgoing arrows labeled with the same letter a.

For nondeterministic automata, we recursively define a similar function $\widehat{\delta} : Q \times \mathbb{W} \to \wp(Q)$ by

$$\begin{aligned} \widehat{\delta}(q, \varepsilon) &:= \{q\} \text{ and} \\ \widehat{\delta}(q, wa) &:= \bigcup\{\delta(p, a)\,;\, p \in \widehat{\delta}(q, w)\} \end{aligned}$$

and define the *language accepted by* N via $\mathcal{L}(N) := \{w\,;\, \widehat{\delta}(q_0, w) \cap F \neq \varnothing\}$. The function $\widehat{\delta}$ collects all possible resulting states for all possible paths through the automaton. The automaton accepts a word if there is at least one such path that results in an accepting state. For deterministic automata, we had a *state sequence* given by the transition function. Similarly, $\widehat{\delta}$ produces a *state set sequence* for nondeterministic automata where $X_0 = \{q_0\}$ and $X_{i+1} = \bigcup\{\delta(p, a_i)\,;\, p \in X_i\}$.

Nondeterministic automata are a generalisation of deterministic automata: If $D = (\Sigma, Q, \delta, q_0, F)$ is a deterministic automaton, then $\delta^{\mathrm{n}}(q, a) := \{\delta(q, a)\}$ defines a transition function of a nondeterministic automaton $N := (\Sigma, Q, \delta^{\mathrm{n}}, q_0, F)$, and by induction, it is easy to see that $\widehat{\delta^{\mathrm{n}}}(q, a) = \{\widehat{\delta}(q, a)\}$, so $\mathcal{L}(N) = \mathcal{L}(D)$. However, at least superficially, nondeterministic automata feel much more general as they are able to check many computation

sequences at the same time. The following theorem shows that this superficial intution is wrong.

Theorem 2.7. For every nondeterministic automaton N there is a deterministic automaton D such that $\mathcal{L}(D) = \mathcal{L}(N)$.

Proof. The construction in this proof is known as the *subset construction*. Let $N = (\Sigma, Q, \delta, q_0, F)$ be a nondeterministic automaton. We define the a deterministic automaton $D := (\Sigma, \wp(Q), \Delta, \{q_0\}, G)$ where

$$\Delta(X, a) := \bigcup \{\delta(q, a)\,;\, q \in X\} \text{ and}$$
$$X \in G \iff X \cap F \neq \varnothing.$$

Since states in D are sets of states in N, any sequence of states in D is a sequence of sets of states in N. Fix a word $w = a_0...a_n$ and let $(X_0, ..., X_{n+1})$ be the state sequence corresponding to w in D and $(Y_0, ..., Y_{n+1})$ be the state set sequence corresponding to w in N, then we easily see by induction that $X_i = Y_i$. Thus

$$w \in \mathcal{L}(D) \iff X_{n+1} \in G \iff X_{n+1} \cap F \neq \varnothing \iff Y_{n+1} \cap F \neq \varnothing \iff w \in \mathcal{L}(N).$$

Q.E.D.

Note that the subset construction in general produces a deterministic automaton with 2^n states if the original nondeterministic automaton had n states.

Theorem 2.8. Every regular language is accepted by a nondeterministic automaton.

Proof. Let $G = (\Sigma, V, P, S)$ be a regular grammar. Let $H \notin \Sigma \cup V$ and define $Q := V \cup \{H\}$. We define $N := (\Sigma, Q, \delta, S, \{H\})$ with

$$\begin{array}{lcll} \delta(A, a) & := & \{B\,;\, A \to aB \in P\} & \text{if } A \to a \notin P \text{ and} \\ \delta(A, a) & := & \{B\,;\, A \to aB \in P\} \cup \{H\} & \text{if } A \to a \in P. \end{array}$$

Note that since $H \notin V$, we have that $\delta(H, a) = \varnothing$ for all $a \in \Sigma$.

If $a_0...a_n = w \in \mathcal{L}(G)$, by Lemma 2.1, there is a G-derivation

$$A_0 = S \xrightarrow{G}_1 a_0A_1 \xrightarrow{G}_1 a_0a_1A_2 \xrightarrow{G}_1 \ldots \xrightarrow{G}_1 a_0a_1...a_{n-1}A_n \xrightarrow{G}_1 a_0a_1...a_n = w,$$

with production rules $A_i \to a_iA_{i+1}$ and $A_n \to a_n$ in P. By definition (and induction), we obtain $H \in \widehat{\delta}(S, w)$, and thus $w \in \mathcal{L}(N)$.

Conversely, if $a_0...a_n = w \in \mathcal{L}(N)$, then there is a path through N via arrows labelled with the a_i leading to H, i.e., a path of states $q_0 = S$, q_1, ..., q_n, $q_{n+1} = H$ such that for each i, we have that $q_{i+1} \in \delta(q_i, a_i)$. In particular, all of the q_i except for the last one must be elements of V, and so $q_{i+1} \in \delta(q_i, a_i)$ must be witnessed by a production rule $q_i \to a_iq_{i+1} \in P$; furthermore, the fact that $q_{n+1} = H \in \delta(q_n, a_n)$ means that $q_n \to a_n \in P$. Combining these results in a G-derivation

$$q_0 = S \xrightarrow{G}_1 a_0q_1 \xrightarrow{G}_1 a_0a_1q_2 \xrightarrow{G}_1 \ldots \xrightarrow{G}_1 a_0...a_{n-1}q_n \xrightarrow{G}_1 a_0...a_n = w,$$

thus showing that $w \in \mathcal{L}(G)$. Q.E.D.

Corollary 2.9. A language L is regular if and only if it is accepted by a deterministic automaton.

Proof. Follows directly from Theorems 2.6, 2.7, & 2.8. Q.E.D.

2.4 The pumping lemma for regular languages

We now have a good understanding of how we can show that a language is regular, but we are still missing tools to prove that a language is not regular, even though we already know (Proposition 1.14) that almost all languages are not regular. In this section, we shall provide the main tool for proving non-regularity.

Definition 2.10. Let $L \subseteq \mathbb{W}$ be a language. We say that L *satisfies the (regular) pumping lemma with pumping number* n if for every word $w \in L$ such that $|w| \geq n$ there are words x, y, z such that $w = xyz$, $|y| > 0$, $|xy| \leq n$ and for all $k \in \mathbb{N}$, we have that $xy^k z \in L$. We say that L *satisfies the (regular) pumping lemma* if there is some n such that it satisfies the (regular) pumping lemma with pumping number n.

If a language L satisfies the pumping lemma and we have written $w = xyz$ as in the definition, then $xz = xy^0z$, xy^2z, xy^3z, etc. are all in L. We call the transition from $w = xyz$ to xz *pumping down* and the transition to xy^kz (for $k > 1$) *pumping up*.

Theorem 2.11 (The regular pumping lemma)**.** For every regular language L, there is a number n such that L satisfies the regular pumping lemma with pumping number n.

Proof. By Corollary 2.9, we know that L is accepted by a deterministic automaton $D = (\Sigma, Q, \delta, q_0, F)$. Let $n := |Q|$ and suppose that $w \in \mathcal{L}(D)$ such that $|w| \geq n$. We write $w = a_0...a_{n-1}v$ for some $v \in \mathbb{W}$. Consider the state sequence $q_0, q_1, ..., q_n$ obtained by letting D read $a_0...a_{n-1}$, i.e., we have $\delta(q_i, a_i) = q_{i+1}$. The state sequence has $n+1$ elements, and so by the pigeonhole principle, one of the states must occur twice in the state sequence. Let's fix $0 \leq i < j \leq n$ such that $q_i = q_j$ and let

$$\begin{aligned} x &:= a_0...a_{i-1}, \\ y &:= a_i...a_{j-1}, \text{ and} \\ z &:= a_j...a_{n-1}v, \end{aligned}$$

where the latter means $z = \varepsilon v = v$ if $j = n$. Note that our construction implies that $w = xyz$, $|y| > 0$, and $|xy| \leq n$. We also observe that

$$\widehat{\delta}(q_0, x) = q_i, \tag{a}$$
$$\widehat{\delta}(q_i, y) = \widehat{\delta}(q_j, y) = q_j = q_i, \text{ and} \tag{b}$$
$$\widehat{\delta}(q_i, z) = \widehat{\delta}(q_j, z) \in F. \tag{c}$$

Fix any k and prove that $xy^kz \in \mathcal{L}(D)$. For this, we prove by induction that $\widehat{\delta}(q_0, xy^k) = q_i$ for all k. For $k = 0$, this is just (a). If $\widehat{\delta}(q_0, xy^k) = q_i$, then $\widehat{\delta}(q_0, xy^{k+1}) = \widehat{\delta}(q_i, y) = q_i$ by

(b). Now, this implies that $\widehat{\delta}(q_0, xy^k z) = \widehat{\delta}(q_i, z) \in F$ by (c). Q.E.D.

The pumping lemma is our main tool to prove that languages are not regular.

Example 2.12. The language $L := \{0^k 1^k \,;\, k > 0\}$ is not regular.

[Suppose it was. Then it satisfies the pumping lemma by Theorem 2.11, i.e., there is some n such that it satisfies the pumping lemma with pumping number n. Consider the word $w = 0^n 1^n \in L$. Clearly, $|w| = 2n \geq n$, so the word can be pumped. This means that we can write $w = xyz$ with $|y| > 0$ and $|xy| \leq n$. By choice of w, this means that both x and y consist entirely of zeros. If we now pump down, we obtain that $xy^0z = xz \in L$, but this word contains $n - |y| < n$ many zeros and n many ones. Hence it's not in L: contradiction!]

Since the proof of the pumping lemma tells us that the pumping number is the number of states of the automaton accepting the language, it also gives us lower bounds on its size.

Example 2.13. Fix some positive number $n \in \mathbb{N}$. Then the language $L := \{0^n w \,;\, w \in \mathbb{W}\}$ is regular and there cannot be an automaton D with n or fewer states such that $\mathcal{L}(D) = L$.

[Towards a contradiction, let's assume that there is such an automaton. By the proof of Theorem 2.11, we get that L satisfies the pumping lemma with pumping number n. Consider the word $w = 0^n \in L$. Clearly, $|w| = n$, so the word can be pumped, in particular, pumped down. Since it consists entirely of zeros, we know that for $w = xyz$, the words x, y, and z also consist entirely of zeros and $xy^0z = xz$ is a sequence of $n - |y| < n$ zeros. Hence it's not in L: contradiction!]

Corollary 2.14. If D is an automaton with n states and there is a path from q to q', then there is a path from q to q' of length at most n.

Proof. As in the proof of the pumping lemma, if the path is longer, a state repeats, and thus, the loop from the first to the second occurrence of that repeating state can be removed to obtain a shorter path. Therefore, the shortest path must have length at most n. Q.E.D.

Since the pumping lemma is a very useful tool to prove that languages are not regular, it is quite natural to wonder whether the statement of the pumping lemma is equivalent to regularity, i.e., whether a language L is regular if and only if it satisfies the regular pumping lemma. The answer is "No" as we shall show now.

If $w \in \{0,1\}^*$ is a word that contains at least one zero, we write $\mathrm{tail}(w)$ for the number of ones in w that follow the last occurring zero. E.g., $\mathrm{tail}(0101111) = 4$. Let $X \subseteq \mathbb{N}$ be an arbitrary set of natural numbers (by Proposition 1.3, there are uncountably many of those). We define a language $L_X \subseteq \{0,1\}^*$ by $w \in L_X$ if w consists entirely of ones or if w has some zero, then $\mathrm{tail}(w) \in X$. Let us show that if $X \neq Y$, then $L_X \neq L_Y$: w.l.o.g., we can assume that there is some $n \in X \backslash Y$. Then $01^n \in L_X \backslash L_Y$. This shows that $X \mapsto L_X$ is an injection from the power set of $\mathbb{N}$ into the collection of languages of the form L_X, so there are uncountably many such languages.

Proposition 2.15. Every language L_X satisfies the (regular) pumping lemma.

Proof. We shall prove that it satisfies the pumping lemma with pumping number 2. Any word w with $|w| \geq 2$ starts either with 0 or 1.

Case 1. It starts with 0. Let $x = \varepsilon$, $y = 0$, and z such that $w = xyz = 0z$. Pumping up produces $0^k z$ (for $k > 1$), but clearly $\mathrm{tail}(0^k z) = \mathrm{tail}(0z) \in X$, so $0^k z \in L_X$. Pumping down produces z: if z still contains a 0, then $\mathrm{tail}(z) = \mathrm{tail}(0z) \in X$, so $z \in L_X$; if z contains no 0s, then $z \in L_X$ anyway.

Case 2. It starts with 1. Let $x = \varepsilon$, $y = 1$, and z such that $w = xyz = 1z$. If z does not contain any 0s, then all results of pumping y will result in a word without 0s, so they are all in L_X. If z contains a 0, then all results of pumping y will result in a word that has the same tail as $1z$, and hence they are all in L_X. Q.E.D.

Corollary 2.16. There are languages satisfying the (regular) pumping lemma that are not regular.

Proof. There are only countably many regular languages (by Proposition 1.14), but uncountably many languages satisfying the regular pumping lemma by Proposition 2.15. Q.E.D.

2.5 Closure properties

We shall now show that the class of regular languages is closed under all five closure properties listed in § 1.7: *Concatenation*, *Union*, *Intersection*, *Complement*, and *Difference*. Union and Concatenation were proved in Corollary 2.3.

Proposition 2.17. The class of regular languages is closed under complementation, intersection, and difference.

Proof. We are going to show closure under complementation; the other claims follow from Lemma 1.20 (a) & (c). Suppose that $L = \mathcal{L}(D)$ for some deterministic automaton $D = (\Sigma, Q, \delta, q_0, F)$. W.l.o.g., we can assume that q_0 is not in the range of δ. [Just add a new state q_0^* and let

$$\delta'(q,a) := \begin{cases} \delta(q,a) & \text{if } \delta(q,a) \neq q_0 \text{ and} \\ q_0^* & \text{otherwise.} \end{cases}$$

Then $(\Sigma, Q \cup \{q_0^*\}, \delta', q_0, F)$ accepts the same language as D and does not have q_0 in the range of its transition function.] Thus, let us assume that D has this property and define

$$\overline{D} := (\Sigma, Q, \delta, q_0, Q \backslash (F \cup \{q_0\}),$$

then we claim that $\mathcal{L}(\overline{D}) = \mathbb{W}^+ \backslash \mathcal{L}(D)$.

"$\subseteq$": Suppose $w \in \mathcal{L}(\overline{D})$, i.e., $q := \widehat{\delta}(q_0, w) \in Q \backslash (F \cup \{q_0\})$. This means that $w \neq \varepsilon$ (since $\widehat{\delta}(q_0, \varepsilon) = q_0$) and $\widehat{\delta}(q_0, w) \notin F$, so $w \notin \mathcal{L}(D)$.

"$\supseteq$": Suppose $\varepsilon \neq w$ is such that $w \notin \mathcal{L}(D)$, i.e., $\widehat{\delta}(q_0, w) \notin F$. Since $w \neq \varepsilon$, we know that $\widehat{\delta}(q_0, w) \neq q_0$ (by our assumption about the range of δ), so together, this implies that $w \in \mathcal{L}(\overline{D})$. Q.E.D.

There is an alternative proof for union and intersection that can be instructive in certain contexts. Given nonempty sets Q and Q', as well as $F \subseteq Q$ and $F' \subseteq Q'$, we let

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

We can now give a product construction of two automata: if $D = (\Sigma, Q, \delta, q_0, F)$ and $D' = (\Sigma, Q', \delta', q_0', F')$ are two automata, we define

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

This allows us to define product automata for intersection and union as follows:

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

Proposition 2.18. For any automata D and D', we have

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

Proof. By definition (and induction), $\widehat{\delta \times \delta'}(w, (q, q')) = (\widehat{\delta}(w, q), \widehat{\delta'}(w, q'))$. Therefore,

$$\begin{aligned} w \in \mathcal{L}(D \wedge D') &\iff \widehat{\delta \times \delta'}(w, (q_0, q_0')) \in F \wedge F' \\ &\iff (\widehat{\delta}(w, q_0), \widehat{\delta'}(w, q_0')) \in F \times F' \\ &\iff \widehat{\delta}(w, q_0) \in F \text{ and } \widehat{\delta'}(w, q_0') \in F' \\ &\iff w \in \mathcal{L}(D) \text{ and } w \in \mathcal{L}(D'), \end{aligned}$$

and similarly for $D \vee D'$. Q.E.D.

2.6 Regular expressions

We shall consider two more operations on languages, the *Kleene plus* and the *Kleene star operation*. If L is a language, we write

$$L^+ := \{w \,;\, \exists w_0 ... \exists w_n \in L(w = w_0 ... w_n)\},$$

i.e., L^+ is a finite concatenation of elements of L. Furthermore, we write

$$L^* := L^+ \cup \{\varepsilon\}.$$

Note that this notation clashes slightly with our earlier star-operation X^* which denoted the set of finite sequences of elements of X. The result L^* of applying the Kleene star operation to a language is not the set of finite sequences of words in L, but the set of their

concatenations. These two notions are closely related, but not quite identical. If there is any chance of confusion, we shall be explicit about what we mean.

Let Σ be an alphabet. Among the finite strings over the set $\Sigma \cup \{\varnothing, \varepsilon, (,), +, ^+, ^*\}$; we shall define the notion of *regular expressions over* Σ by recursion:[9]

(1) The symbol $\varnothing$ is a regular expression;

(2) the symbol ε is a regular expression;

(3) every $a \in \Sigma$ is a regular expression,

(4) if R and S are regular expressions, then $(R + S)$ is a regular expression;

(5) if R and S are regular expressions, then (RS) is a regular expression;

(6) if R is a regular expression, then R^+ is a regular expression;

(7) if R is a regular expression, then R^* is a regular expression;

(8) nothing else is a regular expression.

Note that construction steps (3) and (4) introduce a lot of parentheses that will turn out to be unnecessary since the corresponding operations on languages turn out to be associative. So, informally, we shall often drop some of these parentheses and write, e.g., $R + S$ instead of $(R + S)$, $R + S + T$ instead of $((R + S) + T)$ or $((R + (S + T))$, and $R(S + T)$ instead of $(R(S + T))$. We shall also assume that concatenation has higher binding priority than $+$ and write $RS + T$ for $(RS) + T$, or more accurately $((RS) + T)$.

We now associate languages to regular expressions by recursion:

(1) If $E = \varnothing$, then $\mathcal{L}(E) = \varnothing$;

(2) if $E = \varepsilon$, then $\mathcal{L}(E) = \{\varepsilon\}$;

(3) if $E = a$ for $a \in \Sigma$, then $\mathcal{L}(E) = \{a\}$;

(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 $\mathcal{L}((RS)) = \mathcal{L}(R)\mathcal{L}(S)$;

(6) if R is a regular expression, then $\mathcal{L}(R^*) = \mathcal{L}(R)^*$;

(7) if R is a regular expression, then $\mathcal{L}(R^+) = \mathcal{L}(R)^+$.[10]

Proposition 2.19. If R is a regular expression, then $\mathcal{L}(R)$ is an essentially regular language.

Proof. This follows inductively via the recursive definition of regular expressions. Clearly, $\varnothing$, $\{\varepsilon\}$, and $\{a\}$ are essentially regular languages and we have proved in Corollary 1.26 and Proposition 2.2 that the regular languages are closed under union and concatenation, respectively. This implies by Proposition 1.28 that essentially regular languages are closed under union and concatenation as well.

[9]Note that $\varnothing$, ε, $+$, $^+$, and * are *symbols* here, not objects or operations. They will, however, be *interpreted* as the empty set, the empty sequence, the operation of union, the Kleene star operation, and the Kleene plus operation, respectively.

[10]These equations are nice examples of the issue raised in footnote 9: e.g., in the equation $\mathcal{L}(R^+) = \mathcal{L}(R)^+$, the first $^+$ is a symbol that is part of the regular expression R^+, whereas the second $^+$ is the operation of Kleene plus applied to the language $\mathcal{L}(R)$.

So, we only need to show that the essentially regular languages are closed under the Kleene star and plus operations. Since $\mathcal{L}(L)^* = \mathcal{L}(L)^+ \cup \{\varepsilon\}$, it is enough to show that if L is regular, then so is L^+.

Let $G = (\Sigma, V, P, S)$ be a regular grammar for L. Let $P^+ := P \cup \{A \to aS\,;\, A \to a \in P\}$ and $G^+ := (\Sigma, V, P^+, S)$. Note that G^+ is a regular grammar. We claim that $\mathcal{L}(G^+) = L^+$.

For "$\supseteq$", let $w = w_0...w_n \in L^+$ where $w_0, ..., w_n \in L$. We prove the claim by induction on n. If $n = 0$, then $w = w_0 \in L = \mathcal{L}(G) \subseteq \mathcal{L}(G^+)$. Suppose the claim holds for n and $w = w_0...w_nw_{n+1}$. By induction hypothesis, we have $S \xrightarrow{G^+} w_0...w_n$ and by assumption, we have that $S \xrightarrow{G} w_{n+1}$. By Lemma 2.1, we know that the last rule application in the derivation of $w_0...w_n$ is a rule of the form $A \to a$; replacing it with the rule $A \to aS \in P^+$, we obtain $S \xrightarrow{G^+} w_0...w_nS$. Prefixing $w_0...w_n$ to every string in the derivation of w_{n+1}, we obtain $w_0...w_nS \xrightarrow{G} w_0...w_nw_{n+1} = w$. Since $P \subseteq P^+$, these two derivations yield $S \xrightarrow{G^+} w$.

For "$\subseteq$", using Proposition 1.27, we may assume w.l.o.g. that P is ε-adequate, i.e., that it does not contain any instances of S on the right-hand side of its rules. We shall show the claim by induction of the number of occurrences of S in the derivation $S \xrightarrow{G^+} w$. Note the only rules that are in $P^+\backslash P$ introduce an S and all rules in P remove S from the current string (by ε-adequacy). So, the number of occurrences of S counts how many times one of the additional rules is used in the derivation. If there is exactly one occurrence of S (the start symbol at the beginning), we have that $S \xrightarrow{G} w$, so $w \in \mathcal{L}(G) = L \subseteq L^+$. Suppose that we have shown the claim for derivations with n occurrences of S and let $S \xrightarrow{G^+} w$ be a derivation with $n + 1$ occurrences of S. Then $S \xrightarrow{G^+} vS \xrightarrow{G^+} w$. The final production rule of the first part is of the form $A \to aS \in P^+$ whence $A \to a \in P$. Replacing the former with the latter, we obtain $S \xrightarrow{G^+} v$ and this is now a derivation with n occurrences of S. Hence, $v \in L^+$ by induction hypothesis. By Lemma 2.1, all strings in the remainder of the derivation $vS \xrightarrow{G^+} w$ are prefixed by v and we can remove them to obtain $S \xrightarrow{G^+} u$ with $w = vu$. This derivation has only one occurrence of S, so we have $S \xrightarrow{G} u$, and hence $u \in L$. Thus $w = vu \in L^+L \subseteq L^+$. Q.E.D.

