FORMAL LANGUAGES

Keijo Ruohonen

2009

Contents

Foreword

These lecture notes were translated from the Finnish lecture notes for the TUT course "Formaalit kielet". The notes form the base text for the course "MAT-41186 Formal Languages". They contain an introduction to the basic concepts and constructs, as seen from the point of view of languages and grammars. In a sister course "MAT-41176 Theory of Automata" much similar material is dealt with from the point of view of automata, computational complexity and computability.

Formal languages have their origin in the symbolical notation formalisms of mathematics, and especially in combinatorics and symbolic logic. These were later joined by various codes needed in data encryption, transmission, and error-correction—all these have significantly influenced also the theoretical side of things—and in particular various mathematical models of automation and computation.

It was however only after Noam Chomsky's ground-breaking ideas in the investigation of natural languages, and the algebro-combinatorial approach of Marcel-Paul Schützenberger's in the 1950's that formal language theory really got a push forward. The strong influence of programming languages should be noted, too. During the "heydays" of formal languages, in the 1960's and 1970's, much of the foundation was created for the theory as it is now.[1] Nowadays it could be said that the basis of formal language theory has settled into a fairly standard form, which is seen when old and more recent text-books in the area are compared. The theory is by no means stagnant, however, and research in the field continues to be quite lively and popular.

In these lecture notes the classical Chomskian formal language theory is fairly fully dealt with, omitting however much of automata constructs and computability issues. In

[1] Among the top investigators in the area especially the Finnish academician Arto Salomaa might be mentioned.

addition, surveys of Lindenmayer system theory and the mathematical theory of codes are given. As a somewhat uncommon topic, an overview of formal power series is included. Apart from being a nice algebraic alternative formalism, they give a mechanism for generalizing the concept of language in numerous ways, by changing the underlying concept of set but not the concept of word.[2]

Keijo Ruohonen

[2] There are various ways of generalizing languages by changing the concept of word, say, to a graph, or a picture, or a multidimensional word, or an infinite word, but these are not dealt with here.

”The words. Why did they have to exist?
Without them, there wouldn’t be any of this.”

(MARKUS ZUSAK: *The Book Thief*)

Chapter 1

WORDS AND LANGUAGES

1.1 Words and Alphabets

A *word* (or *string*) is a finite sequence of items, so-called *symbols* or *letters* chosen from a specified finite set called the *alphabet.* Examples of common alphabets are e.g. letters in the Finnish alphabet (+ interword space, punctuation marks, etc.), and the bits 0 and 1. A word of length one is identified with its only symbol. A special word is the *empty word* (or *null word*) having no symbols, denoted by Λ (or λ or ε or 1).

The *length* of the word w is the number of symbols in it, denoted by $|w|$. The length of the empty word is 0. If there are k symbols in the alphabet, then there are k^n words of length n. Thus there are

$$\sum_{i=0}^{n} k^i = \frac{k^{n+1}-1}{k-1}$$

words of length at most n, if $k > 1$, and $n+1$ words, if $k = 1$. The set of all words is denumerably infinite, that is, they can be given as an infinite list, say, by ordering the words first according to their length.

The basic operation of words is *concatenation*, that is, writing words as a compound. The concatenation of the words w_1 and w_2 is denoted simply by w_1w_2. Examples of concatenations in the alphabet $\{a, b, c\}$:

$$\begin{array}{lllll} w_1 = aacbba & , & w_2 = caac & , & w_1w_2 = aacbbacaac \\ w_1 = aacbba & , & w_2 = \Lambda & , & w_1w_2 = w_1 = aacbba \\ w_1 = \Lambda & , & w_2 = caac & , & w_1w_2 = w_2 = caac \end{array}$$

Concatenation is associative, i.e.,

$$w_1(w_2w_3) = (w_1w_2)w_3.$$

As a consequence of this, repeated concatenations can be written without parentheses. On the other hand, concatenation is usually not commutative, As a rule then

$$w_1w_2 \neq w_2w_1,$$

but not always, and in the case of a unary alphabet concatenation is obviously commutative.

The n^{th} (*concatenation*) *power* of the word w is

$$w^n = \underbrace{ww\cdots w}_{n \text{ copies}}.$$

Especially $w^1 = w$ and $w^0 = \Lambda$, and always $\Lambda^n = \Lambda$.

The *mirror image* (or *reversal*) of the word $w = a_1a_2 \cdots a_n$ is the word

$$\hat{w} = a_n \cdots a_2a_1,$$

especially $\hat{\Lambda} = \Lambda$. Clearly we have $\widehat{w_1w_2} = \hat{w}_2\hat{w}_1$. A word u is a *prefix* (resp. *suffix*) of the word w, if $w = uv$ (resp. $w = vu$) for some word v. A word u is a *subword* (or *segment*) of the word w, if $w = v_1uv_2$ for some words v_1 and v_2. A word u is a *scattered subword* of the word w, if

$$w = w_1u_1w_2u_2 \cdots w_nu_nw_{n+1}$$

where $u = u_1u_2 \cdots u_n$, for some n and some words $w_1, w_2, \dots, w_{n+1}$ and $u_1, u_2, \dots, u_n$.