The proof of Proposition 2.19 tells us that the class of regular languages has another closure property: it is closed under the Kleene plus operation.

While we are not going to prove this in this course, the converse of Proposition 2.19 is true: regular expressions describe exactly the essentially regular languages. There are many algorithms to transform an automaton into a regular expression; the oldest is *Kleene's algorithm*.[11] On Example Sheet # 2, we shall look at several special cases transforming regular grammars into regular expressions.

[11] Cf. S. C. Kleene. Representation of events in nerve nets and finite automata. In: C. E. Shannon, J. McCarthy (eds.). Automata Studies. Annals of Mathematics Studies, Vol. 34, Princeton University Press, 1956; pp. 3–42.

2.7 Minimisation of deterministic automata

If $D = (\Sigma, Q, \delta, q_0, F)$ is a deterministic automaton, we call a state $q \in Q$ *inaccessible* if there is no word w such that $\widehat{\delta}(q_0, w) = q$. We call two states $q, q' \in Q$ *indistinguishable* if for all words w, we have that

$$\widehat{\delta}(q, w) \in F \iff \widehat{\delta}(q', w) \in F.$$

A word w such that $\widehat{\delta}(q, w) \in F$ and $\widehat{\delta}(q', w) \notin F$ or vice versa is said to *distinguish* q and q'. Given $q, q' \in Q$ and $a \in \Sigma$ and $\delta(q, a)$ and $\delta(q', a)$ are distinguished by a word w, then q and q' are distinguished by the word aw. If $f : Q \to Q'$ is a homomorphism from an automaton D to an automaton D', then if $p, q \in Q$ are distinguishable, then $f(p) \neq f(q)$. Furthermore, if $q' \in Q'$ is accessible, then $q' \in \operatorname{ran}(f)$.

We write $q \sim q'$ if they are indistinguishable. Note that $\sim$ is an equivalence relation on Q, i.e., reflexive, symmetric, and transitive. We write $[q]$ for the $\sim$-equivalence class of q. We define the *quotient automaton*

$$D/{\sim} := (\Sigma, Q/{\sim}, [\delta], [q_0], [F])$$

where $[\delta]([q], a) := [\delta(q, a)]$ and $[F] := \{[q]\,;\, q \in F\}$. By induction, we get that $\widehat{[\delta]}([q], w) = [\delta(q, w)]$.

Proposition 2.20. The quotient automaton is well defined and no two of its states are indistinguishable.

Proof. Let $q \sim q' \in Q$ and consider $\delta(q, a)$ and $\delta(q', a)$. As mentioned, if they are distinguished by a word, then so are q and q'. Therefore, $\delta(q, a) \sim \delta(q', a)$.

Towards the second claim, we know that since $\{w\,;\, \widehat{[\delta]}([q], w)\} = \{w\,;\, \widehat{\delta}(q, w)\}$, we have that $[q] \sim [q']$ if and only if $q \sim q'$, i.e., $[q] = [q']$. Q.E.D.

Proposition 2.21. For every deterministic automaton D, we have $\mathcal{L}(D) = \mathcal{L}(D/{\sim})$.

Proof. Clearly, the quotient map $q \mapsto [q]$ is a homomorphism and the result follows from Proposition 2.5. Q.E.D.

We call an automaton *irreducible* if it has neither inaccessible states nor indistinguishable distinct states.

Lemma 2.22. If f is a homomorphism between automata D and D', then

(a) if D is irreducible, then f is an injection;

(b) if D' is irreducible, then f is a surjection; and

(c) if both are irreducible, then f is a bijection.

Proof. This follows directly from the observations in § 2.2: if $f(p) = f(q)$, then p and q must be indistinguishable; if $q' \notin \operatorname{ran}(f)$, then q' must be inaccessible. Q.E.D.

Theorem 2.23. For every deterministic automaton D, there is an irreducible automaton I with at most as many states as D such that $\mathcal{L}(D) = \mathcal{L}(I)$.

Proof. Clearly, if q is an accessible state, then all states of the form $\delta(q, a)$ with $a \in \Sigma$ are accessible. As a consequence, if $A \subseteq Q$ denotes the accessible states, then if $\delta^* := \delta \restriction A \times \Sigma$, we have that $\delta^* : A \times \Sigma \to A$. Thus, $D^* := (A, \delta^*, q_0, F \cap A)$ is a deterministic automaton. Clearly, if $w \in \mathcal{L}(D)$, then $\widehat{\delta}(q_0, w) \in F \cap A$, so $w \in \mathcal{L}(D^*)$ if and only if $w \in \mathcal{L}(D)$.

We now consider $I := D^*/\!\sim$, the quotient automaton of D^*. By the previous argument and Proposition 2.21, we obtain

$$w \in \mathcal{L}(D) \iff w \in \mathcal{L}(D^*) \iff w \in \mathcal{L}(D^*/\!\sim) = \mathcal{L}(I).$$

Clearly, the quotient construction preserves the property that there are no inaccessible states (since if $\widehat{\delta}(q_0, w) = q$, then $\widehat{[\delta]}([q_0], w) = [q]$), so I has all the desired properties. Q.E.D.

We shall see now that up to isomorphism, there is a unique irreducible automaton.

Theorem 2.24. If I and I' are two irreducible automata such that $\mathcal{L}(I) = \mathcal{L}(I')$, then there is a homomorphism from I to I'.

Proof. Let $I := (\Sigma, Q, \delta, q_0, F)$ and $I' := (\Sigma, Q', \delta', q_0', F')$. As usual, w.l.o.g., we can assume that $Q \cap Q' = \varnothing$. The notion of indistinguishability is an equivalence relation on both Q and Q'; we now extend it to $Q \cup Q'$ and say that if $q \in Q$ and $q' \in Q'$, then $q \sim q'$ if

$$\{w \,;\, \widehat{\delta}(q, w) \in F\} = \{w \,;\, \widehat{\delta'}(q', w) \in F'\};$$

note that the new relation is an equivalence relation on $Q \cup Q'$. We use the same terminology as before, e.g., we say that "w distinguishes q and q'" if $\widehat{\delta}(q, w) \neq \widehat{\delta'}(q', w)$. By the assumption that $\mathcal{L}(I) = \mathcal{L}(I')$, we have that the two start states are not distinguished by any word, i.e.,

$$\{w \,;\, \widehat{\delta}(q_0, w) \in F\} = \mathcal{L}(I) = \mathcal{L}(I') = \{w \,;\, \widehat{\delta'}(q_0', w) \in F'\}.$$

Claim 1. Every state in Q is indistinguishable from some state in Q'.

[Since I does not have any inaccessible states, every state in Q is reachable from q_0. We let $\mathrm{sp}(q)$ be the length of the shortest path from q_0 to q and prove the claim by induction on $\mathrm{sp}(q)$. Clearly, $\mathrm{sp}(q) = 0$ if and only if $q = q_0$; as mentioned above, we have $\{w \,;\, \widehat{\delta}(q_0, w) \in F\} = \{w \,;\, \widehat{\delta'}(q_0', w) \in F'\}$. Let us assume that $\mathrm{sp}(q) = k + 1$ and find $p \in Q$ and $a \in \Sigma$ such that $\mathrm{sp}(p) = k$ and $\delta(p, a) = q$. By induction hypothesis, there is some $p' \in Q'$ such that $\{w \,;\, \widehat{\delta}(p, w) \in F\} = \{w \,;\, \widehat{\delta'}(p', w) \in F'\}$. Let $q' := \delta'(p', a)$. Then if w distinguishes q and q', then aw distinguishes p and p', so $q \sim q'$.]

Claim 2. No two states in Q are indistinguishable from the same state in Q'. Similarly, no two states in Q' are indistinguishable from the same state in Q.

[If $p \sim q' \sim q$, then by transitivity, we have that $p \sim q$, but by irreducibility, then $p = q$. The second claim follows by symmetry.]

Thus, we can define $f(q)$ to be the unique $q' \in Q'$ such that $q \sim q'$. Claims 1 and 2 imply that this is an injection from Q to Q'. Let us check that it is a homomorphism:

(i) Let $q \sim q'$, $\delta(q, a) = p$, and $p \sim p'$. We need to show that $\delta'(q', a) \sim p'$. Suppose they are not equivalent, say, there is a w such that

$$\widehat{\delta'}(\delta'(q', a), w) = \widehat{\delta'}(q', aw) \in F' \text{ and } \widehat{\delta'}(p', w) \notin F'.$$

Since $p \sim p'$, we have that $\widehat{\delta}(p, w) \notin F$ and thus $\widehat{\delta}(q, aw) \notin F$. But then aw distinguishes between q and q'.

(ii) By definition, $q_0 \sim q_0'$.

(iii) If $q \in F$ and $q' \notin F'$, then ε distinguishes between them, so $q \not\sim q'$.

Q.E.D.

Corollary 2.25. Any two irreducible automata that accept the same language are isomorphic.

Proof. Follows directly from Theorem 2.24 and Lemma 2.22. Q.E.D.

This also means that all irreducible automata producing the language L have the same size and any automaton producing L must be at least as large in terms of its number of states. Thus, the (up to isomorphism) unique irreducible automaton for the language L is minimal in size and we call it the *minimal automaton*.

2.8 Decision problems

As mentioned in § 1.6, we shall consider the word problem, the emptiness problem, and the equivalence problem for our classes of languages. In Theorem 1.19, we already solved the word problem for regular languages positively. Note that the connection to deterministic automata makes this particularly obvious since a deterministic automaton *is* an algorithm and therefore the automaton provides the evidence that whether $w \in \mathcal{L}(D)$ can be checked by an algorithm.

The positive solution to the emptiness problem follows easily from the pumping lemma:

Corollary 2.26. If L satisfies the regular pumping lemma with pumping number n, then if $L \neq \varnothing$, then there is a word $w \in L$ with $|w| < n$.

Proof. If $|w| \geq n$ and $w \in L$, then w can be pumped down. In particular, w cannot be the shortest word in L. Since $L \neq \varnothing$, the language L has a shortest word which then must have length smaller than n. Q.E.D.

Corollary 2.27. There is an algorithm that on input of a regular grammar G determines whether $\mathcal{L}(G) = \varnothing$.

Proof. We know that there is a deterministic automaton D such that $\mathcal{L}(G) = \mathcal{L}(D)$ and the number of states of D is at most 2^{m+1} where m is the number of nonterminal symbols of G (cf. Example Sheet # 1). Thus, $\mathcal{L}(G)$ satisfies the regular pumping lemma with pumping number 2^{m+1}. Now check for every single word w of length at most 2^{m+1} (there are only finitely many such words) whether $w \in \mathcal{L}(D)$ or not; if it is, then $\mathcal{L}(G)$ is non-empty; if none of them are, then $\mathcal{L}(G) = \varnothing$ by Corollary 2.26. Q.E.D.

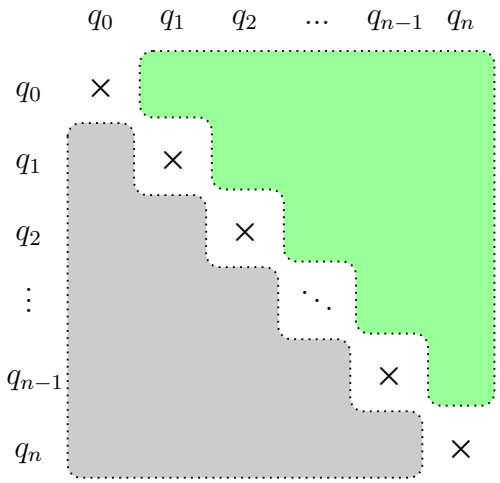
The positive answer to the equivalence problem will follow from our construction of the minimal automaton as a quotient of the original automaton. We need to check that the construction steps that we used can be done algorithmically. Given D, we only need to find an irreducible automaton as a quotient; this will be unique up to isomorphism by Corollary 2.25.

Proposition 2.28. There is an algorithm that determines which states of an automaton are inaccessible.

Proof. Let $D = (\Sigma, Q, \delta, q_0, F)$ and $n := |Q|$. By Corollary 2.14, a state q is inaccessible if and only if there is no word w of length $\leq n$ such that $\widehat{\delta}(q_0, w) = q$. Since there are finitely many such words, we can just check $\widehat{\delta}(q_0, w)$ for all such words to determine which states are accessible; the remaining states must be inaccessible. Q.E.D.

Proposition 2.29. There is an algorithm that determines whether two states of an automaton are equivalent.

Proof. We determine whether the states are indistinguishable; from this, we can easily determine equivalence. This algorithm is known as the *table filling algorithm*. We write $Q \times Q$ as a table; note that due to the fact that indistinguishability is an equivalence relation, we only need to fill half of the table, so we can ignore the lower left triangle.



In the first step of the algorithm, we check all relevant pairs (q, q') and mark them as distinguished if $q \in F$ and $q' \notin F$ or vice versa. These states are distinguished by ε, and so we can write ε as the witness into the table.

In subsequent steps, we check every pair (q, q') that is not yet marked as follows: for every $a \in \Sigma$, we let $q_* := \delta(q, a)$ and $q'_* := \delta(q', a)$ and check whether the pair (q_*, q'_*) is already marked. If it is marked by w, then we mark (q, q') by aw.

At the end of each step of the algorithm, we check whether a new pair was marked or not. If not, then we terminate the algorithm; otherwise, we go into the next step. Note that since only finitely many table entries can be filled, this algorithm will eventually terminate.

Claim. Two states q and q' are indistinguishable if and only if (q, q') is unmarked at the end of the algorithm.

[For the forward direction, let a pair (q, q') is marked by w, then by construction and induction, $\widehat{\delta}(q, w) \in F$ and $\widehat{\delta}(q', w) \notin F$ or vice versa, so q and q' are distinguished by w.

Towards proving the backward direction, assume towards a contradiction that there is a pair that can be distinguished by a word and is not marked by the end of the algorithm. Let's call such a pair a *bad pair*. Each bad pair has a distinguishing word that witnesses that it is bad. Find a bad pair (q, q') with a distinguishing word w of minimal length, i.e., no other bad pair can have a shorter distinguishing word. Note furthermore that $|w| > 0$ since pairs that are distinguished by ε are marked by definition of the table-filling algorithm and so can't be a bad pair. Thus, let a be the first letter of w, i.e., $w = av$. Then consider $q_* := \delta(q, a)$ and $q'_* := \delta(q', a)$. Clearly, q_* and q'_* are distinguished by v, since

$$\widehat{\delta}(q_*, v) = \widehat{\delta}(\delta(q, a), v) = \widehat{\delta}(q, av) = \widehat{\delta}(q, w) \text{ and}$$
$$\widehat{\delta}(q'_*, v) = \widehat{\delta}(\delta(q', a), v) = \widehat{\delta}(q', av) = \widehat{\delta}(q', w).$$

However, q_* and q'_* cannot be marked: if they were, then in the step after the pair (q_*, q'_*) is marked in the algorithm, (q, q') would be marked. So, (q_*, q'_*) is a bad pair, but it has a distinguishing word of length $|w|-1$ in contradiction to the minimality assumption.] Q.E.D.

Theorem 2.30. Given two deterministic automata, there is an algorithm to determine whether they accept the same language. In other words, the equivalence problem for regular grammar has a positive solution.

Proof. Using Propositions 2.28 & 2.29, we can produce the minimal automata for each of the two given automata. Now we only need to determine whether they are isomorphic: note that this can be done algorithmically. If the minimal automata are of different sizes, the answer is "no"; otherwise, they have the same number n of states and there are at most n^n functions that need to be checked to see whether they are an isomorphism.

Given a regular grammar, transform it to a deterministic automaton via the algorithms in the proofs of Theorems 2.7 & 2.8 and then apply the first statement to check equivalence. Q.E.D.

3 Context-free languages

We remember that a grammar is *context-free* if all of its rules are of the form $A \to \alpha$ where $A \in V$ and $\alpha \in \Omega^* \backslash \{\varepsilon\}$. We can quite easily see some non-regular languages are context-free.

Example 3.1. The grammar consisting of the rules $S \to 0S1$ and $S \to 01$ produces the language $\{0^k 1^k \,;\, k > 0\}$, our standard example of a non-regular language.

[Clearly, every derivation is just the application of some (possibly none) applications of the rule $S \to 0S1$ followed by a final application of the rule $S \to 01$. This implies the claim.]

While more general than regular languages, the specific form of context-free grammars still gives us a great deal of control over its productions. In the next sections, we shall exploit this control to understand better how context-free languages work.

3.1 Parse trees

We call a subset $T \subseteq \mathbb{N}^*$ a *(finitely branching) tree* if it is closed under initial segments (i.e., if $t \in T$ and $s \subseteq t$, then $s \in T$) and for each $t \in T$ there is a natural number n such that $tk \in T$ if and only if $k < n$. In this case, we say that t has n *successors* or that t is n*-branching*. An element $t \in T$ that has no successors is called a *leaf* (or *terminal node*). The sequence ε is contained in every non-empty tree and is called the *root* of the tree. The elements of length k in a tree form its k*th level*. If T is a finite tree, then there is a maximal k such that T has an element on the kth level. This number is called the *height* of T. If T is a tree and $t \in T$ is the kth level (i.e., $|t| = k$), then the corresponding *branch* through T is the sequence $\{t\restriction m \,;\, m \leq k\}$; it is a sequence of nodes of T of length $k+1$.

If T is a tree, we can define a partial order $<$ called the *left-to-right order* as follows:

$$s < t :\iff s \neq t \text{ and if } k \text{ is least such that } s(k) \neq t(k), \text{ then } s(k) < t(k).$$

If X is a set of nodes on the same level of a tree T, then $<$ is a total order on X; similarly, if X is a set of leaves of T, then $<$ is a total order on X. In particular, the leaves of a tree are totally ordered from left to right via the order $<$. This is a partial order as wa, w are not comparable.

If $G = (\Sigma, V, P, S)$ is a context-free grammar and $A \in V$, we say that a pair $\mathbf{T} := (T, \ell)$ is a *G-parse tree starting from A* if T is a finite finitely branching tree and $\ell : T \to \Omega$ is a function satisfying

(a) $\ell(\varepsilon) = A$,

(b) if $\ell(t) \in \Sigma$, then t is a leaf in T, and

(c) if $\ell(t) = B \in V$ and t is $n+1$-branching, then there is a rule $B \to x_0 ... x_n \in P$ such that $\ell(tk) = x_k$ for all $k < n+1$.

Note this defn allows variables to be leaf nodes.

Since a parse tree is finite, it has finitely many leaves which are totally ordered by the left-to-right order. Let $t_0 < t_1 < ... < t_m$ be the leaves of $\mathbf{T}$; we then write $\sigma_{\mathbf{T}} := \ell(t_0)...\ell(t_m)$ for the *string parsed by* $\mathbf{T}$. As a finite tree, $\mathbf{T}$ has a last level, and we can graphically extend all leaves to that level; if we do so, then the left-to-right order corresponds to reading the string

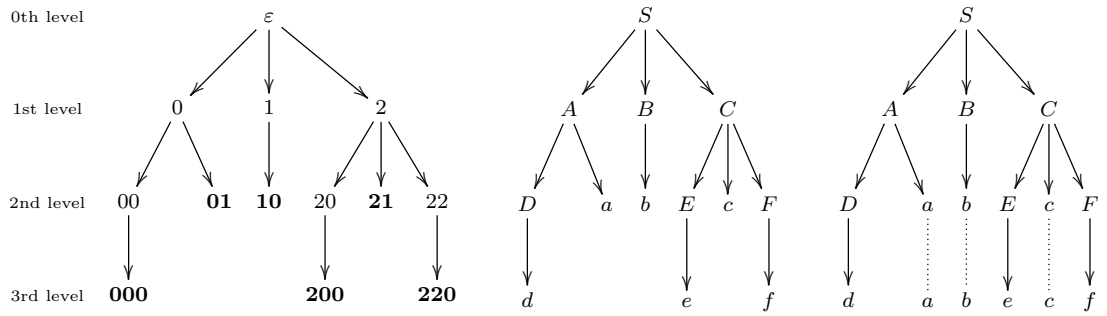


Figure 2: *Left.* A finitely branching tree; leaves are marked in boldface font. *Middle.* The same tree labelled to form a G-parse tree. *Right.* The same parse tree with the leaves extended to the final level to highlight that the parse tree parses the word $dabecf$.

from left to right on the last level. This is depicted in Figure 2. If we have a branch in a parse tree (starting from A) of height k, then it is a sequence of length $k+1$ and its labels form a sequence of $k+1$ symbols.

Proposition 3.2. If G is a context-free grammar, then $w \in \mathcal{L}(G)$ if and only if there is a G-parse tree $\mathbf{T}$ starting from S such that $w = \sigma_{\mathbf{T}}$.

Proof. A sequence $(\mathbf{T}_0, ..., \mathbf{T}_n)$ of G-parse trees is called *derivative* if

(a) $T_0 = \{\varepsilon\}$ and $\ell_0(\varepsilon) = S$,

(b) for each $i < n$, the parse tree $\mathbf{T}_{i+1} = (T_{i+1}, \ell_{i+1})$ is obtained by taking a terminal node t of T_i with $\ell_i(t) \in V$ and a rule $\ell_i(t) \to x_0...x_m \in P$, adding $m+1$ successors to t and labelling them by $\ell_{i+1}(tk) = x_k$.

Clearly, there is a one-to-one correspondence between G-derivations and derivative sequences of G-parse trees: a derivation $S = \sigma_0 \xrightarrow{G}_1 \sigma_1 \xrightarrow{G}_1 ... \xrightarrow{G}_1 \sigma_n$ uniquely defines a derivative sequence of G-parse trees $(\mathbf{T}_0, ..., \mathbf{T}_n)$ such that $\sigma_{\mathbf{T}_i} = \sigma_i$ and vice versa. This shows the direction "⇒" of our claim.

For the other direction, let $\mathbf{T}$ be a G-parse tree starting from S with $\sigma_{\mathbf{T}} = w$. We construct a derivative sequence of subtrees of $\mathbf{T}$, starting with $\mathbf{T}_0 = (\{\varepsilon\}, \ell{\restriction}\{\varepsilon\})$. In each step of the construction, assume that $\mathbf{T}_i$ was already constructed and find a $t \in T_i$ that is a leaf in T_i, but not in T. Form T_{i+1} by adding the T-successors of t to T_i. If we cannot find a leaf in T_i that is no leaf in T, we terminate the construction.

We claim that the construction terminates when $T_i = T$. Suppose it's not, then there is a $t \in T \backslash T_i$. Consider the branch leading to t in T: there must be a maximal element of T_i on this branch (note that $\varepsilon \in T_i$): by construction (whenever we add successors, we add all successors), that is a leaf in T_i, but no leaf in T. So, the construction has not terminated in contradiction with the assumption. Q.E.D.

If $\mathbf{T}$ is any G-parse tree and $t \in T$ is not a leaf, we can define the *subtree at* t by $\mathbf{T}_t := (T_t, \ell_t)$ with $T_t := \{s\,;\, ts \in T\}$ and $\ell_t(s) := \ell(ts)$. Clearly, if $\ell(t) = A$, then $\mathbf{T}_t$ is a

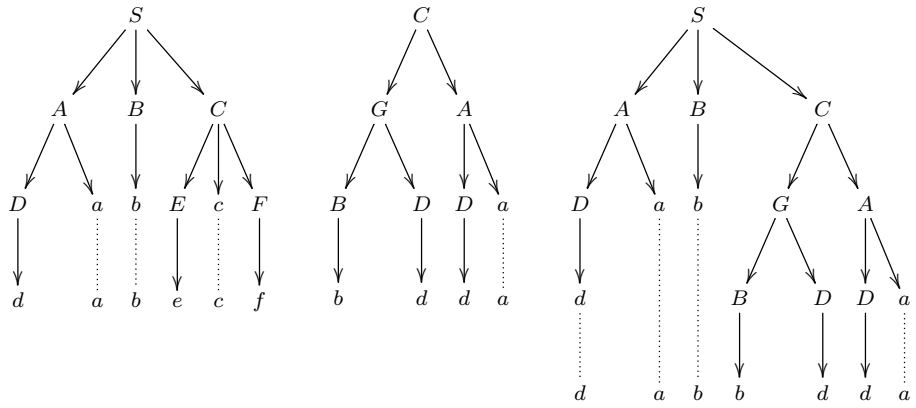


Figure 3: A G-parse tree $\mathbf{T}$ (left), a G-parse tree $\mathbf{T}'$ starting from C, and the result of grafting $\mathbf{T}'$ into the unique node t labelled C in $\mathbf{T}$. Note that $\sigma_{\mathbf{T}} = dabecf$, $\sigma_{\mathbf{T}_t} = ecf$, $\sigma_{\mathbf{T}'} = bdda$, and that $bdda$ replaces ecf in the word parsed by the result of the graft, i.e., $dabbdda$.

G-parse tree starting from A. Moreover, the left-to-right ordering of the leaves means that the leaves of $\mathbf{T}$ are a consecutive subsequence of the leaves in $\mathbf{T}$: therefore, there are strings τ and τ' such that $\sigma_{\mathbf{T}} = \tau\sigma_{\mathbf{T}_t}\tau'$. The string τ corresponds to all of the leaves of $\mathbf{T}$ that are to the left of the leaves of $\mathbf{T}_t$ in the left-to-right order; similarly, the string τ' corresponds to all of the leaves to the right of those of $\mathbf{T}_t$.

Note that if $\mathbf{T}$ and $\mathbf{T}'$ are G-parse trees and $t \in T$ with $\ell(t) = A$ and $\mathbf{T}'$ starts from A, then we can *graft* $\mathbf{T}'$ into $\mathbf{T}$ as follows: we remove $\mathbf{T}_t$ and replace it by $\mathbf{T}'$. By definition, this results in a G-parse tree. More formally, we define $\mathrm{graft}(\mathbf{T}, t, \mathbf{T}') := (S, \ell^*)$ with $S = \{s \in T\,;\, t \not\sqsubseteq s\} \cup \{ts\,;\, s \in T'\}$ and

$$\ell^*(s) := \begin{cases} \ell(s) & \text{if } t \not\sqsubseteq s \text{ and} \\ \ell'(u) & \text{if } s = tu \text{ for some } u \in T'. \end{cases}$$

In terms of the parsed strings, grafting a tree $\mathbf{T}'$ into the position of t in $\mathbf{T}$ corresponds to removing the substring $\sigma_{\mathbf{T}_t}$ from $\sigma_{\mathbf{T}}$ and replacing it with $\sigma_{\mathbf{T}'}$. This can be seen in Figure 3.

3.2 Chomsky normal form

We say that a grammar $G = (\Sigma, V, P, S)$ is in *Chomsky normal form* if all of its production rules are either of the form $A \to BC$ for $A, B, C \in V$ or of the form $A \to a$ for $A \in V$ and $a \in \Sigma$. Clearly, a grammar in Chomsky normal form is context-free. Moreover, the parse trees of these grammars are particularly nice: all nodes are either binary branching with two non-leaves as successors or not branching with a leaf as successor.

Lemma 3.3. If $G = (\Sigma, V, P, S)$ is a grammar in Chomsky normal form and $w \in \mathcal{L}(G)$ with $|w| = n$, then any G-derivation of w has length $2n - 1$.

Proof. If $\sigma \in \Omega^*$ is a string, write $\mathrm{v}(\sigma)$ for the number of variables in σ. Let's call a rule of the form $A \to BC$ *binary* and a rule of the form $A \to a$ *unary*. A binary rule increases both $|\sigma|$ and $\mathrm{v}(\sigma)$ by 1; a unary rule keeps $|\sigma|$ the same and decreases $\mathrm{v}(\sigma)$ by 1. Since $|S| = \mathrm{v}(S) = 1$,

we need $n-1$ applications of a binary rule to reach length n; these $n-1$ applications of binary rules will increase the number of variables by $n-1$, i.e., to $1+(n-1)=n$. Since w is a word and has no variables, we need n applications of a unary rule to ensure $\mathrm{v}(w)=0$. Together, this shows that any G-derivation consists of $n-1$ many applications of a binary rule and n many applications of a unary rule, i.e., has length $2n-1$. Q.E.D.

Lemma 3.4. If G is a grammar in Chomsky normal form, $\mathbf{T}$ a G-parse tree of height $h+1$, and $\sigma_{\mathbf{T}} = w \in \mathbb{W}$, then $|w| \leq 2^h$.

Proof. By definition, $|w|$ is the number of leaves in $\mathbf{T}$. Parse trees for grammars in Chomsky normal form are at most binary branching. The full binary tree of height $h+1$ has 2^{h+1} many leaves. Every rule in a Chomsky normal form grammar is either binary and does not produce letters or unary and produces a single letter. So, if $w \in \mathbb{W}$, then the parse tree must have at least $|w|$ many unary rule applications. Each unary rule application in $\mathbf{T}$ reduces the number of leaves by at least one. As a consequence, we have that $|w| \leq 2^{h+1} - |w|$, whence $|w| \leq 2^h$. Q.E.D.

Theorem 3.5 (Chomsky)**.** For every context-free grammar $G = (\Sigma, V, P, S)$, there is a grammar in Chomsky normal form G' such that $\mathcal{L}(G) = \mathcal{L}(G')$.

In order to prove Theorem 3.5, we need to provide some technical lemmas. We call a context-free production rule $A \to \alpha$ a *problematic production* if $|\alpha| \geq 2$ and α contains letters; we call it a *unit production* if α is just a single variable, i.e., the rule is of the form $A \to B$.

Lemma 3.6. If $G = (\Sigma, V, P, S)$ is any context-free grammar, then there is a context-free grammar G' that contains no problematic productions such that $\mathcal{L}(G) = \mathcal{L}(G')$.

Proof. Fix $G = (\Sigma, V, P, S)$. We use the ideas from the proof of Lemma 1.22: for each $a \in \Sigma$, we introduce a new variable X_a. For $\alpha \in \Omega^*$, let $X(\alpha)$ be the string α which each occurrence of a letter replaced by the corresponding new variable. Let $V' := V \cup \{X_a \,;\, a \in \Sigma\}$,

$$P' := \{A \to a \,;\, A \to a \in P\} \cup \{A \to X(\alpha) \,;\, A \to \alpha \in P, |\alpha| \geq 2\} \cup \{X_a \to a \,;\, a \in \Sigma\}, \text{ and}$$

$G' = (\Sigma, V', P', S)$. Then G' produces the same language as G and has no problematic productions. Q.E.D.

We call a grammar *unit closed* if for any unit production $A \to B \in P$ and any production $B \to \alpha \in P$, we also have $A \to \alpha \in P$.

Lemma 3.7. If G is any context-free grammar, then there is a unit closed grammar G' with $\mathcal{L}(G) = \mathcal{L}(G')$.

Proof. Form the unit closure by iteratively adding $A \to \alpha$ if it was not in P already. (Note that it's not necessarily enough to do this once: if $A \to B, B \to C, C \to \alpha \in P$, then the first step will add $B \to \alpha$ to the set of productions, but only the second step will add $A \to \alpha$. However, the number of new rules to be added is bounded by $|V||P|$.) Clearly, this does not change the language. Q.E.D.

Lemma 3.8. If $G = (\Sigma, V, P, S)$ is any context-free unit closed grammar, then removing all unit productions from it does not change the language.

Proof. Clearly, if G' is G with the unit productions removed, then $\mathcal{L}(G') \subseteq \mathcal{L}(G)$, so we need to show the other direction. We prove that by showing that any G-derivation that uses a unit production can be shortened. This means that the shortest G-derivation for a word cannot use unit productions and thus is a G'-derivation.

Let

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

where $\alpha A\beta \xrightarrow{G}_1 \alpha B\beta$ is the final unit production that occurs in the derivation. Since B is a variable, does not occur in w, and G is context-free, we know that there is some rule $B \to \zeta$ applied to B in the last part of the derivation. Let us write

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

where $\gamma B\delta \xrightarrow{G}_1 \gamma\zeta\delta$ is the first rule applied to that instance of B after the use of the unit production.

By our assumptions (and because G is context-free), all derivations between the application of $A \to B$ and the application of $B \to \zeta$ in that derivation are independent of which symbol is in place of the B, so we also have $\alpha A\beta \xrightarrow{G} \gamma A\delta$ with the very same derivation (i.e., it has precisely the same length as $\alpha B\beta \xrightarrow{G} \gamma B\delta$).

We know that both $A \to B$ and $B \to \zeta$ are in P, so by unit closure, we also have $A \to \zeta \in P$. Now, we put the various parts together and get

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

which is a production that is one step shorter than the one in (+). This proves our claim and thus the lemma. Q.E.D.

Lemma 3.9. Let $G = (\Sigma, V, P, S)$ be a context-free grammar and $A \to \alpha = A_0 ... A_n \in P$. Assume that $V' = V \cup \{X_0, ..., X_{n-2}\}$ where the X_i are new variables not occurring in V,

$$P_{A\to\alpha} := \{A \to A_0X_0, X_0 \to A_1X_1, ..., X_{n-3} \to A_{n-2}X_{n-2}, X_{n-2} \to A_{n-1}A_n\},$$

$P' := P \backslash \{A \to \alpha\} \cup P_{A\to\alpha}$, and $G' = (\Sigma, V', P', S)$. Then $\mathcal{L}(G) = \mathcal{L}(G')$.

Proof. Clearly, $\mathcal{L}(G) \subseteq \mathcal{L}(G')$. For the other direction: if a G'-derivation of a word $w \in \mathbb{W}$ uses any of the rules in $P_{A\to\alpha}$, they have to be used in the order given since the variables X_i do not show up in any other rules: thus, X_0 has to appear first, is rewritten by $X_0 \to A_1X_1$, etc., until the rule $X_{n-2} \to A_{n-1}A_n$ removes the new variables. The fact that G is context-free means that any other rule applications between the rules of $P_{A\to\alpha}$ can be moved before or after the cycle. [E.g., if

$$S \xrightarrow{G} \alpha A \beta \xrightarrow{G'}_1 \alpha A_0 X_0 \beta \xrightarrow{G} \gamma X_0 \delta \xrightarrow{G'}_1 \gamma A_1 A_2 \delta \xrightarrow{G} w,$$

then we have by context-freeness that $\alpha A_0 \xrightarrow{G} \gamma$ and $\beta \xrightarrow{G} \delta$, and thus

$$S \xrightarrow{G} \alpha A \beta \xrightarrow{G} \alpha A_0 A_1 A_2 \beta \xrightarrow{G} \gamma A_1 A_2 \beta \xrightarrow{G} \gamma A_1 A_2 \delta \xrightarrow{G} w.]$$

Q.E.D.

Proof of Theorem 3.5. Let $G = (\Sigma, V, P, S)$ be a context-free grammar. We now apply the constructions from Lemmas 3.6, 3.7, & 3.9: in the first step, we make sure that all rules are either unary or have only variables on the right-hand side; in the second step, we form the unit closure; then we remove unit productions; finally, we iteratively replace all rules of the form $A \to \alpha = A_0 ... A_n$ for $n \geq 3$ by $P_{A \to \alpha}$. Note that all of these steps only require making finitely many changes to the grammars. The resulting grammar is in Chomsky normal form; Lemmas 3.6, 3.7, 3.8 & 3.9 show that the resulting grammar is equivalent to the original grammar. Q.E.D.

3.3 The pumping lemma for context-free languages

Definition 3.10. Let $L \subseteq \mathbb{W}$ be a language. We say that L *satisfies the (context-free) pumping lemma with pumping number* n if for every word $w \in L$ such that $|w| \geq n$ there are words u, v, x, y, z such that $w = xuyvz$, $|uv| > 0$, $|uyv| \leq n$ and for all $k \in \mathbb{N}$, we have that $xu^k yv^k z \in L$. We say that L *satisfies the (context-free) pumping lemma* if there is some n such that it satisfies the (context-free) pumping lemma with pumping number n.

The first proof of the context-free pumping lemma is usually attributed to Yehoshua Bar-Hillel (1915–1975); the statement is therefore also known as the *Bar-Hillel Lemma*.[12]

Proposition 3.11. Every language that satisfies the (regular) pumping lemma satisfies the (context-free) pumping lemma.

Proof. If $w = xuz$ with $|u| > 0$ and $|xu| \leq n$, then let $y := \varepsilon$ and $v := \varepsilon$. Clearly, $|uv| \geq |u| > 0$ and $|uyv| = |u\varepsilon\varepsilon| = |u| \leq |xu| \leq n$ and $xu^k yv^k z = xu^k \varepsilon\varepsilon^k z = xu^k z$. Q.E.D.

Therefore, the proof of Corollary 2.16 implies that there are uncountably many languages satisfying the context-free pumping lemma. As in the case of the regular pumping lemma, this means that the pumping lemma cannot characterise any of our classes of languages.

Theorem 3.12 (The context-free pumping lemma)**.** For every context-free language L, there is an n such that L satisfies the context-free pumping lemma with pumping number n.

Proof. By Theorem 3.5, there is a grammar $G = (\Sigma, V, P, S)$ in Chomsky normal form such that $L = \mathcal{L}(G)$. Let $m := |V|$ and $n := 2^m + 1$. We claim that n is a pumping number of L. Let $w \in \mathcal{L}(G)$ such that $|w| \geq n$ and let $\mathbf{T}$ be a G-parse tree starting with S such that

[12] Cf. Y. Bar-Hillel, M. Perles, & E. Shamir (1961). On formal properties of simple phrase-structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung* 14:2, 143–172.

$\sigma_{\mathbf{T}} = w$. By Lemma 3.4, we know that the height of $\mathbf{T}$ must be at least $m+1$. Find some terminal node $t \in T$ such that $|t| \geq m+1$ and some $s \subseteq t$ on the branch leading to t such that the subtree T_s has height precisely $m+1$. In the subtree T_s, the branch from ε to t has length $m+2$ and its labels are $m+1$ many variables and one letter (labelling the terminal node t itself). Since $|V| = m$, by the pigeonhole principle, there are two nodes on the branch with the same label, say, $t_0 \subsetneq t_1$ such that $\ell(t_0) = \ell(t_1) = A \in V$. In particular, $\mathbf{T}_{t_0}$ and $\mathbf{T}_{t_1}$ are both parse trees starting with A. We write

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

Observe that $|uv| > 0$ since $t_0 \neq t_1$ and that $|uyv| = |\sigma_{\mathbf{T}_{t_0}}| \leq |\sigma_{\mathbf{T}_s}| \leq 2^m \lneq n$ by Lemma 3.4. Let $x := x_0 x_1$ and $z := z_1 z_0$. Then $w = xuyvz$ satisfies the length bounds of the context-free pumping lemma. All that's left to show is that for all $k \in \mathbb{N}$, $xu^k y v^k z \in L$. We define recursively

$$\begin{aligned}
\mathbf{T}_{(0)} &:= \mathbf{T}_{t_1}, \\
\mathbf{T}_{(i+1)} &:= \operatorname{graft}(\mathbf{T}_{t_0}, t_1, \mathbf{T}_{(i)}), \text{ and} \\
\mathbf{T}_k &:= \operatorname{graft}(\mathbf{T}, t_0, \mathbf{T}_{(k)}).
\end{aligned}$$

Then $\mathbf{T}_{(k)}$ is a G-parse tree starting with A and $\sigma_{\mathbf{T}_{(k)}} = u^k y v^k$ [by induction]. Therefore $\mathbf{T}_k$ is a G-parse tree starting with S and $\sigma_{\mathbf{T}_k} = xu^k y v^k z \in L$. Q.E.D.

Example 3.13. The language $L := \{a^k b^k c^k \,;\, k \geq 1\}$ is not context free.

[Suppose it were, then by the pumping lemma, there is a pumping number n. Consider the word $w = a^n b^n c^n \in L$ with $|w| = 3n \geq n$. Thus, we can write $w = xuyvz$ with $|uv| > 0$ and $|uyv| \leq n$. This means that the subword uyv cannot contain all three letters a, b, and c, so it is of the form $a^k b^\ell c^m$ where either $k = 0$ or $m = 0$; the condition $|uv| > 0$ means that $k + \ell + m > 0$. So, if we pump down, we have two cases to consider:

Case 1. We have $k = 0$. Then the word still contains n many as, but at least one of the numbers of bs or cs has been reduced. Thus the pumped word is not in L anymore.

Case 2. We have $m = 0$. Then the word still contains n many cs, but at least one of the numbers of as or bs has been reduced. Thus the pumped word is not in L anymore.

Together, this yields a contradiction.]

3.4 Closure properties

Proposition 3.14. The class of context-free languages is not closed under intersection.

Proof. This follows readily from Example 3.13. Consider

$$L_0 := \{a^m b^m c^k \,;\, m, k \geq 1\} \text{ and}$$
$$L_1 := \{a^k b^m c^m \,;\, m, k \geq 1\};$$

clearly, $L_0 \cap L_1 = \{a^k b^k c^k \,;\, k \geq 1\}$ which is not context-free. So, we only need to argue that both L_0 and L_1 are context-free. Let $G_0 = (\{a, b, c\}, \{S, X, C\}, P_0, S)$ and $G_1 = (\{a, b, c\}, \{S, A, Y\}, P_1, S)$ with $P_0 := \{S \to XC, X \to aXb, X \to ab, C \to cC, C \to c\}$ and $P_1 := \{S \to AY, A \to aA, A \to a, Y \to bYc, Y \to bc\}$. Clearly, $\mathcal{L}(G_0) = L_0$ and $\mathcal{L}(G_1) = L_1$.[13] Q.E.D.

Therefore by Proposition 1.20, the class of context-free languages cannot be closed under complements and differences. In light of the product automaton construction from § 2.5, this tells us that any model of computation that characterises the context-free languages cannot have a product construction.

In this lecture course, we shall not see the corresponding model of computation: it is the notion of *pushdown automaton*. A pushdown automaton is like a regular automaton, but it as a storage device known as a *stack*. A stack is a storage unit in which you can store, remove, and read letters by the *last-in-first-out* (LIFO) principle. The transition function δ of the automaton not only determines the state of the automaton, but also the actions to be performed with respect to the stack, and it depends on what the automaton can see on the stack. It can be proved that a language is context-free if and only if it is accepted by such a pushdown automaton. The failure of closure by intersection informs us that there cannot be a product construction for pushdown automata.

On Example Sheet # 2, we shall see that the class of context-free grammars is (like the class of regular grammars) closed under the Kleene plus operation.

3.5 Decision problems

Again, we shall consider the word problem, the emptiness problem, and the equivalence problem for our classes of languages. In Theorem 1.19, we already solved the word problem for context-free languages positively. The proof of Theorem 1.19 was not very efficient: it potentially requires to check a vast (yet finite) amount of possible derivations. Remember that in § 2.8, the solution to the word problem was much more straightforward: the automaton provided an algorithm that would determine in $|w|$ steps whether the automaton accepted w. A similar situation can be found in context-free grammars that are in Chomsky normal form: by Lemma 3.3, we know that the derivation of a word w will have length $2|w| - 1$.

The emptiness problem was essentially solved in § 2.8: we proved that any language satisfying the regular pumping lemma with pumping number n that is non-empty must contain a word of length less than n. Re-checking the proof, we realise that this had nothing to do with the regular pumping lemma: also the context-free pumping lemma allows us to pump down every word of length the pumping number or longer, so a word of minimal length must be shorter than n.

[13] Alternatively, observe that $\{a^n \,;\, n > 0\}$ and $\{c^n \,;\, n > 0\}$ are regular, hence context-free, and that $\{b^n c^n \,;\, n > 0\}$ and $\{a^n b^n \,;\, n > 0\}$ are context-free by Example 3.1. The closure of the context-free languages under concatenation does the job.

	regular (type 3)	context-free (type 2)
Closure properties.		
Concatenation	✓	✓
Union	✓	✓
Intersection	✓	×
Complementation	✓	×
Difference	✓	×
Decision problems.		
Word problem	✓	✓
Emptiness problem	✓	✓
Equivalence problem	✓	×

Figure 4: Closure properties and decision problems of regular and context-free grammars in an overview.