1.2 Languages

A *language* is a set words over some alphabet. Special examples of languages are *finite languages* having only a finite number of words, *cofinite languages* missing only a finite number of words, and the *empty language* $\emptyset$ having no words. Often a singleton language $\{w\}$ is identified with its only word w, and the language is denoted simply by w.

The customary set-theoretic notation is used for languages: $\subseteq$ (inclusion), $\subset$ (proper inclusion), $\cup$ (union), $\cap$ (intersection), $-$ (difference) and $\overline{}$ (complement against the set of all words over the alphabet). Belonging of a word w in the language L is denoted by $w \in L$, as usual. Note also the "negated" relations $\not\subseteq$, $\not\subset$ and $\notin$.

The language of all words over the alphabet Σ, in particular Λ, is denoted by Σ^*. The language of all nonempty words over the alphabet Σ is denoted by Σ^+. Thus $\overline{L} = \Sigma^* - L$ and $\Sigma^+ = \Sigma^* - \{\Lambda\}$.

Theorem 1. *There is a nondenumerably infinite amount of languages over any alphabet, thus the languages cannot be given in an infinite list.*

Proof. Let us assume the contrary: All languages (over some alphabet Σ) appear in the list $L_1, L_2, \dots$ We then define the language L as follows: Let $w_1, w_2, \dots$ be a list containg all words over the alphabet Σ. The word w_i is in the language L if and only if it is not in the language L_i. Clearly the language L is not then any of the languages in the list $L_1, L_2, \dots$ The counter hypothesis is thus false, and the theorem holds true. □

The above method of proof is an instance of the so-called *diagonal method.* There can be only a denumerably infinite amount of ways of defining languages, since all such definitions must be expressible in some natural language, and thus listable in lexicographic order. In formal language theory defining languages and investigating languages via their definitions is paramount. Thus only a (minuscule) portion of all possible languages enters the investigation!

There are many other operations of languages in addition to the set-theoretic ones above. The *concatenation* of the languages L_1 and L_2 is

$$L_1L_2 = \{w_1w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}.$$

The n^{th} (*concatenation*) *power* of the language L is

$$L^n = \{w_1w_2 \cdots w_n \mid w_1, w_2, \dots, w_n \in L\},$$

and especially $L^1 = L$ and $L^0 = \{\Lambda\}$. In particular $\emptyset^0 = \{\Lambda\}$! The *concatenation closure* or *Kleenean star* of the language L is

$$L^* = \bigcup_{n=0}^{\infty} L^n,$$

i.e., the set obtained by concatenating words of L in all possible ways, including the "empty" concatenation giving Λ. Similarly

$$L^+ = \bigcup_{n=1}^{\infty} L^n,$$

which contains the empty word Λ only if it is already in L. (Cf. Σ^* and Σ^+ above.) Thus $\emptyset^* = \{\Lambda\}$, but $\emptyset^+ = \emptyset$. Note that in fact $L^+ = L^*L = LL^*$.

The *left and right quotients* of the languages L_1 and L_2 are defined as

$$L_1 \backslash L_2 = \{w_2 \mid w_1 w_2 \in L_2 \text{ for some word } w_1 \in L_1\}$$

(remove from words of L_2 prefixes belonging in L_1 in all possible ways) and

$$L_1 / L_2 = \{w_1 \mid w_1 w_2 \in L_1 \text{ for some word } w_2 \in L_2\}$$

(remove from words of L_1 suffixes belonging in L_2 in all possible ways). Note that the prefix or the suffix can be empty. The *mirror image* (or *reversal*) of the language L is the language $\hat{L} = \{\hat{w} \mid w \in L\}$.

There are two fundamental machineries of defining languages: *grammars,* which generate words of the language, and *automata,* which recognize words of the language. There are many other ways of defining languages, e.g. defining regular languages using regular expressions.

> ”Some people, when confronted with
> a problem, think ”I know, I'll use
> regular expressions.” Now they
> have two problems.”
>
> (Jamie Zawinski)

Chapter 2

REGULAR LANGUAGES

2.1 Regular Expressions and Languages

A *regular expression* is a formula which defines a language using set-theoretical union, denoted here by $+$, concatenation and concatenation closure. These operations are combined according to a set of rules, adding parentheses (and) when necessary. The atoms of the formula are symbols of the alphabet, the empty language $\emptyset$, and the empty word Λ, the braces $\{$ and $\}$ indicating sets are omitted.

Languages defined by regular expressions are the so-called *regular languages.* Let us denote the family of regular languages over the alphabet Σ by $\mathcal{R}_\Sigma$, or simply by $\mathcal{R}$ if the alphabet is clear by the context.