Corollary 3.15. The emptiness problem for context-free grammars is solvable.

Proof. Given a context-free grammar G, first transform it into Chomsky normal form by the operations in Lemmas 3.6, 3.7, & 3.9. Note that this is an algorithmic procedure. Now count the number m of variables and calculate $n := 2^m$. By the above argument, $\mathcal{L}(G)$ is non-empty if and only if there is a word of length $< n$ in $\mathcal{L}(G)$. Therefore, we can now systematically check for all words of length $< n$ whether they are in $\mathcal{L}(G)$ (either use Theorem 1.19 or, more efficiently, Lemma 3.3). If at least one of them is, $\mathcal{L}(G) \neq \varnothing$; otherwise $\mathcal{L}(G) = \varnothing$. Q.E.D.

In contrast, the *Equivalence problem* for context-free grammars is undecidable. We will not prove this in this course, but a proof can be found in Sipser's textbook,[14] using the unsolvability of the halting problem (Theorem 4.33) and the technique of reduction functions from § 4.11: Sipser's Exercise 5.1 (p. 211) reduces the equivalence problem for context-free grammars to the *universality problem* for context-free grammars $\{G\,;\, \mathcal{L}(G) = \mathbb{W}\}$ and Sipser's Theorem 5.13 (p. 197) reduces that problem to the non-computable set $\mathbf{K}_0$ (cf. § 4.8). We summarise what we know so far (including the unproved claim about unsolvability) in Figure 4.

[14] M. Sipser. Introduction to the theory of computing. Second edition. Thomson Course Technology, 2006

4 Computability theory

4.1 Register machines

Let Σ be an alphabet and Q a non-empty finite set whose elements we shall call *states*. A tuple of the form

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

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

$$\begin{aligned}+(k,a,q)&:=(0,k,a,q), && (\text{“add”})\\?(k,a,q,q')&:=(1,k,a,q,q'), && (\text{“check”})\\?(k,\varepsilon,q,q')&:=(2,k,q,q') \text{ and} && (\text{“check”})\\-(k,q,q')&:=(3,k,q,q') && (\text{“remove”})\end{aligned}$$

and interpret these instructions as listed in Table 4.1. There are infinitely many instructions of each type, but if we bound the natural number k occurring in them, we only have a finite number of instructions: there are $n\cdot(|\Sigma|\cdot|Q|+|\Sigma|\cdot|Q|^2+2\cdot|Q|^2)$ many instructions with $k<n$.

Definition 4.1. A tuple $M:=(\Sigma,Q,P)$ is called a Σ-*register machine* (or just *register machine*, if Σ is clear from the context) if Q is a non-empty finite set with two special elements $q_{\mathrm{S}}\neq q_{\mathrm{H}}$, the *start state* and the *halt state*, and P is a function with domain Q such that each $P(q)$ is a (Σ,Q)-instruction. The function P is called the *program* of the register machine. For a fixed $q\in Q$, we also refer to $q\mapsto P(q)$ as a *program line*.

We observe that because Q is finite, the range of P contains only finitely many instructions, so for any given register machine M there is a maximal number k that shows up in any of the instructions in the range of P. This number is called the *upper register index* of

Instruction	Interpretation
$+(k,a,q)$	“Add the letter a to the content of register k and go to state q.”
$?(k,a,q,q')$	“Check whether the last letter in register k is a; if so, go to state q; otherwise, go to state q'.”
$?(k,\varepsilon,q,q')$	“Check whether register k is empty; if so, go to state q; otherwise, go to state q'.”
$-(k,q,q')$	“Check whether register k is empty; if so, go to state q; otherwise, remove the final letter of its content and go to state q'”.

Table 1: Interpretations of register machine instructions.

M. If n is the upper register index of a register machine M, we can think of M as a device that has $n+1$ many storage units, called *registers*, that can contain words in $\mathbb{W}$ and is in a state $q \in Q$ that determines what it going to do next via the program P. So, at any given time, the situation of the register machine is determined by its state and what is in the $n+1$ many registers.

We say that a sequence $C := (q, w_0, ..., w_n) \in Q \times \mathbb{W}^{n+1}$ is a *configuration* or *snapshot* of length $n+1$. In such a configuration, the first entry q is called the *state* of the configuration and the rest is called the *register content* of the configuration. If M is a register machine with upper register index n and C is any configuration of length $m \geq n+1$, then we can define the action of M on C: we say that M *transforms* C *to* C' if the following is true:

Case 1. If $P(q) = +(k, a, q')$ and $C' = (q', w_0, ..., w_{k-1}, w_k a, w_{k+1}, ..., w_m)$.

Case 2. If $P(q) = ?(k, a, q', q'')$,

Subcase 2a. $w_k = wa$ for some w and $C' = (q', w_0, ..., w_m)$ or

Subcase 2b. $w_k \neq wa$ for any w and $C' = (q'', w_0, ..., w_m)$.

Case 3. If $P(q) = ?(k, \varepsilon, q', q'')$,

Subcase 3a. $w_k = \varepsilon$ and $C' = (q', w_0, ..., w_m)$ or

Subcase 3b. $w_k \neq \varepsilon$ and $C' = (q'', w_0, ..., w_m)$.

Case 4. If $P(q) = -(k, q', q'')$,

Subcase 4a. $w_k = \varepsilon$ and $C' = (q', w_0, ..., w_m)$ or

Subcase 4b. $w_k = wa$ for some a and $C' = (q'', w_0, ..., w_{k-1}, w, w_{k+1}, ..., w_m)$.

We think of a register machine M as a *model of computation* in the following sense: the start state q_S is the state the machine is in at the beginning of the computation. We give the machine some input in its registers, i.e., a sequence $\vec{w} = (w_0, ..., w_n) \in \mathbb{W}^{n+1}$ where n is the upper register index of M. Then we can define the sequence of computational snapshots by recursion:

Definition 4.2. If $M = (\Sigma, Q, P)$ is a register machine with upper register index n and $\vec{w} := (w_0, ..., w_n) \in \mathbb{W}^{n+1}$, then the *computation sequence of* M *with input* $\vec{w}$ is defined by recursion as follows:

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

In order to apply the recursion, we need an input sequence $\vec{w}$ that has at least length $n+1$ where n is the upper register index of M. We shall use the following notational convention: if $\vec{v} = (v_0, ..., v_k)$ is a shorter sequence, we interpret it as $\vec{w} = (v_0, ..., v_k, w_{k+1}, ..., w_n)$ where $w_i = \varepsilon$. In particular, if $k = 0$, we talk about "input w" for a single word w (which is then interpreted as a sequence of length $n+1$ with all other registers being empty).

Note that the function $k \mapsto C(k, M, \vec{w})$ is always defined, so any computation sequence represents an infinitely long computation. Of course, we are not interested in infinitely long

computations, but rather in those computations that will eventually halt. This is where our second special state, the halt state q_{H} comes into play. We say that a computation sequence *halts* if there is some element $(q, \vec{w})$ in the sequence such that $q = q_{\mathrm{H}}$. Otherwise, we say that the computation sequence *does not halt*.

If M is a register machine and $\vec{w}$ a sequence of words, we say that M *halts on input* $\vec{w}$ *in* k *steps* if the computation sequence of M with input $\vec{w}$ halts and k is the least number such that $C(k, M, \vec{w}) = (q_{\mathrm{H}}, \vec{v})$ for some $\vec{v}$; this sequence $\vec{v}$ is called the *register content at the time of halting*; we also use the terminology M *converges on input* $\vec{w}$ for this. If the computation sequence does not halt, we also say that M *diverges on input* $\vec{w}$.

We can call two Σ-register machines $M = (\Sigma, Q, P)$ and $M' = (\Sigma, Q', P')$ *strongly equivalent* if the register content of each of the elements of their computation sequences is the same, i.e., for all k and all $\vec{w}$, if $C(k, M, \vec{w}) = (q, \vec{v})$ and $C(k, M', \vec{w}) = (q', \vec{u})$, then $\vec{v} = \vec{u}$, and furthermore the state of a configuration in the computation sequence of one of the machines is the halting state if and only if the state in the corresponding configuration in the other computation sequence is the halting state, i.e., $C(k, M, \vec{w}) = (q_{\mathrm{H}}, \vec{v})$ if and only if $C(k, M', \vec{w}) = (q'_{\mathrm{H}}, \vec{v})$. As with grammars (cf. the proof of Proposition 1.14), we observe that if $|Q| = |Q'|$, then for each register machine $M = (\Sigma, Q, P)$, there is a register machine $M' = (\Sigma, Q', P')$ that is strongly equivalent, so the precise nature of Q is irrelevant, only its size matters.

Proposition 4.3. For any fixed Σ, there are only countably many register machines up to strong equivalence.

Proof. Fix k and n and observe that for any $|Q| = n$, there are only finitely many register machines with upper register index $\leq k$. [This follows from our previous finite upper bound on the number of (Σ, Q)-instructions.] By the previous remark, only the size of the set Q matters up to strong equivalence, so for fixed k and n, the set of register machines with any state set of size n and upper register index $\leq k$ up to strong equivalence is finite. But then the set of all Σ-register machines up to strong equivalence is a countable union of finite sets, thus countable. Q.E.D.

Proposition 4.4 (Padding Lemma)**.** For each register machine there are infinitely many strongly equivalent register machines.

Proof. Let $M = (\Sigma, Q, P)$ be any register machine and let $\widehat{q} \notin Q$. Because $\widehat{q}$ is not in Q, it does not show up in any instructions in the range of P. Define $M^+ := (\Sigma, Q \cup \{\widehat{q}\}, P^+)$ where $P^+ \restriction Q = P$ and $P^+(\widehat{q}) := ?(0, \varepsilon, \widehat{q}, \widehat{q})$. Clearly, the state set of M^+ has one element more than the state set of M. By construction, if C is a configuration with state in Q, then M^+ and M will transform C in precisely the same way. Since $q_{\mathrm{S}} \in Q$, we can show by induction that the computation sequences of M^+ are precisely the computation sequences of M (actually, the entire sequences, not just the register content of the configurations). The construction $M \mapsto M^+$ produces a strongly equivalent machine with strictly bigger state set. We can now produce infinitely many pairwise distinct machines by recursively adding additional new elements that are irrelevant for the computation. Q.E.D.

4.2 Performing operations and answering questions

In the following, we shall talk about partial functions, i.e., functions that are not necessarily defined everywhere. In this lecture, we shall use the notation $f : X \dashrightarrow Y$ for "f is a partial function from X to Y", i.e., $\mathrm{dom}(f) \subseteq X$ and $\mathrm{ran}(f) \subseteq Y$. In addition, for partial functions, we introduce the following useful notation: if $f : X \dashrightarrow Y$, we write

$$\begin{array}{ll} f(x)\downarrow & \text{if and only if } x \in \mathrm{dom}(f) \text{ and} \\ f(x)\uparrow & \text{otherwise} \end{array}$$

and use our terminology for computations by saying "f *converges* on input x" for $f(x)\downarrow$ and "f diverges on input x" for $f(x)\uparrow$. If $f : X \dashrightarrow Y$ and $g : Y \dashrightarrow Z$, then the *concatenation of f and g*, denoted by $g \circ f : X \dashrightarrow Z$ is defined by $g \circ f(x) = g(f(x))$; in particular, if $x \notin \mathrm{dom}(f)$, then $x \notin \mathrm{dom}(g \circ f)$.

Fix an upper register index m and $n \leq m$. If $\vec{w} = (w_0, ..., w_n) \in \mathbb{W}^{n+1}$, we write $\vec{w}^+$ for the $m+1$-tuple $(w_0, ..., w_n, \varepsilon, ..., \varepsilon) \in \mathbb{W}^{m+1}$, i.e., the tuple $\vec{w}$ with all remaining entries filled up with the empty word. We think of the registers with the indices $n+1$ to m as *scratch space* that the machine can use to keep information while performing its computation.

Let $F : \mathbb{W}^{n+1} \dashrightarrow \mathbb{W}^{n+1}$ be any partial function. We say that a register machine M with upper register index m *performs the operation F* if for all $\vec{w} \in \mathbb{W}^{n+1}$

(i) if $F(\vec{w})\uparrow$, then M diverges on input $\vec{w}^+$ and

(ii) if $F(\vec{w})\downarrow = \vec{v}$, then M converges on input $\vec{w}^+$ with register content $\vec{v}^+$ at time of halting.

If F is a total function, we sometimes emphasise this by using the phrase "M performs the total operation F".

A *question about $n+1$-tuples with $k+1$ answers* is a partition of $\mathbb{W}^{n+1}$ into $k+1$ disjoint sets $A_0, ..., A_k$. E.g., the question "does the second register end with a?" is the partition $A_0 := \{\vec{w}\,;\, \exists v(w_2 = va)\}$ and $A_1 := \mathbb{W}^{n+1} \backslash A_0$. A register machine M with upper register index m *answers a question about $n+1$-tuples with $k+1$ answers* if it has $k+1$ designated *answer states* $\widehat{q_0}, ..., \widehat{q_k}$, and for any input $\vec{w}^+ \in \mathbb{W}^{m+1}$, the computation of M with input $\vec{w}$ produces in finitely many steps a configuration $(\widehat{q_i}, \vec{w}^+)$ if and only if $\vec{w} \in A_i$.

Example 4.5. (1) The operation "*never halt*" corresponds to the partial function $f : \mathbb{W}^{n+1} \dashrightarrow \mathbb{W}^{n+1}$ with $\mathrm{dom}(f) = \varnothing$ and is performed by the register machine with programme $q_{\mathrm{S}} \mapsto +(0, a, q_{\mathrm{S}})$. Note that many register machines perform this operation: e.g., any register machine that does not have q_{H} in any of its instructions.

(2) The operation "*halt without changing anything*" corresponds to the total identity function $f(\vec{w}) = \vec{w}$ and is performed by the register machine with programme $q_{\mathrm{S}} \mapsto ?(0, \varepsilon, q_{\mathrm{H}}, q_{\mathrm{H}})$.

(3) The question "*Is register i empty?*" corresponds to the partition given by $A_0 := \{\vec{w}\,;\, w_i = \varepsilon\}$ and $A_1 := \{\vec{w}\,;\, w_i \neq \varepsilon\}$ and is answered by the register machine with programme $q_{\mathrm{S}} \mapsto ?(i, \varepsilon, \widehat{q_0}, \widehat{q_1})$.

(4) The question "*Does register i end with letter a?*" corresponds to the partition given by $A_0 := \{\vec{w}\,;\, \exists v(w_i = va)\}$ and $A_1 := \mathbb{W}^{n+1} \backslash A_0$ and is answered by the register machine with programme $q_{\mathrm{S}} \mapsto ?(i, a, \widehat{q_0}, \widehat{q_1})$.

Lemma 4.6 (Concatenation Lemma or Subroutine Lemma)**.** Let $M = (\Sigma, Q, P)$ and $M' = (\Sigma, Q', P')$ be two register machines. If M performs operation F and M' performs operation F', then we can construct a register machine that performs operation $F' \circ F$.

Proof. We can assume w.l.o.g. that $Q \cap Q' = \varnothing$. If we only care about whether a machine performs an operation, the value of $P(q_{\mathrm{H}})$ never matters: for each input, the state q_{H} is either never reached, or if it is reached, the tail of the computation sequence after the computation reaches q_{H} is irrelevant for the output of the computation. So, we can alter that instruction without affecting the fact that M performs F. Let $\widehat{Q} := Q \cup Q' \backslash \{q_{\mathrm{H}}\}$. Define a set P^* consisting of P without $(q_{\mathrm{H}}, P(q_{\mathrm{H}}))$ and all instances of q_{H} in the instructions replaced by q'_{S}. Then $\widehat{P} := P^* \cup P'$ is a (Σ, Q^*)-program and $\widehat{M} := (\Sigma, \widehat{Q}, \widehat{P})$ is a register machine that performs the operation $F' \circ F$. Q.E.D.

Lemma 4.7 (Case Distinction Lemma)**.** Let $Q = \{A_i \,;\, i \leq k\}$ be a question with $k+1$ answers and $f_i : \mathbb{W}^{n+1} \dashrightarrow \mathbb{W}^{n+1}$ be operations for $i \leq k$. If Q is answered by a register machine $M = (\Sigma, Q, P)$ and f_i is performed by $M_i := (\Sigma, Q_i, P_i)$ (for $i \leq k$), then we can construct a register machine that performs the operation defined by $g(\vec{w}) := f_i(\vec{w})$ if and only of $\vec{w} \in A_i$.

Proof. As in the proof of Lemma 4.6, we observe that the instructions $P(q_{\mathrm{H}})$ and $P'(q'_{\mathrm{H}})$ are irrelevant, so we can w.l.o.g. assume that the machines M_i all share the same halt state q_{H} and that their programs agree on that state, i.e., $\bigcap_{i \leq k} Q_i = \{q_{\mathrm{H}}\}$ and $P_i(q_{\mathrm{H}}) = P_j(q_{\mathrm{H}})$ for all $i, j \leq k$; furthermore, we can assume w.l.o.g. that for all $i \leq k$, we have that $Q \cap Q_i = \varnothing$. Let $q_{\mathrm{S},i}$ be the start state of M_i and $\widehat{q_i}$ be the answer states of the machine M. Let P_i^* be the program consisting of the program lines of P_i with all occurrences of $q_{\mathrm{S},i}$ replaced by $\widehat{q_i}$. Let $\widehat{Q} := Q \cup \bigcup_{i \leq k} (Q_i \backslash \{q_{\mathrm{S},i}\})$, $\widehat{P} := P \cup \bigcup_{i \leq k} P_i^*$, and $\widehat{M} := (\Sigma, \widehat{Q}, \widehat{P})$. Then $\widehat{M}$ performs the operation g. Q.E.D.

Let $Q = \{A_0, A_1\}$ be a question about n-tuples with two answers and $F : \mathbb{W}^n \dashrightarrow \mathbb{W}^n$ an operation. Define by recursion $F^0(\vec{w}) := \vec{w}$ and $F^{m+1}(\vec{w}) := F(F^m(\vec{w}))$ and

$$R_{F,Q}(\vec{w}) := \begin{cases} F^m(\vec{w}) & \text{if } m \text{ is the least number such that } F^m(\vec{w}) \in A_1 \text{ and} \\ \uparrow & \text{if there is no such number.} \end{cases}$$

The operation $R_{F,Q}$ can be described as "repeat F until the answer to Q is A_1".

Lemma 4.8 (Repeat Lemma)**.** If $Q = \{A_0, A_1\}$ be a question about n-tuples with two answers that can be answered by a register machine and $F : \mathbb{W}^n \dashrightarrow \mathbb{W}^n$ an operation performed by a register machine. Then $R_{F,Q}$ is performed by a register machine.

Proof. Let $M = (\Sigma, Q, P)$ be a register machine performing F and $M' = (\Sigma, Q', P')$ be a register machine answering Q. As before, we can assume that $Q \cap Q' = \varnothing$; let q_{S}, q'_{S}, q_{H}, and q'_{H} be the start and halt states of M and M', respectively. We define $\widehat{M} := (\Sigma, \widehat{Q}, \widehat{P})$ where $\widehat{Q} := Q \cup Q'$ and $\widehat{P}$ is $P \cup P'$ where all occurrences of $\widehat{q_0}$ (the answer state for A_0) in the instructions are replaced by q_{S} and all occurrences of q_{H} are replaced by q'_{S}. The start state

of $\widehat{M}$ is q'_{S} and the halt state is $\widehat{q_1}$ (the answer state for A_1). Then $\widehat{M}$ performs the operation $R_{F,Q}$. Q.E.D.

We emphasise that the operations producing the machines $\widehat{M}$ in the proofs of Lemmas 4.6, 4.7, & 4.8 are concrete constructions: given the required register machines, the proofs provide a concrete definition of the desired machine $\widehat{M}$. As a consequence, any description of the construction of a register machine using subroutines, case distinctions, and repeat loops is not just an informal description, but rather a concrete instruction how to explicitly write down the resulting register machine (if one cares to do the hard work and provide all of the detail). Thus, in the following, we shall not explicitly give the program lines for the machine, but rather build descriptions of register machines performing operations or answering questions using previously described register machines and linking them by subroutines, case distinctions, or repeat loops. E.g.,

$$f(\vec{w}) = \begin{cases} \vec{w} & \text{if } w_i \neq \varepsilon \text{ and} \\ \uparrow & \text{if } w_i = \varepsilon \end{cases} \qquad (\ddagger)$$

can be performed by a register machine as follows: check if the ith register is empty; if so, halt without any change; if not, never halt.

Example 4.9. The following operations and questions are performed or answered by register machines:

(1) "*Delete the final letter in register i, if it exists.*"

[The program $q_{\mathrm{S}} \mapsto -(i, q_{\mathrm{H}}, q_{\mathrm{H}})$ performs this operation.]

(2) "*Delete the content of register i.*"

[The program $q_{\mathrm{S}} \mapsto -(i, q_{\mathrm{H}}, q_{\mathrm{S}})$ performs this operation.]

(3) "*Add a to the end of register i.*"

[The program $q_{\mathrm{S}} \mapsto +(i, a, q_{\mathrm{H}})$ performs this operation. Note that this also performs the operation "*guarantee that register i is not empty*".]

(4) "*Add w to the end of register i.*"

[If $w = a_0...a_m$, then concatenate the operations "Add a_i to the end of register i" by Lemma 4.6 and (3).]

(5) "*Replace the content of register i with w.*"

[First empty register i by (2), then add w to register i by (4).]

(6) "*What is the final letter of register i?*"

[If $\Sigma = \{a_0, ..., a_k\}$, then this is a question with $k + 2$ answers, i.e.,

$$A_\ell := \{\vec{w}\,;\, \exists v(w_i = va_\ell)\}$$

(for $\ell \leq k$) and $A_{k+1} := \{\vec{w}\,;\, w_i = \varepsilon\}$. We can answer this question by checking each letter in turn with Example 4.5 (4): "Does register i end in letter a_ℓ?" If yes, we go to state $\widehat{q_\ell}$, if not and $\ell \neq k$, we answer the next question; if not, and $\ell = k$, we go to answer state $\widehat{q_{k+1}}$.]

(7) "*Copy the final letter of register i (if it exists) to register j.*"

[Determine the final letter of register i by (6). If the answer is $\widehat{q_{k+1}}$, perform "halt" via Example 4.5 (2); if it is $\widehat{q_\ell}$ for some $\ell \leq k$, perform "add a_ℓ to the end of register j" (3).]

(8) "*Move the final letter of register i (if it exists) to register j.*"

[Check whether register i is empty via Example 4.5 (3). If so, perform "halt". If not, perform "copy the final letter of register i to register j" (7) and then "delete the final letter in register i" (1).]

(9) "*Move the content of register i into register j in reverse order.*"

[Using Lemma 4.8, perform the operation "move the final letter of register i (if it exists) to register j" (8) repeatedly until register i is empty and halt.]

(10) "*Move the content of register i into register j.*"

[Take a register k from the scratch space. Then move the content of register i to register k in reverse order (9) and after that move the content of register k to register j in reverse order.]

(11) "*Copy the content of register i into register j in reverse order.*"

[Take a register k from the scratch space. Using Lemma 4.8, perform the operations "Copy the final final letter of register i (if it exists) to register k" (7) and "move the final letter of register i (if it exists) to register j" (8) repeatedly until register i is empty. After that, move the content of register k to register i in reverse order (9).]

(12) "*Copy the content of register i to register j.*"

[Take a register k from the scratch space. Copy the content of register i to register k in reverse order (11) and then move the content of register k to register j in reverse order (9).]

(13) "*Is the content of register i exactly w?*"

[If $w = a_0...a_k$, answer the questions "Is a_ℓ the final letter of register i?" from the back of the word. If one of the questions gets a negative answer, answer "no". If the answer is positive, move the final letter to a scratch register and continue. If all $k+1$ checks are positive, move the word back from the scratch register and answer "yes".]

Note that some of these operations and questions require the use of *scratch space* to store information that would otherwise be lost: it is not always possible to perform an operation on $\mathbb{W}^{n+1}$ with only $n+1$ many registers. E.g., copying the content of register i to register j in Example 4.9 (12) requires the storage of the word in a scratch register k.

Also observe that there is some informality in our descriptions since we do not precisely determine which registers are our scratch registers. This could be formalised if necessary (e.g., by always using the next register that hasn't been used in the program before), but this choice is immaterial for the performance of the machine. Note that two machines that use different scratch registers but are otherwise the same are not *strongly equivalent*: this suggests that for many purposes, the notion of strong equivalence of register machines is too strong.

4.3 Computable functions & sets

If M is a Σ-register machine and $k \in \mathbb{N}$, we can define a partial function $f_{M,k} : \mathbb{W}^k \dashrightarrow \mathbb{W}$ by

$$\begin{aligned} f_{M,k}(\vec{w})\!\uparrow &\text{ if and only if } M \text{ does not halt on input } \vec{w}, \\ f_{M,k}(\vec{w}) = v_0 &\text{ if and only if } M \text{ halts on input } \vec{w} \text{ with register content} \\ &\qquad \vec{v} \text{ at the time of halting.} \end{aligned}$$

Note the subtle difference to the setting in § 4.2: there we considered the entire $\vec{v}$ as the output of the computation; here, we consider only what happens in register 0 as output. Everything else is considered as part of the input and scratch space. If two Σ-register machines M and M' are strongly equivalent, then the partial functions defined by them are equal, i.e., $f_{M,k} = f_{M',k}$. However, it can be easily seen that the converse does not hold, i.e., there can be machines that produce the same function, but are not strongly equivalent: they do not produce the same computation sequences, but they still produce the same halting behaviour and the same output (which only lives in register 0). The domain of the partial function $f_{M,k}$ is exactly the set of k-tuples of words $\vec{w}$ for which the machine M halts if given the input $\vec{w}$. If $k = 1$, we write $\mathrm{W}_M := \mathrm{dom}(f_{M,1})$. This gives rise to an even weaker notion of equivalence: machine M and M' are called *weakly equivalent* if $\mathrm{W}_M = \mathrm{W}_{M'}$. Again, if the defined functions $f_{M,k}$ and $f_{M',k}$ are the same, then M and M' are weakly equivalent, but the converse need not hold.

Definition 4.10. A partial function $f : \mathbb{W}^k \dashrightarrow \mathbb{W}$ is called *computable* if there is a Σ-register machine M such that $f = f_{M,k}$.

The Padding Lemma (Proposition 4.4) immediately implies that the machine computing f is not unique; in fact, for every computable partial function f there are infinitely many different machines that compute f. Proposition 4.3 yields that there are at most countably many computable partial functions.

Example 4.11. Based on the constructions in § 4.2, we already know many examples of computable partial functions:

(1) The identity function $\mathrm{id} : w \mapsto w$ (Example 4.5 (2));

(2) constant functions $\mathrm{c}_{k,v} : \mathbb{W}^k \to \mathbb{W} : \vec{w} \mapsto v$ (Example 4.9 (5));

(3) projection functions $\pi_{k,i} : \mathbb{W}^k \to \mathbb{W} : \vec{w} \mapsto w_i$ for some $i < k$ (if $k = 0$, the identity does the job; otherwise, empty register 0 and copy the content of register i to register 0).

If $X \subseteq \mathbb{W}^k$, we call any total function $f : \mathbb{W}^k \to \mathbb{W}$ with the property

$$f(\vec{w}) \neq \varepsilon \iff \vec{w} \in X$$

a *characteristic function of* X. We can fix a particular $a \in \Sigma$ and then call the function

$$\chi_X(\vec{w}) := \begin{cases} a & \text{if } \vec{w} \in X \text{ and} \\ \varepsilon & \text{if } \vec{w} \notin X. \end{cases}$$

the characteristic function of X. Similarly, we call any partial function $f : \mathbb{W}^k \dashrightarrow \mathbb{W}$ with $\mathrm{dom}(f) = X$ a *pseudo-characteristic function of* X and

$$\psi_X(\vec{w}) := \begin{cases} a & \text{if } \vec{w} \in X \text{ and} \\ \uparrow & \text{if } \vec{w} \notin X. \end{cases}$$

the pseudo-characteristic function of X. A set $X \subseteq \mathbb{W}^k$ is called *computable* if its characteristic function is computable; it is called *computably enumerable* if its pseudo-characteristic function is computable.

Proposition 4.12. Let $X \subseteq \mathbb{W}^k$.

(a) If X is computable, then so is $\mathbb{W}^k \backslash X$, i.e., being computable is closed under complementation.

(b) Then ψ_X is computably enumerable if and only if there is a computable pseudo-characteristic function. In particular, a set is computably enumerable if and only if it is the domain of a computable partial function. For $X \subseteq \mathbb{W}$, the set X is computably enumerable if and only if there is an M such that $X = \mathrm{W}_M$.

(c) If X is computable, then it is computably enumerable.

We'll see later that the converse of (c) does not hold and that (a) does not hold for computably enumerable sets.

Proof. (a) Consider $g : \mathbb{W} \to \mathbb{W}$ defined by $g(\varepsilon) = a$ and $g(w) = \varepsilon$ for any $w \neq \varepsilon$. This is computable: first check whether w is empty; if so, empty register 0, add a to that register, and halt. If not, empty register 0 and halt.

Clearly, $\chi_{\mathbb{W}^k \backslash X} = g \circ \chi_X$, so the claim follows from Lemma 4.6.

(b) Only the backwards implication is non-trivial. Let c be the constant function that maps everything to the designated element a and ψ be any computable pseudo-characteristic function. Then $c \circ \psi = \psi_X$.

(c) In (‡) on page 49, we had seen that the partial function f that is the identity on non-empty words and diverges on the empty word is computable. Then $\psi_L = f \circ \chi_L$. Q.E.D.

Theorem 4.13. Every regular language is computable.

Proof. Let L be regular and let $D = (\Sigma, Q, \delta, q_0, F)$ be a deterministic automaton such that $\mathcal{L}(D) = L$. We describe a register machine $\widehat{M} = (\Sigma, \widehat{Q}, P)$ that takes w as input in register 0, mimics the computation of the automaton D, and outputs a if $w \in L$ and ε if $w \notin L$. For each state of the automaton $q \in Q$, our register machine will have a subset $Q_q \subseteq \widehat{Q}$ of states: this subset of register machine states will only be left if we explicitly say so, and while the register machine is in states from Q_q, it is mimicking steps of automaton computation in state q.

First we reverse the order of w in register 0 (since automata read words from the front and register machines read words from the back). We do this by reversing the content of

register 0 into register 1 via Example 4.9 (9). We then move into the subset Q_{q_0}, i.e., those states that correspond to the automaton being in the start state q_0.

Whenever the register machine gets into a state in Q_q, it reads and removes the final letter in register 1, say b, and then moves into a state in the subset $Q_{q'}$ where $q' = \delta(q, b)$. If there are no letters remaining in register 1, it either empties register 0 and halts (in case $q \notin F$) or empties register 0 and after that writes a into register 0 (in case $q \in F$). Q.E.D.

On Example Sheet # 3, we shall see that the *algorithms* that we exhibited in the proof of the solvability of the word problem for noncontracting grammars (Theorem 1.19) can be performed by a register machine which shows that if G is a type 1 grammar, $\mathcal{L}(G)$ is computable, yielding the following chain of implications:

$$\text{regular} \Rightarrow \text{context-free} \Rightarrow \text{type 1} \Rightarrow \text{computable} \Rightarrow \text{computably enumerable}.$$

This leaves the question where type 0 grammars fit into this chain of implications: we'll discuss this in § 4.8.

4.4 The shortlex ordering and its computability

Let us assume that our finite alphabet Σ comes with some ordering of the letters, i.e., $\Sigma = \{a_0, ..., a_n\}$ and

$$a_0 < a_1 < ... < a_n.$$

If $w, v \in \mathbb{W}$ and $w = b_0...b_k$ and $v = c_0...c_\ell$, we can define the following order relation:

$$\begin{aligned} w < v \quad :\Longleftrightarrow \quad & |w| < |v| \text{ or} \\ & |w| = |v| \text{ and } w \neq v \text{ and} \\ & \quad \text{if } i \text{ is mìnimal such that } b_i \neq c_i, \text{ then } b_i < c_i. \end{aligned}$$

This order relation is called the *shortlex order*. It is a total order, i.e., irreflexive (for no w, it is the case that $w < w$), transitive (if $u < v < w$, then $u < w$) and *trichotomous* (for any v and w, we either have $v < w$ or $w < v$ or $v = w$), and ε is its minimal element. We write $\text{pred}_<(w) := \{v \in \mathbb{W}\,;\, v < w\}$ for the initial segment of words smaller than w.

Theorem 4.14. The shortlex order $(\mathbb{W}, <)$ is isomorphic to $(\mathbb{N}, <)$.

We also say that the shortlex order and the natural numbers *have the same order type* or that the shortlex order has *order type* ω.[15]

Proof. If $w \in \mathbb{W}$ is any word, then the set $\text{pred}_<(w)$ is finite: all such v have length $\leq |w|$ and there are only finitely many words like this. So, there is a canonical map assigning a natural number to each such v, viz. if $v \mapsto n$ if v is the nth word in the shortlex ordering of $\text{pred}_<(w)$. Since $\text{pred}_<(w)$ is an initial segment of the entire shortlex ordering, if v is the nth word in the shortlex ordering of $\text{pred}_<(w)$, it is also the nth word in $(\mathbb{W}, <)$. This provides the canonical isomorphism denoted by the *hash symbol* #: $\#v = n$ if and only if there is some w such that v is the nth word in $\text{pred}_<(w)$. Q.E.D.

[15] In set theory, the symbol ω is customarily used for the smallest infinite ordinal number which is the same as the set of von Neumann natural numbers. For more details, cf. Part II *Logic & Set Theory*.

Notice that the isomorphism # provided in the proof of Theorem 4.14 is unique and allows us to identify words with natural numbers in a concrete way. We shall now show that the shortlex ordering is computable.

Proposition 4.15. The set $\{(v,w)\,;\, v < w\}$ and the function $s(w) := v$ where v is the $<$-immediate successor of w (i.e., $\#v = \#w + 1$) are computable.

Proof. The question "*What is the relationship of the lengths of the contents in registers i and j?*" with answers "the content of register i is shorter than the content of register j", "the content of register j is shorter than the content of register i", and "they are of equal length" can be answered by a register machine (copy the contents into scratch registers and remove letters one by one; if one of them is empty before the other, it's shorter; if they both become empty in the same step, they are of equal length).

If $|v| < |w|$, we empty register 0, add a to it, and halt; if $|w| < |v|$, we empty register 0 and halt; if they are of the same length, we copy the contents into scratch registers and remove letters one by one, checking whether they are the same; once we see that they are different, say $a = v(i) \neq w(i) = b$, we either empty register 0, add a to it, and halt (if $a < b$) or empty register 0 and halt (if $b > a$). Finally, if we emptied both registers without finding a difference, we empty register 0 and halt.

Finding the $<$-immediate successor of a word w with $|w| = k$ can be described as follows (using the ordering $a_0 < a_1 < ... < a_n$ on Σ): move all instances of the letter a_n from the back of the word into a scratch register (say, j) until you either hit an instance of $a_\ell \neq a_n$ or the register is empty. If the former, then change a_ℓ to $a_{\ell+1}$ and then add precise as many letters a_0 after this as you have letters in the scratch register j. If the latter, add precisely as many letters a_0 as you have letters in the scratch register j (that's k many) and after that add another a_0. Q.E.D.

4.5 Church's recursive functions

The following operations on partial functions were considered by Alonzo Church (1903–1995). The functions

$$\begin{aligned}
\pi_{k,i} &: \mathbb{W}^k \to \mathbb{W} : \vec{w} \to w_i \ (\textit{projection functions})\\
\mathrm{c}_{k,\varepsilon} &: \mathbb{W}^k \to \mathbb{W} : \vec{w} \to \varepsilon \ (\textit{constant functions})\\
s &: \mathbb{W} \to \mathbb{W} : w \mapsto v \quad (\text{where } \#(v) = \#(w) + 1;\ \text{the } \textit{successor function}).
\end{aligned}$$

are called *basic functions*. We have already proved that all basic functions are computable.

Suppose $f : \mathbb{W}^m \dashrightarrow \mathbb{W}$ and $g_1, ..., g_m : \mathbb{W}^k \dashrightarrow \mathbb{W}$ are partial functions, then the partial function h defined by

$$h(\vec{w}) := f(g_1(\vec{w}), ..., g_m(\vec{w}))$$

is called the *composition of f with $(g_1, ..., g_m)$*. The notational convention used for operations applies here as well: if any term on the right hand side is undefined, then so is the left hand side.

Suppose $f : \mathbb{W}^k \dashrightarrow \mathbb{W}$ and $g : \mathbb{W}^{k+2} \dashrightarrow \mathbb{W}$ are partial functions, then the function h defined by the recursion equations

$$\begin{aligned} h(\vec{w}, \varepsilon) &= f(\vec{w}) \text{ and} \\ h(\vec{w}, s(v)) &= g(\vec{w}, v, h(\vec{w}, v)) \end{aligned}$$

is called the *recursion result of f and g*.

Suppose $f : \mathbb{W}^{k+1} \dashrightarrow \mathbb{W}$ is a partial function, then the partial function h defined by

$$h(\vec{w}) := \begin{cases} v & \text{if for all } u \leq v, \text{ we have that } f(\vec{w}, u)\downarrow \text{ and} \\ & \quad v \text{ is } <\text{-minimal such that } f(\vec{w}, v) = \varepsilon \text{ or} \\ \uparrow & \text{otherwise} \end{cases}$$

is called the *minimisation result of f*.

We say that a class $\mathcal{C}$ of partial functions is closed under composition, recursion, or minimisation if, whenever $f, g, g_1, ..., g_m$ are in $\mathcal{C}$, then the composition of f with $(g_1, ..., g_m)$, the recursion result of f and g, or the minimisation result of f, respectively, are in $\mathcal{C}$.

Definition 4.16. The class of *primitive recursive (partial) functions* is the smallest class of partial functions containing all basic functions that is closed under composition and recursion. The class of *recursive (partial) functions* is the smallest class of partial functions containing all basic functions that is closed under composition, recursion and minimisation.

Example 4.17. We build a function using the basic functions and operations.

(a) The function $\pi_{1,0}(w) = w$ is a basic function and thus recursive.

(b) The function $\pi_{3,2}(w, v, u) = u$ is a basic function and thus recursive.

(c) The function s is a basic function and thus recursive.

(d) The function $s \circ \pi_{3,2}$ is a concatenation of recursive functions and thus recursive: $s \circ \pi_{3,2}(w, v, u) = s(u)$.

(e) We now apply recursion to the functions in (a) and (d), i.e.,

$$\begin{aligned} h(w, \varepsilon) &= \pi_{1,0}(w) \\ h(w, s(v)) &= s(\pi_{3,2}(w, v, h(w, v))) = s(h(w, v)). \end{aligned}$$

The function h defined by these recursion equations is primitive recursive.

The recursion equations given in (e) are the so-called *Grassmann equations for addition* (written in ordinary natural number notation: $h(n, 0) := n$ and $h(n, m + 1) := h(n, m) + 1$). So, in our setting, we have $\#(h(w, v)) = \#w + \#v$. Thus, $h(w, v)$ is the unique word such that its shortlex number is the sum of the shortlex numbers of w and v.

Similar recursions using the Grassmann equations for multiplication and exponentiation, respectively, allow us to show that the standard arithmetical functions such as $m(w, v)$ with $\#(m(w, v)) = \#w \cdot \#v$ and $e(w, v)$ with $\#(e(w, v)) = \#w^{\#v}$ are primitive recursive.

Lemma 4.18. Suppose $f : \mathbb{W}^{k+1} \dashrightarrow \mathbb{W}$ is a recursive partial function, then the partial function h defined by

$$h(\vec{w}) := \begin{cases} v & \text{if for all } u \leq v, \text{ we have that } f(\vec{w}, u)\downarrow \text{ and} \\ & \quad v \text{ is } <\text{-minimal such that } f(\vec{w}, v) \neq \varepsilon \text{ or} \\ \uparrow & \text{otherwise} \end{cases}$$

is recursive.

Proof. As in the proof of Proposition 4.12 (a), consider $g(\varepsilon) = a$ and $g(w) = \varepsilon$ for any $w \neq \varepsilon$. Then $g \circ f$ is computable and the minimisation result of $g \circ f$ is the desired function h. Q.E.D.

Let $T \subseteq \mathbb{N}^*$ be a finitely branching tree (cf. § 3.1). For $t \in T$, we write $\mathrm{succ}_T(t) := \{s \in T\,;\, s \text{ is an immediate successor of } t\}$. As for our parse trees in § 3.1, we assign a labelling function ℓ; each label comes with an *arity* and a *branching number* listed in Table 2.

Definition 4.19. A *recursion tree* (T, ℓ) is a non-empty finite tree together with a *labelling function* ℓ with $\mathrm{dom}(\ell) = T$ such that each value of ℓ is a label with the additional properties that

(i) for every $\alpha \in T$, $|\mathrm{succ}_T(\alpha)|$ is the branching number of $\ell(\alpha)$;

(ii) if $\ell(\alpha) = \mathsf{C}_{n,k}$, then the first successor of α has a label with arity n and all other successors of α have labels of arity k;

(iii) if $\ell(\alpha) = \mathsf{R}_k$, then the first successor of α has a label of arity k and the second successor has a label or arity $k + 2$;

(iv) if $\ell(\alpha) = \mathsf{M}_k$, then the unique successor of α has a label of arity $k + 1$.

A recursion tree is called *primitive* if no M-labels occur.

By recursion on the height of the tree, we can assign functions $f_{T,\ell}$ to each recursion tree (T, ℓ) such that the arity of the function assigned to the root is equal to the arity of the label $\ell(\varepsilon)$:

Label	Arity	Branching number	Interpretation
$\mathsf{B}^{\pi}_{k,i}$	k	0	Projection
$\mathsf{B}^{\mathrm{c}}_{k}$	k	0	Constant
B^{s}	1	0	Successor
$\mathsf{C}_{n,k}$	k	$n + 1$	Composition
R_k	$k + 1$	2	Recursion
M_k	k	1	Minimisation

Table 2: Labels for recursion trees and their arities and branching numbers

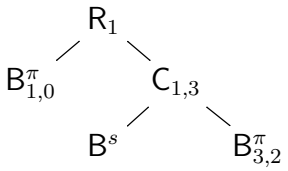
(a) By construction, a recursion tree has height 0 if and only if $\ell(\varepsilon)$ is a basic label, i.e., $\mathsf{B}^{\pi}_{k,i}$, $\mathsf{B}^{\mathrm{c}}_{k}$, or B^{s}. In this case, let $f_{T,\ell} = \pi_{k,i}$, $f_{T,\ell} = \mathrm{c}_{k,\varepsilon}$, or $f_{T,\ell} = s$, respectively.

(b) Suppose the height of the tree is $m+1$ and $\ell(\varepsilon) = \mathsf{C}_{n,k}$. Recursively, we assume that the construction is already done for all trees of height $\leq m$. Note that all of the subtrees starting with the immediate successors of ε are recursion trees with height $\leq m$, so we have already assigned functions of the right arity to them. By construction, ε has $n+1$ successors: the first one is assigned a function f of arity n, and all others are assigned functions g_i of arity k. We let $f_{T,\ell}$ be the composition of f with $(g_1, ..., g_n)$.

(c) Suppose the height of the tree is $m+1$ and $\ell(\varepsilon) = \mathsf{R}_k$. By construction, ε has two successors: the first one is assigned a function f of arity k and the second one is assigned a function g with arity $k+2$. We let $f_{T,\ell}$ be the recursion result of f and g.

(d) Suppose the height of the tree is $m+1 > 1$ and $\ell(\varepsilon) = \mathsf{M}_k$. By construction, ε has a unique successor that is assigned a function f of arity $k+1$. We let $f_{T,\ell}$ be the minimisation result of f.

The following recursion tree gives us the Grassmann definition of addition:



Theorem 4.20. A partial function f is recursive if and only if there is a non-empty recursion tree (T, ℓ) such that $f = f_{T,\ell}$; it is primitive recursive if and only if there is a primitive recursion tree with that property.

Proof. Clearly, all basic functions are represented by recursion trees and the class of all partial functions represented by recursion trees is closed under composition, recursion, and minimisation. Since the recursive functions were the smallest class with these properties, we get that every recursive function is represented by a recursion tree.

Assume towards a contradiction that there is a function represented by a recursion tree (T, ℓ) that is not recursive. Let's assume that (T, ℓ) is a counterexample of minimal height, i.e., all shorter trees represent recursive functions. Since functions represented by trees of height 1 are clearly recursive (they are basic functions), the height must be > 1, so the label of ε is either a C, R, or M label, so by definition $f_{T,\ell}$ is the result of either composition, recursion, or minimisation applies to the functions associated with the immediate successors of ε. But the immediate successors determine trees of strictly lower height, so by induction hypothesis, these functions are all recursive. Since the class of recursive functions is closed under the three operations, $f_{T,\ell}$ is recursive. Contradiction! Q.E.D.

Corollary 4.21. Every primitive recursive function is total.

Proof. This follows directly from Theorem 4.20 by induction on the height of the primitive recursion tree because all basic functions are total and the operations composition and recursion preserve totality. Q.E.D.

Theorem 4.22. Every recursive function is computable.

Proof. We have already established that the basic functions are all computable (Example 4.11 (2) & (3) and Proposition 4.15) and that the computable functions are closed under composition (Lemma 4.6), so we only need closure under recursion and minimisation. Together this shows that the computable functions form a class of functions containing the basic functions and closed under all three operations. Since the recursive functions are the smallest such class, they are contained in the computable functions.

Recursion. Suppose f and g are computable and that $h(\vec{w}, \varepsilon) = f(\vec{w})$ and $h(\vec{w}, s(v)) = g(\vec{w}, v, h(\vec{w}, v))$. Fix $\vec{w}$ and v and describe how to compute $h(\vec{w}, v)$: we use two registers that will not be needed otherwise, say, registers k and ℓ, and empty them. We then calculate $f(\vec{w})$ and write it into register ℓ. In each step of the computation, we check whether v is equal to the content of register k. It this happens to be the case at the beginning of the computation (i.e., when register k is empty), then we just output $f(\vec{w})$ and halt. If not, we repeat the following routine: we apply the successor function (which is computable by Proposition 4.15) to the content of register k and calculate $g(\vec{w}, v, u)$ where u is the current context of register ℓ and write this into register ℓ. If w is equal to the content of register k, then we output what is in register ℓ. Otherwise, we go back to the beginning of the routine. By Theorem 4.14, this loop will eventually reach a point when w is equal to the content of register k and therefore, the computation will eventually halt (unless one of the f- or g-calculations fails to halt).

Minimisation. Assume that f is computable. Use a register that is not needed otherwise, say, register k and empty it. Now apply the following routine iteratively: Check whether f applied to $(\vec{w}, u)$ halts, where u is the current content of register k; if it does, check whether $f(\vec{w}, u) = \varepsilon$; if so, then output the current content of register k. If it halted, but is not empty, apply the (computable) successor function to the content of register k and restart the routine. Q.E.D.

Among other things, Theorem 4.22 gives us computable access to the arithmetical functions (addition, multiplication, exponentiation) since we mentioned earlier that they are primitive recursive.

Splitting & merging words. We use our access to arithmetical functions to define splitting and merging operations on words. Consider the arithmetical function

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

which is the well-known Cantor *zigzag bijection* that Cantor used to prove the countability of the rational numbers $\mathbb{Q}$. This function is a composition of the basic arithmetical functions

that we already established are primitive recursive. So, the maps $(v, w) \mapsto u$ if $\#u = z(\#v, \#w)$ is a primitive recursive function. We write $v * w := u$ and call this operation *merging v and w into a single word*.

The merging function is a total computable bijection between $\mathbb{W}^2$ and $\mathbb{W}$. It's inverse taking a word u and finding v and w such that $v * w = u$ can also be performed by a register machine: note that we know that these words must exist, since the Cantor zigzag function is a bijection and that if the formula is valid, then $u, v < w$, so we only need to search through finitely many possible values of u and v. This operation is called *splitting u into two words*. It gives rise to two computable total functions $\cdot_{(0)} : \mathbb{W} \to \mathbb{W}$ and $\cdot_{(1)} : \mathbb{W} \to \mathbb{W}$ such that $u_{(0)} * u_{(1)} = u$.

4.6 Remark on the choice of alphabet

Not lectured

We defined computability for partial functions $f : \mathbb{W}^k \dashrightarrow \mathbb{W}$ in terms of Σ-register machines: the instructions and behaviour of register machines are closely tied to their alphabet and register machines can only compute partial functions that use the letters that the machines are built for. Clearly, if $\Sigma \subseteq \Sigma'$ and $f : \mathbb{W}^k \dashrightarrow \mathbb{W}$ is computable by a Σ-register machine, then it is computable by a Σ'-register machine. But could it be that the notion of computability gets stronger if we add more letters to the alphabet? The answer is no as will be shown in this section.

We shall encode computations in binary notation. For this, let us assume that we have two special symbols $\mathbf{0}$ and $\mathbf{1}$ in Σ. Suppose $2 \leq n = |\Sigma|$ and k is such that $2^{\mathsf{k}} \geq n$. Then we can represent the elements of Σ by binary sequences of length m by using our favourite injection i from Σ into $\{\mathbf{0}, \mathbf{1}\}^k$. The injection i induces an injection (also denoted by i) from $\mathbb{W}$ into $(\{0, 1\}^m)^* \subseteq \{\mathbf{0}, \mathbf{1}\}^* \subseteq \mathbb{W}$. We extend that induced injection further to injections $i : \mathbb{W}^n \to \mathbb{W}^n$, defined componentwise and again using the same notation.

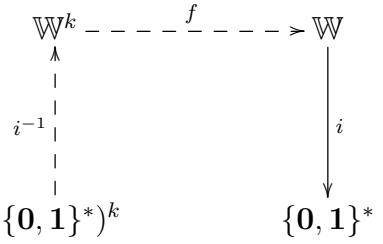
Lemma 4.23. The injection $i : \mathbb{W} \to \mathbb{W}$ is computable and so is its inverse $i^{-1} : \mathbb{W} \dashrightarrow \mathbb{W}$ (which has domain $(\{\mathbf{0}, \mathbf{1}\}^m)^*$).

Proof. We can easily write a register machine program that removes the final letter of register k, say, a and copies $i(a)$ in reverse order into register ℓ. Repeating this until register k is empty results in the reverse of the i-image of the original content of register k to be stored in register ℓ. Now reverse the order and you obtain the i-value of the content of register k.

For the inverse, we do the same except that the program reads m many letters from the content of the register k, check that it's an element of $\{\mathbf{0}, \mathbf{1}\}^k$ (if not, we loop forever) and writes the i-preimage of that string into register ℓ. The rest of the construction is the same. Q.E.D.

Using the map i, we can represent a partial function on $\mathbb{W}$ by a partial function on $\{\mathbf{0}, \mathbf{1}\}^*$

as follows:



Let us write $\widehat{f}$ for this partial function $i \circ f \circ i^{-1}$.

Proposition 4.24. The partial function f is computable by a Σ-register machine if and only if the partial function $\widehat{f}$ is computable by a $\{\mathbf{0},\mathbf{1}\}$-register machine.

Proof. If M is the Σ-register machine computing f, all we need to do is to replace all instructions by sequences of instructions that do the same for the represented sequences. I.e., if the instruction is $+(\ell, a, q)$ we replace it with m many instructions that add the m bits that form $i(a)$ to register ℓ; if the instruction is $?(\ell, a, q, q')$, we replace it with a sequence of instructions that reads the final m bits from register ℓ and checks whether this sequence is $i(a)$; if the instruction is $-(\ell, q, q')$, we remove the final m bits from register ℓ instead. The instruction $?(\ell, \varepsilon, q, q')$ can remain unchanged. Q.E.D.

Corollary 4.25. Suppose $\{\mathbf{0},\mathbf{1}\} \subseteq \Sigma \subseteq \Sigma'$ and $f : \mathbb{W}^k \dashrightarrow \mathbb{W}$ is computable by a Σ'-register machine. Then it is computable by a Σ-register machine.

Proof. We consider f as a partial function from $((\Sigma')^*)^k$ to $(\Sigma')^*$ and apply Proposition 4.24, making use of an appropriate injection $i : \Sigma' \to \{\mathbf{0},\mathbf{1}\}^m$. This gives us a $\{\mathbf{0},\mathbf{1}\}$-register machine that computes $\widehat{f}$. Consider $j = i{\restriction}\Sigma : \Sigma \to \{\mathbf{0},\mathbf{1}\}^m$. The injection j and the induced injections for $\mathbb{W}$ and $\mathbb{W}^k$ as well as all of the partial inverses are computable by a Σ-register machine by Lemma 4.23. But $f = j^{-1} \circ \widehat{f} \circ j$, so f is computable by a Σ-register machine. Q.E.D.

Corollary 4.25 allows us to use the word *computable* without referring to the alphabet. It also allows us to extend the alphabet with additional letters for the convenience of proofs and show that a function $f : \mathbb{W}^k \to \mathbb{W}$ is computable by a machine using these additional letters: Corollary 4.25 tells us that these additional letters are not really needed since they can be coded away appropriately.

4.7 Software and universality

Coding in an expanded alphabet. Fix an alphabet Σ and enlarge it by new symbols to a larger alphabet Σ':

$$\mathbf{0} \quad \mathbf{1} \quad \boldsymbol{\varepsilon} \quad \mathbf{+} \quad \mathbf{?} \quad \mathbf{-} \quad \mathbf{(} \quad \mathbf{)} \quad \mathbf{,} \quad \mathbf{\mapsto} \quad \mathbf{\square}.$$

We'll use these symbols to encode all of the elements of our descriptions of Σ-register machines in $\mathbb{W}' := (\Sigma')^*$. At the end, we'll use the coding techniques from § 4.6 to encode these elements

in $\mathbb{W}$. We use the notation code' for the coding function mapping into $\mathbb{W}'$ and the notation code for the coding function mapping into $\mathbb{W}$.

First of all, a natural number k will be represented in binary notation using **0** and **1**, e.g., 13 will be represented by **1101**; we write $\mathrm{code}'(k)$ for this word. The ability to refer directly to natural numbers with words (rather than via the #-function that we get from the shortlex ordering) allows us to write arithmetical functions in a more direct way. Note that the function $w \mapsto \mathrm{code}'(k)$ where $\#(w) = k$ is computable (use recursion) and has a computable inverse. This means that the function $h(\mathrm{code}'(k), \mathrm{code}'(\ell)) := \mathrm{code}'(k+\ell)$ is computable: find w and v such that $\#(w) = k$ and $\#(v) = \ell$, use the arithmetical functions defined in § 4.5 to obtain u such that $\#(u) = k + \ell$, and transform u into $\mathrm{code}'(k+\ell)$.

Similarly, we shall represent states by binary sequences: as we have seen before, the actual set of states does not matter for a register machine, only its size. As a consequence, we can assume w.l.o.g. that the states of a register machine are binary number representations; again, we write $\mathrm{code}'(q)$ for the word in $\{\mathbf{0}, \mathbf{1}\}^*$ that represents the state q.

Instructions are represented by the obvious string of letters in $\mathbb{W}'$ using **+**, **?**, and **−** to represent the types of instructions. E.g., $+(k, q, q')$ will be represented by the word $\mathbf{+}(\mathrm{code}'(k)\mathbf{,}\mathrm{code}'(q)\mathbf{,}\mathrm{code}'(q')\mathbf{)}$. If I is an instruction, we once more write $\mathrm{code}'(I)$ for the word representing it.

A program line of the form $q \mapsto P(q)$ will be represented by $\mathrm{code}'(q) \mapsto \mathrm{code}'(P(q))$, and finally, a register machine $M = (\Sigma, Q, P)$ will be represented by the word

$$\mathrm{code}'(q_0) \mapsto \mathrm{code}'(P(q_0))\mathbf{,}\mathrm{code}'(q_1) \mapsto \mathrm{code}'(P(q_1))\mathbf{,}...\mathbf{,}\mathrm{code}'(q_n) \mapsto \mathrm{code}'(P(q_n))$$

if Q has $n+1$ elements; we write $\mathrm{code}'(M)$ for that word.

We encode a sequence $\vec{w} \in \mathbb{W}^{n+1}$ by a single word in $\mathbb{W}'$, viz. $w_0\square...\square w_n\square$; we write $\mathrm{code}'(\vec{w})$ for this word. If $q \in Q$ and $\vec{w} \in \mathbb{W}^{n+1}$, we encode the configuration $C = (q, \vec{w})$ by the word $\mathrm{code}'(q)\square\mathrm{code}'(\vec{w})$; once more, we write $\mathrm{code}'(C)$ for this word.

Coding in the original alphabet. Using the results of § 4.6, we can encode words from $\mathbb{W}'$ by words in $\mathbb{W}$ in computable way; write $c : \mathbb{W}' \to \mathbb{W}$ for the encoding function and define $\mathrm{code} := c \circ \mathrm{code}'$. So, all codes of the form $\mathrm{code}(k)$, $\mathrm{code}(q)$, $\mathrm{code}(I)$, $\mathrm{code}(q \mapsto P(q))$, $\mathrm{code}(M)$, $\mathrm{code}(\vec{w})$, and $\mathrm{code}(C)$ are elements of $\mathbb{W}$, viz. the c-image of the concrete codes in $\mathbb{W}'$ defined above.

Register machines can determine whether something is a code of the right type, i.e., whether a word $w \in \mathbb{W}$ is a code for a number, a state, an instruction, a program line, a register machine, a configuration, or a sequence of words. Similarly, register machines can answer all relevant concrete questions about these objects, e.g., in which state a given configuration is or what the instruction of a register machine for a given state is, etc. In the following, when we write $f\colon \mathrm{code}(x) \mapsto y$ we mean the operation that is defined precisely when the input is of the form $\mathrm{code}(x)$ for some x of the right type (where the type of x is implicitly determined by the letter we use for it).

Just look at the old notes for this section up to software principle, they match what was lectured and thus presumably what's examinable.

Lemma 4.26. The *transformation function*

$$f_{\mathrm{T}}\colon \mathbb{W}^2 \dashrightarrow \mathbb{W}\colon (\mathrm{code}(M), \mathrm{code}(C)) \mapsto \mathrm{code}(C') \text{ if } M \text{ transforms } C \text{ to } C'$$

is computable.

Proof. Let $C = (q, \vec{w})$. The code of M contains information about what $P(q)$ is; apply that concrete operation corresponding to the instruction $P(q)$ to $\vec{w}$ as given on p. 45. Q.E.D.

Lemma 4.27. The *computation sequence function*

$$f_{\mathrm{CS}} \colon \mathbb{W}^3 \dashrightarrow \mathbb{W} \colon (\mathrm{code}(M), \mathrm{code}(\vec{w}), v) \mapsto \mathrm{code}(C(M, \vec{w}, \#v))$$

is computable.

Configuration after #v steps.

Proof. We have seen that the computable functions are closed under recursion (proof of Theorem 4.22), so we define this by recursion via

$$f_{\mathrm{CS}}(\mathrm{code}(M), \mathrm{code}(\vec{w}), \varepsilon) := \mathrm{code}(q_{\mathrm{S}}, \vec{w}),$$
$$f_{\mathrm{CS}}(\mathrm{code}(M), \mathrm{code}(\vec{w}), s(v)) := f_{\mathrm{T}}(f_{\mathrm{CS}}(\mathrm{code}(M), \mathrm{code}(\vec{w}), v))$$

where f_{T} is the transformation function from Lemma 4.26. Q.E.D.

We define the sets $T \subseteq \mathbb{W}^{k+2}$ and $\widehat{T} \subseteq \mathbb{W}^{2k+2}$ as follows:

$$T := \{(\mathrm{code}(M), \vec{w}, u)\,;\, M \text{ has halted with input } \vec{w} \text{ after at most } \#u \text{ steps}\} \text{ and}$$
$$\widehat{T} := \{(\mathrm{code}(M), \vec{w}, u, \vec{v})\,;\, (\mathrm{code}(M), \vec{w}, u) \in T \text{ and } \vec{v} \text{ is the register content at time of halting}\}.$$

Corollary 4.28. The sets T and $\widehat{T}$ are computable.

Proof. We describe the characteristic function of $\widehat{T}$; the case of T is the same except for the check of the register content at time of halting. Use an initially empty scratch register k and repeat the following subroutine until register k contains $s(u)$. Once this happens, empty register 0 and halt (i.e., the input was not in $\widehat{T}$).

In the subroutine, let t be the content of register k. Use the computation sequence function f_{CS} from Lemma 4.27 to calculate $C(M, \vec{w}, t) = (q, \vec{s})$. If $q = q_{\mathrm{H}}$, check whether $\vec{s} = \vec{v}$; if so, write a in register 0 and halt (i.e., the input was in $\widehat{T}$); otherwise empty register 0 and halt (i.e., the input was not in $\widehat{T}$). If $q \neq q_{\mathrm{H}}$, apply the successor function to the register content of register k and end the subroutine.

Note that this procedure will always terminate in a finite number of steps due to Theorem 4.14. Q.E.D.

From the sets T and $\widehat{T}$, we derive

$$T_M := \{(\vec{w}, u)\,;\, (\mathrm{code}(M), \vec{w}, u) \in T\},$$
$$T_v := \{(\vec{w}, u)\,;\, (v, \vec{w}, u) \in T\},$$
$$\widehat{T}_M := \{(\vec{w}, u, \vec{v})\,;\, (\mathrm{code}(M), \vec{w}, u, \vec{v}) \in \widehat{T}\}, \text{ and}$$
$$\widehat{T}_v := \{(\vec{w}, u, \vec{v})\,;\, (v, \vec{w}, u, \vec{v}) \in \widehat{T}\},$$

all of which are computable by Corollary 4.28. We call these sets *truncated computation sets* and their characteristic functions *truncated computation functions*.

Theorem 4.29 (The Software Principle)**.** There is a Σ-register machine U, called a *universal Σ-register machine* such that for every Σ-register machine M and sequence of words $\vec{w}$, we have that

$$f_{U,2}(v,u) = \begin{cases} f_{M,k}(\vec{w}) & \text{if } v = \mathrm{code}(M) \text{ for a } \Sigma\text{-register machine } M \\ & \quad \text{and } u = \mathrm{code}(\vec{w}) \text{ for a sequence of words of length } k, \\ \uparrow & \text{otherwise.} \end{cases}$$

Theorem 4.29 tells us that there is a single register machine that can mimic the behaviour of all register machines. This is quite remarkable since the universal register machine is a finite object and, in particular, has a fixed upper register index and a fixed number of states and instructions. The register machines whose behaviour it can mimic can use many more registers than U and can be a lot bigger than U in terms of the number of states. But U will need this information in the input (since it uses $\mathrm{code}(M)$ as part of its input data) and so we have moved the additional registers and states that would require a much larger machine than U into the realm of *software* (hence the name). We can think of U as the actual machine with its storage space and universal program and of $\mathrm{code}(M)$ as the software that is being installed on U to run the program that produces $f_{M,k}$.

Proof. We describe the register machine U by the operations it performs: at the beginning, we check whether v is a code for a register machine and whether u is a code for a sequence of words; if not, we diverge. Now use initially empty scratch registers k, ℓ, and m and repeat the following subroutine until register ℓ contains $\mathrm{code}(q_{\mathrm{H}})$. Once this happens, output the register content of register m.

In the subroutine, let t be the content of register k. Use the computation sequence function f_{CS} from Lemma 4.27 to calculate $C(M, \vec{w}, t) = (q, \vec{s})$. Write $\mathrm{code}(q)$ into register ℓ and s_0 (i.e., the register content in register 0 at time $\#t$) into register m.

Note that this repeat loop does not always terminate: it terminates if and only if $f_M, k(\vec{w})\!\downarrow$. Q.E.D.

Theorem 4.29 allows us to simplify our notation in a natural way: instead of using the register machine M as parameter of our computable functions, we can define for arbitrary words v

$$f_{v,k}(\vec{w}) := f_{U,2}(v, \mathrm{code}(\vec{w})).$$

If $v = \mathrm{code}(M)$, this partial function coincides with $f_{M,k}$; if v is not the code of a register machine, it'll give the nowhere defined partial function. We extend this notation to the computably enumerable sets W_M by writing $\mathrm{W}_w := \mathrm{dom}(f_{w,1})$. This parametrises all computably enumerable sets in a single list.

Theorem 4.30 (The *s-m-n* Theorem)**.** Let $g : \mathbb{W}^{k+1} \dashrightarrow \mathbb{W}$ be any partial computable function. Then there is a total computable function $h : \mathbb{W} \to \mathbb{W}$ such that for all $v \in \mathbb{W}$ and all $\vec{w} \in \mathbb{W}^k$, we have $f_{h(v),k}(\vec{w}) = g(\vec{w}, v)$.

The curious name of this theorem derives from the notation S_n^m used for the function h in the original publication.[16] The *s-m-n* Theorem pulls one of the parameters of the

[16] Cf. S. C. Kleene (1938), On notation for ordinal numbers, *Journal of Symbolic Logic* 3 (4): 150–155; p. 153.

function g into the index. This process is also called *Currying*, after the logician Haskell Curry (1900–1982).[17]

Proof. Clearly, for a fixed v, the function $g_v : \vec{w} \mapsto g(\vec{w}, v)$ is computable, so there is some word u such that $f_{u,k}(\vec{w}) = g(\vec{w}, v)$. However, what we need to establish here is that a register machine can find such a u.

For a fixed v, the operation $\vec{w} \mapsto (\vec{w}, v)$ is performed by a register machine M_v: the register machine consists of the instructions that add the word v letter by letter into register k. We can explicitly construct a register machine that performs the operation $v \mapsto \mathrm{code}(M_v)$.

Since g is computable, there is a register machine M that computes it, i.e., $f_{M,k+1}(\vec{w}, v) = g(\vec{w}, v)$. This means that for a fixed word v, the function g_v is performed the concatenation of the two register machines M_v and M. In the comment after Lemma 4.6, we highlighted that the concatenation of register machines is a concrete operation that provides a definition for the register machine that performs the concatenated operation (making the state sets disjoint and replacing the halt state of the first register machine with the start state of the second). For register machines M_0 and M_1, let us write $M_0 \circ M_1$ for their concatenation machine from the proof of Lemma 4.6. In the terminology of this section, the remark means that the operation $(\mathrm{code}(M_1), \mathrm{code}(M_0)) \mapsto \mathrm{code}(M_0 \circ M_1)$ is performed by a register machine.

Clearly, the operation $w \mapsto (w, \mathrm{code}(M))$ is performed by a register machine (viz. the case $k = 1$ and $v = \mathrm{code}(M)$ of the machine M_v above). Thus, fitting all of these together,

$$v \mapsto \mathrm{code}(M_v) \mapsto (\mathrm{code}(M_v), \mathrm{code}(M)) \mapsto \mathrm{code}(M \circ M_v)$$

is performed by a register machine and thus $h(v) := \mathrm{code}(M \circ M_v)$ is a total computable function with

$$f_{h(v),k}(\vec{w}) = f_{M \circ M_v,k}(\vec{w}) = g_v(\vec{w}) = g(v, \vec{w}).$$

Q.E.D.

The Recursion Theorem. We close this section by mentioning another important result that follows from the conceptual work performed here: the *Recursion Theorem* or *Fixed Point Theorem*. If $\varphi\colon \mathbb{W} \dashrightarrow \mathbb{W}$ and $w \in \mathbb{W}$, we call w a *fixed point of* φ if $f_{\varphi(w),1} = f_{w,1}$.

Theorem 4.31 (Recursion Theorem or Fixed Point Theorem)**.** If $\varphi\colon \mathbb{W} \to \mathbb{W}$ is total, then φ has a fixed point.

This theorem is not lectured in this course since students are asked to prove it (with a useful hint) on Example Sheet #4. Theorem 4.31 allows us to find interesting examples of words, e.g., a word w such that $\mathrm{W}_w = \{w\}$; again, this will be discussed on Example Sheet #4. The Fixed Point Theorem also plays an important role in the constructions of unprovable sentences in the proof of *Gödel's Incompleteness Theorem*.

[17] This is conceptually related to the fact that functions from $X \times Y$ into Z can be considered as functions from X into the set of functions from Y into Z, sometimes referred to as *Curry-Howard Correspondence*; arithmetically, this is just the equality $z^{yx} = (z^y)^x$.

4.8 Computably enumerable sets

Using our universal register machine, we can now get the most important computably enumerable set, the *halting problem* and its two-variable variant:

$$\mathbf{K} := \{w\,;\, f_{w,1}(w)\downarrow\} \text{ and}$$
$$\mathbf{K}_0 := \{(w,v)\,;\, f_{w,1}(v)\downarrow\}.$$

Theorem 4.32. The sets $\mathbf{K}$ and $\mathbf{K}_0$ are computably enumerable.

Proof. By Proposition 4.12, we only need to show that they are domains of a computable functions. By Theorem 4.29, we have that $f_{U,2}(w,v) = f_{w,1}(v)$, so $\mathbf{K}_0 = \mathrm{dom}(f_{U,2})$. The operation $w \mapsto (w,w)$ can be performed by a register machine; hence, $f : w \mapsto f_{U,2}(w,w)$ is computable and $\mathbf{K} = \mathrm{dom}(f)$. Q.E.D.

Theorem 4.33. The sets $\mathbf{K}$ and $\mathbf{K}_0$ are not computable. In particular, they are computably enumerable set that are not computable.

Proof. Suppose either of them is, i.e., $\chi_{\mathbf{K}}$ or $\chi_{\mathbf{K}_0}$ are computable functions. Define

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

This is clearly computable, so let $d \in \mathbb{W}$ be a word such that $f_{d,1} = f$. Then we have that

$$\begin{aligned} f(d)\uparrow \iff \chi_{\mathbf{K}}(d) = a \iff \chi_{\mathbf{K}_0}(d,d) = a &\iff (d,d) \in \mathbf{K}_0 \\ &\iff d \in \mathbf{K} \iff f_{d,1}(d)\downarrow \iff f(d)\downarrow. \end{aligned}$$

Contradiction! Q.E.D.

Definition 4.34. A set $X \subseteq \mathbb{W}^k$ is called Σ_1 if there is a computable set $Y \subseteq \mathbb{W}^{k+1}$ such that for all $\vec{w} \in \mathbb{W}^k$, we have

$$\vec{w} \in X \iff \exists v((\vec{w}, v) \in Y).$$

It is called Π_1 if it is the complement of a Σ_1 set; it is called Δ_1 if it is both Σ_1 and Π_1.

The terminology derives from the fact that Σ_1 sets are defined using one existential quantifier and logicians tend to think of existential quantifiers as analogues of sums; similarly, Π_1 sets are defined using one universal quantifier and logicians tend to think of these as analogies of products. The letter Δ comes from the German word "*Durchschnitt*" (intersection) since the class of Δ_1 sets is the intersection of the classes of Σ_1 and Π_1 sets.

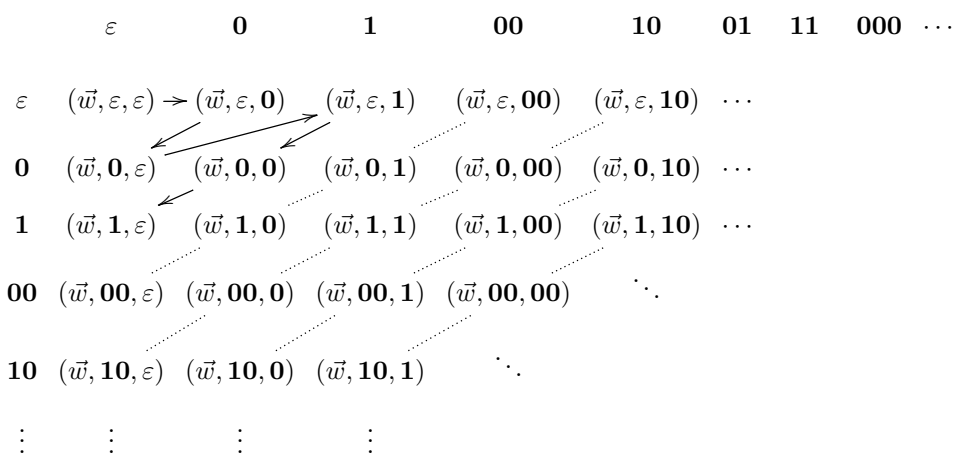


Figure 5: The search space $\mathbb{W}^2$ (or, more precisely, $\{\vec{w}\} \times \mathbb{W}^2$) traversed by Cantor's zigzag function as in the argument of Example 4.37.

Proposition 4.35. Every computable set is Δ_1.

Proof. By Proposition 4.12 (a), we only need to show that every computable set is Σ_1. Let $X \subseteq \mathbb{W}^k$ be computable. Define Y by $(\vec{w}, v) \in Y$ if and only if $\vec{w} \in X$. The set Y is clearly computable (ignore v and do the computation to check that $\vec{w} \in X$). But then $\vec{w} \in X$ if and only of there is a v such that $(\vec{w}, v) \in Y$ by definition. Q.E.D.

Theorem 4.36. Let $X \subseteq \mathbb{W}^k$. Then the following are equivalent:

(i) The set X is computably enumerable and

(ii) the set X is Σ_1.

Proof. "(i)⇒(ii)." Let $X = \mathrm{dom}(f)$ and let M be a register machine that computes f and consider the truncated computation set T_M which is computable by Corollary 4.28. Then

$$\vec{w} \in \mathrm{dom}(f) \iff \exists v((\vec{w}, v) \in T_M).$$

"(ii)⇒(i)." Let $Y \subseteq \mathbb{W}^{k+1}$. Using Lemma 4.18, apply minimisation to χ_Y to find a function $h : \mathbb{W}^k \dashrightarrow \mathbb{W}$ that searches for the least v such that $(\vec{w}, v) \in Y$. Thus, $X = \mathrm{dom}(h)$ and so X is computably enumerable. Q.E.D.

The following results use a method we call the *zigzag method*: using Cantor's zigzag function, we can line up the elements of $\mathbb{W}^2$ in an ordering of order type ω (i.e., isomorphic to $(\mathbb{N}, <)$; cf. Footnote 15) and thus fold two existential quantifiers into one. We can visualise the computation process as running through $\mathbb{W}^2$ via Cantor's zigzag function and checking the relevant question for each pair in the ordering that is determined by the zigzag function as depicted in Figure 5.

More specifically, if $Y \subseteq \mathbb{W}^{k+2}$ is computable, then

$$X := \{\vec{w}\,;\, \exists v \exists u((\vec{w}, v, u) \in Y)\}$$

is computably enumerable: write $Z := \{(\vec{w}, v)\,;\, ((\vec{w}, v_{(0)}, v_{(1)}) \in Y)\}$. This is computable since the two component functions of the splitting operation are. Then

$$\begin{aligned} \vec{w} \in X &\iff \exists v \exists u((\vec{w}, v, u) \in Y) \\ &\iff \exists v((\vec{w}, v_{(0)}, v_{(1)}) \in Y) \\ &\iff \exists v((\vec{w}, v) \in Z), \end{aligned}$$

whence X is Σ_1 and so computably enumerable by Theorem 4.36.

Example 4.37. If $f : \mathbb{W}^{k+1} \dashrightarrow \mathbb{W}$ is computable, then the set

$$X := \{\vec{w} \in \mathbb{W}^k\,;\, \exists v(f(\vec{w}, v)\!\downarrow)\}$$

is computably enumerable.

[Let M be such that $f_{M,k+1} = f$. Consider the truncated computation set T_M which is computable by (the remark after) Corollary 4.28, i.e., $(\vec{w}, v, u) \in T_M$ if and only if the computation of M with input $\vec{w}$ has halted after at most $\#u$ steps. Clearly, $\vec{w} \in X$ if and only if there are v and u such that $(\vec{w}, v, u) \in T_M$, so the zigzag method yields that X is computably enumerable.]

The following important results are further applications of the zigzag method.

Proposition 4.38. Let $\varnothing \neq X \subseteq \mathbb{W}$. Then X is computably enumerable if and only if there is a computable $g\colon \mathbb{W} \dashrightarrow \mathbb{W}$ such that $X = \mathrm{ran}(g)$.

Proof. For the forward direction, let $f : \mathbb{W} \dashrightarrow \mathbb{W}$ be computable; then so is

$$g(w) := \begin{cases} w & \text{if } f(w)\!\downarrow \text{ and} \\ \uparrow & \text{otherwise.} \end{cases}$$

Clearly, $\mathrm{ran}(g) = \mathrm{dom}(g) = \mathrm{dom}(f) = X$.

For the other direction, let $X = \mathrm{ran}(g)$ and let M be a register machine such that $g = f_{M,1}$. Then consider the truncated computation set $\widehat{T}_M$, i.e., $(v, u, w) \in \widehat{T}_M$ if and only if the computation with machine M and input v has halted in at most $\#u$ steps and produced the output w, and observe that

$$w \in X \iff \exists v \exists u((v, u, w) \in \widehat{T}_M),$$

so by the zigzag method, X is Σ_1 and thus computably enumerable by Theorem 4.36. Q.E.D.

Proposition 4.39. A set is computable if and only if it is Δ_1.

Proof. The forward direction was proved in Proposition 4.35. Thus, assume that X is Δ_1, i.e., by Theorem 4.36, there are register machines M and M' such that $X = \mathrm{dom}(f_{M,k})$ and $\mathbb{W}^k \backslash X = \mathrm{dom}(f_{M',k})$. Therefore

$$\begin{aligned} \vec{w} \in X &\iff \exists v (\vec{w}, v) \in T_M \text{ and} \\ \vec{w} \notin X &\iff \exists v (\vec{w}, v) \in T_{M'}. \end{aligned}$$

Let

$$\begin{aligned} Y_0 &:= \{(\vec{w}, u)\,;\, \#u_{(0)} \text{ is even and } (\vec{w}, u_{(1)}) \in T_M\}, \\ Y_1 &:= \{(\vec{w}, u)\,;\, \#u_{(0)} \text{ is odd and } (\vec{w}, u_{(1)}) \in T_{M'}\}, \text{ and} \\ Y &:= Y_0 \cup Y_1. \end{aligned}$$

The sets Y_0 and Y_1 and therefore Y are computable sets, so use Lemma 4.18 to obtain a function h that searches for the least u such that $(\vec{w}, u) \in Y$. Since for each $\vec{w}$, we either have $\vec{w} \in X$ or $\vec{w} \notin X$, we know that such a u must exist, so h is a total function. Now output a if $\#h(\vec{w})_{(0)}$ is even and ε if $\#h(\vec{w})_{(0)}$ is odd. This computes the characteristic function of X. Q.E.D.

Corollary 4.40. The class of Σ_1 sets is not closed under complementation; more specifically, $\mathbb{W}\backslash\mathbf{K}$ is not computably enumerable.

Proof. We proved that the halting problem is computably enumerable but not computable (Theorems 4.32 & 4.33), so by Theorem 4.36 and Proposition 4.39, it's Σ_1, but not Δ_1. In particular, it is not Π_1, so $\mathbb{W}\backslash\mathbf{K}$ cannot be Σ_1. Thus Σ_1 is not closed under complementation. Q.E.D.

Corollary 4.41. Any type 0 language $L \subseteq \mathbb{W}$ is computably enumerable.

Proof. Let $G := (\Sigma, V, P, S)$ be the grammar such that $\mathcal{L}(G) = L$. We let $\Sigma' := \Omega \cup \{\rightarrow\}$. If $\sigma_i \in \Omega^*$, a string of the form

$$\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_n$$

is called a *derivation code* if $(\sigma_0, \ldots, \sigma_n)$ is a G-derivation; we say that σ_0 is the *start string* of the derivation code and σ_n is the *result string*.

Let $Y := \{(v, w)\,;\, v \text{ is a derivation code with start string } S \text{ and result string } w\}$. A register machine can check whether something is a derivation code and that it can check whether its result string coincides with w. So, the set Y is computable. But by construction,

$$w \in \mathcal{L}(G) \iff \exists v (v, w) \in Y,$$

so $\mathcal{L}(G)$ is Σ_1 and hence by Theorem 4.36, it is computably enumerable. Q.E.D.

The converse also holds: every computably enumerable language is type 0. This is not proved in this lecture course; a proof can be found as Theorem 4.4 (p. 37) in Salomaa's textbook,[18] formulated in terms of Turing machines rather than register machines.

We shall give a sketch of the construction that does not require to know precisely what a Turing machine is (cf. p. 71 in § 4.10). Turing machines are more convenient for the technical details which are suppressed here since their actions are entirely local: the head of the machine sits somewhere on the tape and it can only make modifications in the cell where it sits. The machine will start in a starting Turing configuration in the start state and it will terminate in a configuration in the *halt state* q_{H}. Suppose a set X is computably enumerable, i.e., its pseudo-characteristic function ψ_X is computable by a Turing machine, i.e., if we start from $q_{\mathrm{S}}\square w\square$ and $w \in X$, then it will halt and the final configuration will be $q_{\mathrm{H}}\square a\square$.

We construct a grammar whose strings correspond to the Turing configurations that the machine runs through in its computation; the rewrite rules correspond to the *reversed* computation steps that the Turing machine performs, i.e., $\alpha \to \beta$ if β is (part of) a Turing configuration that will be modified to α by the Turing program computing ψ_X; we have additional rewrite rules that take a starting Turing configuration and modify it so that only the input word of the starting configuration remains; the start symbol S of the grammar generates the halting Turing configuration (i.e., $q_{\mathrm{H}}\square a\square$) which is independent of w. The grammar will then produce all possible Turing configuration paths that could have resulted in this halting configuration; if (and only if) it generates a configuration in the *start state*, it will then be able to continue generation by removing the coding bits and producing the actual input word w that resulted in $\psi_X(w) = a$. Thus, a word is produced by this grammar if and only if $\psi_X(w) = a$ if and only if $w \in X$.

4.9 Closure properties

Proposition 4.42. The class of computable languages is closed under union, intersection, complement, and concatenation.

Proof. Let A and B be computable sets, i.e., χ_A and χ_B are computable functions. Then

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

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

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

are computable functions, and so $A \cap B$, $A \cup B$, and $\mathbb{W}\setminus A$ are computable sets. Also, the concatenation AB is computable: given a word w, check all initial segments of w whether they are in A, using the computable function χ_A; if one of them is, check the remainder of w by χ_B; if both checks are successful, output a; after all $|w|$ many initial segments of A have been checked unsuccessfully, output ε. Q.E.D.

[18] A. Salomaa (1973). *Formal Languages*, Academic Press.

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

Figure 6: The closure properties of all classes of languages we discussed in an overview.

Proposition 4.43. The computably enumerable languages are closed under union, intersection, and concatenation, but not under complementation and difference.

Proof. The construction for intersection from the proof of Proposition 4.42 works for pseudo-characteristic functions as well:

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

For union, write both A and B in Σ_1 form, i.e.,

$$\begin{aligned} w \in A &\iff \exists v((w,v) \in C) \text{ and} \\ w \in B &\iff \exists v((w,v) \in D) \end{aligned}$$

and use the zigzag method to get

$$w \in A \cup B \iff \text{ there is } v \text{ such that } \begin{cases} (w, v_{(1)}) \in C & \text{if } \#v_{(0)} \text{ is even and} \\ (w, v_{(1)}) \in D & \text{if } \#v_{(0)} \text{ is odd.} \end{cases}$$

The set described on the right-hand side of the equivalence is computable by Proposition 4.42.

The argument for concatenation is a modification of the concatenation argument from the proof of Proposition 4.42. As before, we let C and D be the computable sets that witness that A and B are Σ_1, respectively. Given $w, v \in \mathbb{W}$, we write $I(w,v)$ for the initial segment of w of length $\#v$ (possibly all of w, if $\#v \geq |w|$) and $F(w,v)$ for the final segment of w that remains after removing $\mathbf{|}(w,v)$ (possibly empty). We need to split a word v into three parts: let $v_{(2)} := (v_{(1)})_{(0)}$ and $v_{(3)} := (v_{(1)})_{(1)}$; then $v = v_{(0)} * (v_{(2)} * v_{(3)})$. With these definitions, we have

$$w \in AB \iff \text{there is some } v \text{ such that } (I(w, v_{(0)}), v_{(2)}) \in C \text{ and } (F(w, v_{(0)}), v_{(3)}) \in D$$

which is in Σ_1 form, so AB is computably enumerable.

That the computably enumerable sets are not closed under complementation (and thus not under differences) is Corollary 4.40. Q.E.D.

We summarise all of the closure properties discussed in this lecture course in Figure 6. The results on type 1 languages were not discussed in this lecture course. The closure of the class of context-sensitive languages under complementation was a famous open problem for

several decades which was solved independently by Immerman and Szelepcsényi in 1987.[19] Using the Immerman-Szelepcsényi theorem, closure under intersection and difference follows by general set algebra.

4.10 The Church-Turing thesis

It turns out that the converse of Theorem 4.22 can also be proved: every computable function is recursive. Two concepts that are (remarkably) different mathematical conceptualisations of what it means to be a computation are equivalent. This equivalence result is not an isolated result: many other *models of computation* have been defined that all define equivalent concepts of computability:

Not lectured in detail

Turing machines. A Turing machine consists of an infinite *tape* and a *head* that moves on the tape and can read and write letters on the tape. The tape is organised into cells ordered like the natural numbers (we think of them indexed by elements of $\mathbb{N}$): each cell can contain a letter or be empty. At the beginning, the head is positioned on cell 0 and the tape contains the word $\text{code}(\vec{w})$ for some sequence of word $\vec{w}$. The behaviour of the head is determined by its current state and the letter it reads in the cell on which the head is currently placed; based on this, it can change the symbol on the cell where it sits, change the state, and either move left or right or stay where it is.

Formally, a *Turing machine* $M = (\Sigma, Q, P)$ consists of an alphabet Σ, augmented to $\Sigma' := \Sigma \cup \{\square\}$, a finite set of states Q, disjoint from Σ' with both a *start state* q_{S} and a *halt state* q_{H}. We write $\Omega := \Sigma' \cup Q$ and $\text{Instr} := \{\mathbf{L}, \mathbf{R}, \mathbf{o}\} \times \Sigma' \times Q$ is the set of *Turing instructions*. We interpret $(\mathbf{L}, a, q)$ as "write a, go to state q, move head left", $(\mathbf{R}, a, q)$ as "write a, go to state q, move head right", and $(\mathbf{o}, a, q)$ as "write a, go to state q, don't move head". The *Turing program* P is a function from $Q \times \Sigma'$ to Instr.

A *Turing configuration* is a string $C \in \Omega^*$ that has precisely one occurrence of a symbol in Q. This symbol indicates the position of the head on the tape, the other symbols are the tape content. A Turing program P transforms a Turing configuration C with state q to another Turing configuration C' by the rewrite rules

$$\begin{aligned} aqb &\to q'ac \text{ if } P(q,b) = (\mathbf{L}, c, q'), \\ aqb &\to acq' \text{ if } P(q,b) = (\mathbf{R}, c, q'), \text{ and} \\ aqb &\to aq'c \text{ if } P(q,b) = (\mathbf{o}, c, q'). \end{aligned}$$

The Turing configuration $C_{\mathrm{S}}(\vec{w}) := q_{\mathrm{S}}\square w_0 \square w_1 \square \ldots \square w_{k-1}\square$ is called the *start configuration for input* $\vec{w}$ and the *Turing configuration with input* $\vec{w}$ is defined by recursion via

$$\begin{aligned} C(0, M, \vec{w}) &:= C_{\mathrm{S}}(\vec{w}) \text{ and} \\ C(k+1, M, \vec{w}) &:= C' \text{ if } M \text{ transforms } C(k, M, \vec{w}) \text{ to } C'. \end{aligned}$$

This computation halts if one of the configurations is in the state q_{H}. The output of this computation in this case is the word that lies between the first and second instance of the

[19] N. Immerman (1988), Nondeterministic space is closed under complementation. SIAM Journal on Computing, 17 (5): 935–938. R. Szelepcsényi, (1987), The method of forcing for nondeterministic automata, Bulletin of the EATCS, 33: 96–100.

letter $\square$ on the tape at the time of halting (compare the proof of Theorem 4.29). If P is a Turing program, we write $f_{P,k}(\vec{w})$ for the partial function that is defined if P halts on input $\vec{w}$ and produces the output of the computation. A partial function $f : \mathbb{W}^k \dashrightarrow \mathbb{W}$ is called *Turing computable* if and only if there is a Turing machine M such that $f = f_{M,k}$.

Not lectured in detail

While programs. A *while program* is defined by recursion using finite number of natural number tokens $\mathtt{1}, ..., \mathtt{n}$. Let $\mathtt{i}$ be one of these natural number tokens and let $a \in \Sigma$. The instructions $\mathtt{add(i,}a\mathtt{)}$ and $\mathtt{remove(i)}$ are while programs. If P and Q are while programs, then PQ is a while program. If P is a while program then so is $\mathtt{while\ i\ not\ empty\ do}\ P$. We interpret these as "add the letter a to the ith word", "remove the last letter from the ith word (if it is empty, do nothing)", and "repeat the while program P until the ith word is empty".

A *while configuration* for a while program P consists of an n-tuple of words and a marker that indicates where in the while program we currently are. The initial while configuration starts at the beginning of the while program. We can then define by recursion a *while computation sequence* consisting of the sequence of configurations that the while program generates. If the marker ever reaches the end of the program, the computation terminates. (So, while computation sequences, in contrast to computation sequences or Turing computation sequences can be finite.) If P is a while programme, we write $f_{P,k}(\vec{w})$ for the partial function that is defined if the while computation sequence is finite and outputs the 0th word at the end of the computation sequence. A partial function $f : \mathbb{W}^k \dashrightarrow \mathbb{W}$ is called *while computable* if and only if there is a while program P such that $f = f_{P,k}$.

Theorem 4.44. If $f : \mathbb{W}^k \dashrightarrow \mathbb{W}$, then the following are equivalent:

(i) the partial function f is computable,

(ii) the partial function f is recursive,

(iii) the partial function f is Turing computable, and

(iv) the partial function f is while computable.

The confluence of so many different attempts to formalise the notion of computability suggests that the concept that we described is robust and reflects something substantial about the pre-theoretical concept of *computation*. In fact, when defining his Turing machines, Turing had the intention to capture the essence of the nature of computation and describe it formally.

> **The Church-Turing Thesis.** The mentioned equivalent formal concepts of computability describe the informal notion of computability successfully: any reasonable attempt to describe the informal notion of computability will lead to a formal notion that is equivalent to the ones we have described.

It is very important to note that the Church-Turing thesis is not a mathematical statement: it cannot be proved or refuted, but it makes a prediction about the human practice of mathematics. It could be refuted in practice if mathematicians find a formal description of a

model of computation that yields a non-equivalent notion of computability and unanimously agree that this formal description describes the informal notion of computability. There have been candidates for this in the decades that followed the Church-Turing discovery: e.g., quantum computing, DNA computing, and other models of so-called *unconventional computing*. While they often produced models where computation behaves rather differently from computation by register machines in various respects, their notions of computability remain equivalent to our notion of computability.

The Church-Turing Thesis finally provides us with an answer to the question raised earlier about the definition of the word "algorithm". Informally, by "algorithm", we meant a computational procedure that produces an answer to the decision problem in a finite amount of time. If we accept the Church-Turing Thesis, this informal notion of a computational procedure is correctly formalised by the concept of a register machine, i.e., it corresponds to the notion of computability: each of our decision problems just becomes a set and the question whether it is solvable becomes the question whether that set is computable.

Not Lectured

More precisely, we consider an encoding of grammars as words in $\mathbb{W}$ such that for every word w, there is a grammar G_w and all grammars are of this form. Then the *word problem* is the set $\{(u,v)\,;\, u \in \mathcal{L}(G_v)\}$, the *emptiness problem* is the set $\{u\,;\, \mathcal{L}(G_u) = \varnothing\}$, and the *equivalence problem* is the set $\{(u,v)\,;\, \mathcal{L}(G_u) = \mathcal{L}(G_v)\}$. Similarly, the decision problems restricted to a class $\mathcal{C}$ of grammars are these sets restricted to words that decode into grammars in $\mathcal{C}$.

Of course, there is no reason to restrict decision problems to grammars only: the same definitions also give us the word, emptiness, and equivalence problem for register machines or any other encodable model of computation.

The interpretation of the word "algorithm" via the Church-Turing thesis finally gives us the mathematical specificity needed to prove the unsolvability of a decision problem: in order to do so, we have to show that the corresponding set is not computable. On Example Sheet # 3, we saw that the solution algorithm for the word problem for type 1 grammars can be performed by a register machine; consequently $\{(u,v)\,;\, u \in \mathcal{L}(\mathrm{d}_v)$ and v is a code for a type 1 grammar$\}$ is computable. We furthermore note that all algorithms given in §§ 1.6, 2.8, & 3.5) can be performed by register machines, so all of our solvability results from previous chapters give computability results for the corresponding sets.

A decision problem A subset of W^k is solvable iff it's computable

Corollary 4.45. The word problem for type 0 languages is unsolvable.

Proof. As mentioned above, the word problem is the set $W := \{(u,v)\,;\, u \in \mathcal{L}(G_v)\}$. We replicate the proof of Theorem 4.33: if W is computable, then so is the function

$$f(w) := \begin{cases} \uparrow & \text{if } w \in \mathcal{L}(G_w) \text{ and} \\ \varepsilon & \text{if } w \notin \mathcal{L}(G_w). \end{cases}$$

Let $\mathrm{dom}(f)$ be computably enumerable, so (by the equivalence of type 0 languages and computably enumerable sets; cf. p. 69) there is a grammar G_d such that $\mathrm{dom}(f) = \mathcal{L}(G_d)$. We obtain the contradiction by

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

Q.E.D.

Note that the set W in the proof of Corollary 4.45 is essentially the halting problem $\mathbf{K}_0$ if you assume that a register machine can perform the transformation between computably enumerable sets and grammars (cf. 4.11).

4.11 Reduction functions

A binary relation $\leq$ on a set X is called a *partial preorder* if it is reflexive and transitive (i.e., for all $x, y, z \in X$, we have $x \leq x$ and if $x \leq y \leq z$, then $x \leq z$). If $\leq$ is a partial preorder, we can define a binary relation $\equiv$ by $x \equiv y$ if and only if $x \leq y$ and $y \leq x$. This is an equivalence relation and $\leq$ respects the equivalence classes, i.e., if $x \equiv x'$ and $x \leq y$, then $x' \leq y$, similarly, if $x \equiv x'$ and $y \leq x$, then $y \leq x'$. If $[x]$ and $[y]$ are $\equiv$-equivalence classes, we can define $[x] \leq [y]$ if and only if $x \leq y$; this is well defined since $\leq$ respects equivalence classes. If $(X, \leq)$ is a partially preordered set, then $(X/\equiv, \leq)$ is a partially ordered set (i.e., partially preordered and anti-symmetric).

If $L, L' \subseteq \mathbb{W}$, we call a total computable function $f : \mathbb{W} \to \mathbb{W}$ a *reduction of L to L'* if for all $w \in \mathbb{W}$, we have

$$w \in L \iff f(w) \in L'.$$

We say that L is *many-one reducible* to L' and write $L \leq_{\mathrm{m}} L'$ if there is a reduction from L to L'.[20] If $L \leq_{\mathrm{m}} L'$ and $L' \leq_{\mathrm{m}} L$, we say that L and L' are *many-one equivalent* and write $L \equiv_{\mathrm{m}} L'$. We observe that the identity is a reduction of L to L and that a concatenation of a reduction of L to L' and a reduction of L' to L'' produces a reduction of L to L''; thus $\leq_{\mathrm{m}}$ is a partial preorder. Note that by definition of what it means to be a reduction function,

$$L \leq_{\mathrm{m}} L' \iff \mathbb{W}\backslash L \leq_{\mathrm{m}} \mathbb{W}\backslash L'. \qquad (\#)$$

As a consequence, a language is many-one reducible to its complement if and only if it is many-one equivalent to its complement. In other words, a language and its complement are either many-one equivalent or incomparable by $\leq_{\mathrm{m}}$.

Proposition 4.46. Let $L, L' \subseteq \mathbb{W}$.

(a) If $L \leq_{\mathrm{m}} L'$ and L' is computable, then so is L.

(b) If $L \leq_{\mathrm{m}} L'$ and L' is computably enumerable, then so is L.

Proof. For (a), let f be the reduction and let $\chi_{L'}$ be computable. Then $\chi_L = \chi_{L'} \circ f$ is computable. For (b), use ψ_L and $\psi_{L'}$ instead of the characteristic functions. Q.E.D.

If there is a reduction of L to L', we can think of L as "at most as complicated as L'": having access to the characteristic function of L' gives us access to the characteristic function of L.

[20] The term "many-one" is a reminder that the function is not required to be injective ("one-one"), but can map several words to one word.

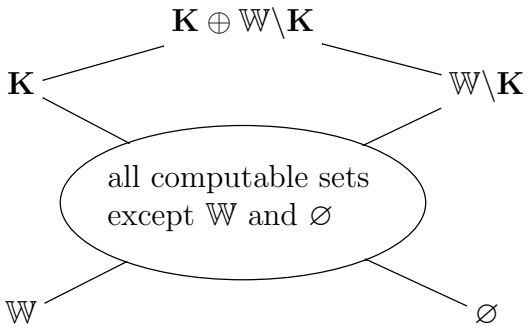


Figure 7: The many-one degrees.

The notion of reduction was implicitly used in some of the discussions about algorithmic solvability. In § 4.10, we claimed that the set $\{(w, v)\,;\, w \in \mathcal{L}(G_v)\}$ representing the word problem for type 0 grammars is "essentially the halting problem $\mathbf{K}_0$ if you assume that a register machine can perform the transformation between computably enumerable sets and grammars". If we let f and g be the total computable functions that translate between codes of grammars and codes of register machines and vice versa, then this just means that f and g witness that

$$\{(w, v)\,;\, w \in \mathcal{L}(G_v)\} \equiv_{\mathrm{m}} \{(w, v)\,;\, w \in \mathrm{W}_v\} = \mathbf{K}_0.$$

Similarly, we can identify the emptiness problem and the equivalence probvlem for type 0 grammars with $\{w\,;\, \mathrm{W}_w = \varnothing\}$ and $\{(w, v)\,;\, \mathrm{W}_w = \mathrm{W}_v\}$, respectively.

Proposition 4.47. The sets $\mathbf{K}$ and $\mathbb{W}\backslash\mathbf{K}$ are incomparable in $\leq_{\mathrm{m}}$.

Proof. Since $\mathbb{W}\backslash\mathbf{K}$ is not computably enumerable, we have $\mathbb{W}\backslash\mathbf{K} \not\leq_{\mathrm{m}} \mathbf{K}$ by Proposition 4.46 (b). But if $\mathbf{K} \leq_{\mathrm{m}} \mathbb{W}\backslash\mathbf{K}$, then $\mathbb{W}\backslash\mathbf{K} \leq_{\mathrm{m}} \mathbf{K}$ by (#). Q.E.D.

If $|\Sigma| \geq 2$ and $a \in \Sigma$, we can define for any two sets $X, Y \subseteq \mathbb{W}$ the *Turing join of X and Y*:

$$X \oplus Y := aX \cup \bigcup_{b \neq a} bY.$$

Clearly, the function $w \mapsto aw$ is a reduction from X to $X \oplus Y$ and the function $w \mapsto bw$ (for any $b \neq a$) is a reduction from Y to $X \oplus Y$, so $X, Y \leq_{\mathrm{m}} X \oplus Y$. In particular, the Turing join produces something that is at least as complicated as the two original sets and thus $\mathbf{K} \oplus \mathbb{W}\backslash\mathbf{K}$ is a set that is strictly more complex than both $\mathbf{K}$ and $\mathbb{W}\backslash\mathbf{K}$, in particular, it cannot be either Σ_1 or Π_1. On Example Sheet # 4, we shall see that the Turing join corresponds to the least upper bound operation in the preorder $\leq_{\mathrm{m}}$. The results about the notion of many-one reducibility are collected in Figure 7 (cf. also Example Sheet # 4).

Hardness & completeness. If $\mathcal{C}$ is a class of languages and L is a language, then L is called *$\mathcal{C}$-hard* if for all $X \in \mathcal{C}$, we have $X \leq_{\mathrm{m}} L$. This means that L is an upper bound for the class $\mathcal{C}$ in terms of computational complexity. If L is $\mathcal{C}$-hard and in addition $L \in \mathcal{C}$, then we call it *$\mathcal{C}$-complete*.

Proposition 4.48. If L is any computable language such that $\varnothing \neq L \neq \mathbb{W}$, then L is Δ_1-complete.

Proof. By Propositions 4.35 & 4.39, computable and Δ_1 are the same, so we only need to show that if X is an arbitrary computable set, then $X \leq_{\mathrm{m}} L$. The assumption implies that there are $v, u \in \mathbb{W}$ such that $v \in L$ and $u \notin L$. Let

$$g(w) := \begin{cases} v & \text{if } w \in X \text{ and} \\ u & \text{if } w \notin X. \end{cases}$$

Since X is computable, g is computable and it is a reduction of X to L. Q.E.D.

Theorem 4.49. The halting problem $\mathbf{K}$ is Σ_1-complete.

Proof. Let $X = \mathrm{dom}(f)$ be computably enumerable. Define $g : \mathbb{W}^2 \dashrightarrow \mathbb{W} : (w, u) \mapsto f(w)$ and apply the *s-m-n* Theorem 4.30 to g to get a total computable function h such that $f_{h(w),1}(u) = g(w, u) = f(w)$. In particular, we observe that $w \in X = \mathrm{dom}(f)$ if and only $f_{h(w),1}$ is everywhere defined and $w \notin X$ if and only if $f_{h(w),1}$ is nowhere defined. But if $f_{h(w),1}$ is everywhere defined, then in particular, $f_{h(w),1}(h(w))\downarrow$, so $h(w) \in \mathbf{K}$ and if $f_{h(w),1}$ is nowhere defined, then in particular, $f_{h(w),1}(h(w))\uparrow$, so $h(w) \notin \mathbf{K}$. Together, we obtain

$$w \in X \iff h(w) \in \mathbf{K}$$

which shows that h is a reduction of X to $\mathbf{K}$. Q.E.D.

I is an index set if $w \in I$ and $W_w = W_v \Rightarrow v \in I$.

4.12 Index sets & Rice's theorem

We remember our notion of weak equivalence (now transferred to words rather than machines): two words $w, v \in \mathbb{W}$ to be *weakly equivalent* if $\mathrm{W}_w = \mathrm{W}_v$. A set $I \subseteq \mathbb{W}$ is called an *index set* if it is closed under weak equivalence. We say that an index set is *nontrivial*, if it is neither $\varnothing$ nor $\mathbb{W}$. Index sets correspond to properties of computably enumerable sets. Henry Gordon Rice (1920–2003) proved that nontrivial index sets cannot be computable.[21]

Example 4.50. The sets $\mathbf{Emp} := \{w\,;\, \mathrm{W}_w = \varnothing\}$, $\mathbf{Fin} := \{w\,;\, \mathrm{W}_w \text{ is finite}\}$, $\mathbf{Inf} := \{w\,;\, \mathrm{W}_w \text{ is infinite}\}$, and $\mathbf{Tot} := \{w\,;\, \mathrm{W}_w = \mathbb{W}\}$ are nontrivial index sets. Non-empty index sets must be infinite (by the Padding Lemma, Proposition 4.4).

Note that the Recursion Theorem (Theorem 4.31) can be used to show that $\mathbf{K}$ is not an index set: details are on Example Sheet #4.

Theorem 4.51 (Rice's Theorem)**.** No nontrivial index set is computable.

[21] H. G. Rice (1953), Classes of recursively enumerable sets and their decision problems, *Transactions of the American Mathematical Society* 74 (2): 358–366.

Proof. For a fixed w, consider the following function:

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

We first observe that g_w is computable: given u and v, we first run the computation $f_{u,1}(u)$. If that diverges, then the computation outputs $\uparrow$ which is the desired result. If it converges, we run the computation of $f_{w,1}$ on input v and output the result (if there is one). Therefore, by the *s-m-n* theorem, we obtain a total computable h_w such that $f_{h_w(u),1}(v) = g_w(u,v)$. If $u \in \mathbf{K}$, then $f_{h_w(u),1}$ is defined whenever $f_{w,1}$ is, so $\mathrm{W}_{h_w(u)} = \mathrm{W}_w$. If $u \notin \mathbf{K}$, then $f_{f_w(u),1}$ is nowhere defined, so $\mathrm{W}_{h_w(u)} = \varnothing$.

Now let I be our index set. Fix some e such that $\mathrm{W}_e = \varnothing$. Then either $e \in I$ or $e \notin I$.

Case 1. If $e \in I$, then by nontriviality, there must be some $w \notin I$. Consider g_w as above and the total function h_w obtained by the *s-m-n* theorem. We claim that h is a reduction of $\mathbb{W}\backslash\mathbf{K}$ to I. If $u \notin \mathbf{K}$, then $\mathrm{W}_{h_w(u)} = \mathrm{W}_e = \varnothing$, so since I is an index set, $h_w(u) \in I$. Conversely, if $u \in \mathbf{K}$, then $\mathrm{W}_{h_w(u)} = \mathrm{W}_w$, so since I is an index set, $h_w(u) \notin I$. So, $\mathbb{W}\backslash\mathbf{K} \leq_{\mathrm{m}} I$.

Case 2. If $e \notin I$, then by nontriviality, there must be some $w \in I$. The above construction yields (just with the roles of e and w reversed) that $u \in \mathbf{K}$ if and only if $h_w(u) \in I$. So, $\mathbf{K} \leq_{\mathrm{m}} I$. Q.E.D.

We note that the proof shows more than the statement of Rice's Theorem: the proof shows that if $e \in I$, then $\mathbb{W}\backslash\mathbf{K} \leq_{\mathrm{m}} I$, and if $e \notin I$, then $\mathbf{K} \leq_{\mathrm{m}} I$, so for our examples of nontrivial index sets, we obtain $\mathbb{W}\backslash\mathbf{K} \leq_{\mathrm{m}} \mathbf{Emp}, \mathbf{Fin}$ and $\mathbf{K} \leq_{\mathrm{m}} \mathbf{Inf}, \mathbf{Tot}$.

Corollary 4.52. The emptiness problem for Type 0 languages is not solvable.

Proof. The emptiness problem is represented by the set **Emp**: since $\mathbb{W}\backslash\mathbf{K} \leq_{\mathrm{m}} \mathbf{Emp}$, this set is not computable (nor computably enumerable). Q.E.D.

On Example Sheet # 4, we shall see that $\mathbf{Emp} \equiv_{\mathrm{m}} \mathbb{W}\backslash\mathbf{K}$; the other sets in our list are even more complex as the following statement shows:

Proposition 4.53. The set **Fin** is neither Σ_1 nor Π_1.

Proof. We already know that $\mathbb{W}\backslash\mathbf{K} \leq_{\mathrm{m}} \mathbf{Fin}$, so **Fin** is not Σ_1. To prove the claim, we shall show that $\mathbf{K} \leq_{\mathrm{m}} \mathbf{Fin}$.

We use the computable truncated computation set T_w from (the remark after) Corollary 4.28: if $(u,v) \in T_w$, then $f_{w,1}(u)$ has halted within $\#v$ steps. Note that if $(u,v) \in T_w$, then for any $v' > v$, we have $(u,v') \in T_w$. Consider the computable function

$$g(w,v) := \begin{cases} \uparrow & \text{if } (w,v) \in T_w \text{ and} \\ \varepsilon & \text{otherwise.} \end{cases}$$

By the *s-m-n* Theorem 4.30, we find a total computable h such that $f_{h(w),1}(v) = g(w,v)$. We claim that h reduces $\mathbf{K}$ to **Fin**.

	Word problem	Emptiness problem	Equivalence problem
regular (type 3)	✓	✓	✓
context-free (type 2)	✓	✓	×
context-sensitive (type 1)	✓	×	×
computably enumerable (type 0)	×	×	×

Figure 8: The decision problems of all classes of languages we discussed in an overview.

Suppose that $w \in \mathbf{K}$. Then $f_{w,1}(w)\downarrow$, so there is some v such that $(w, v) \in T_w$ which remains true for all $v' > v$. Therefore, $f_{h(w),1}$ is undefined for all but finitely many v, and thus $\mathrm{W}_{h(w)}$ is finite, so $h(w) \in \mathbf{Fin}$.

Suppose that $w \notin \mathbf{K}$. Then $f_{w,1}(w)\uparrow$, so for all v, we have that $(w, v) \notin T_w$, and thus $f_{h(w),1}(v) = \varepsilon$. So, $\mathrm{W}_{h(w)} = \mathbb{W}$ and $h(w) \notin \mathbf{Fin}$. Q.E.D.

Note that this implies that **Inf** cannot be Σ_1 or Π_1 either since it is the complement of **Fin**; the set **Tot** will be discussed on Example Sheet # 4.

4.13 Decision problems

We have discussed the word problem and the emptiness problem in §§ 4.8 & 4.12, respectively. The only remaining decision problem for type 0 grammars is the equivalence problem, i.e., the set $\{(w, v)\,;\, \mathrm{W}_w = \mathrm{W}_v\}$. Its unsolvability can be derived immediate from that of the emptiness problem.

Corollary 4.54. The equivalence problem for Type 0 languages is not solvable.

Proof. If e is such that $\mathrm{W}_e = \varnothing$, then the operation $w \mapsto (w, e)$ can be performed by a register machine. If χ is the characteristic function of $\{(w, v)\,;\, \mathrm{W}_w = \mathrm{W}_v\}$, then let $\chi'(w) := \chi(w, e)$. If χ is computable, then so is χ'. But χ' is the characteristic function of the emptiness problem $\{w\,;\, \mathrm{W}_w = \varnothing\}$ in contradiction to Corollary 4.52 Q.E.D.

We observe that the proof of Corollary 4.54 is a fully general argument that shows that if $\mathcal{C}$ is any class of grammars such that there is a $G \in \mathcal{C}$ with $\mathcal{L}(G) = \varnothing$, then the solvability of the equivalence problem for $\mathcal{C}$ implies the solvability of the emptiness problem for $\mathcal{C}$.

We summarise the results concerning our decision problems in Figure 8; note that we did not prove the unsolvability of the equivalence problem for type 2 languages (cf. § 3.5) and the unsolvability of the emptiness problem for type 1 languages. The latter can be found as Theorem 5.10 (p. 223) in Sipser's textbook,[22] albeit expressed in the language if *linear bounded automata* which is the model of computation that corresponds to type 1 grammars. By the above observation, the unsolvability of the emptiness problem for type 1 grammars implies the unsolvability of the equivalence problem for type 1 grammars.

[22] M. Sipser. Introduction to the theory of computing. Second edition. Thomson Course Technology, 2006