Definition. *$\mathcal{R}$ is the family of languages satisfying the following conditions:*

1. *The language $\emptyset$ is in $\mathcal{R}$ and the corresponding regular expression is $\emptyset$.*
2. *The language $\{\Lambda\}$ is in $\mathcal{R}$ and the corresponding regular expression is Λ.*
3. *For each symbol a, the language $\{a\}$ is in $\mathcal{R}$ and the corresponding regular expression is a.*
4. *If L_1 and L_2 are languages in $\mathcal{R}$, and r_1 and r_2 are the corresponding regular expressions, then*
   (a) *the language $L_1 \cup L_2$ is in $\mathcal{R}$ and the corresponding regular expression is (r_1+r_2).*
   (b) *the language L_1L_2 is in $\mathcal{R}$ and the corresponding regular expression is (r_1r_2).*
5. *If L is a language in $\mathcal{R}$ and r is the corresponding regular expression, then L^* is in $\mathcal{R}$ and the corresponding regular expression is (r^*).*
6. *Only languages obtainable by using the above rules* 1.–5. *are in $\mathcal{R}$.*

In order to avoid overly long expressions, certain customary abbreviations are used, e.g.

$$(rr) =_{\text{denote}} (r^2) \quad , \quad (r(rr)) =_{\text{denote}} (r^3) \quad \text{and}$$

$$(r(r^*)) =_{\text{denote}} (r^+).$$

On the other hand, the rules produce fully parenthesized regular expressions. If the order of precedence

$$* \quad , \quad \text{concatenation} \quad , \quad +$$

is agreed on, then a lot of parentheses can be omitted, and for example $a+b^*c$ can be used instead of the "full" expression $(a+((b^*)c))$. It is also often customary to identify a regular expression with the language it defines, e.g. $r_1 = r_2$ then means that the corresponding regular languages are the same, even though the expressions themselves can be quite different. Thus for instance

$$(a^*b^*)^* = (a+b)^*.$$

It follows immediately from the definition that the union and concatenation of two regular languages are regular, and also that the concatenation closure of a regular language is also regular.

2.2 Finite Automata

Automata are used to *recognize* words of a language. An automaton then "processes" a word and, after finishing the processing, "decides" whether or not the word is the language. An automaton is *finite* if it has a finite memory, i.e., the automaton may be thought to be in one of its (finitely many) (memory)states. A finite deterministic automaton is defined formally by giving its states, input symbols (the alphabet), the initial state, rules for the state transition, and the criteria for accepting the input word.

Definition. A finite (deterministic) automaton (DFA) *is a quintuple* $M = (Q, \Sigma, q_0, \delta, A)$ *where*

- $Q = \{q_0, q_1, \ldots, q_m\}$ *is a finite set of states, the elements of which are called* states;
- Σ *is the set* input symbols *(the alphabet of the language);*
- q_0 *is the* initial state ($q_0 \in Q$);
- δ *is the* (state) transition function *which maps each pair* (q_i, a)*, where* q_i *is a state and* a *is an input symbol, to exactly one next state* q_j: $\delta(q_i, a) = q_j$;
- A *is the so-called* set of terminal states ($A \subseteq Q$).

As its input the automaton M receives a word

$$w = a_1 \cdots a_n$$

which it starts to read from the left. In the beginning M is in its initial state q_0 reading the first symbol a_1 of w. The next state q_j is then determined by the transition function:

$$q_j = \delta(q_0, a_1).$$

In general, if M is in state q_j reading the symbol a_i, its next state is $\delta(q_j, a_i)$ and it moves on to read the next input symbol a_{i+1}, if any. If the final state of M after the last input symbol a_n is read is one of the terminal states (a state in A), then M *accepts* w, otherwise it *rejects* w. In particular, M accepts the empty input Λ if the initial state q_0 is also a terminal state.

The language *recognized* by an automaton M is the set of the words accepted by the automaton, denoted by $L(M)$.

Any word $w = a_1 \cdots a_n$, be it an input or not, determines a so-called *state transition chain* of the automaton M from a state q_{j_0} to a state q_{j_n}:

$$q_{j_0}, q_{j_1}, \ldots, q_{j_n},$$

where always $q_{j_{i+1}} = \delta(q_{j_i}, a_{i+1})$. In a similar fashion the transition function can be extended to a function δ^* for words recursively as follows:

1. $\delta^*(q_i, \Lambda) = q_i$
2. For the word $w = ua$, where a is a symbol, $\delta^*(q_i, w) = \delta\big(\delta^*(q_i, u), a\big)$.

This means that a word w is accepted if and only if $\delta^*(q_0, w)$ is a terminal state, and the language $L(M)$ consists of exactly those words w for which $\delta^*(q_0, w)$ is a terminal state.

Theorem 2. (i) *If the languages L_1 and L_2 are recognized by (their corresponding) finite automata M_1 and M_2, then also the languages $L_1 \cup L_2$, $L_1 \cap L_2$ and $L_1 - L_2$ are recognized by finite automata.*

(ii) *If the language L is recognized by a finite automaton M, then also $\overline{L}$ is recognized by a finite automaton.*

Proof. (i) We may assume that L_1 and L_2 share the same alphabet. If this is not the case originally, we use the union of the original alphabets as our alphabet. We may then further assume that the alphabet of the automata M_1 and M_2 is this shared alphabet Σ, as is easily seen by a simple device. Let us then construct a "product automaton" starting from M_1 and M_2 as follows: If

$$M_1 = (Q, \Sigma, q_0, \delta, A)$$

and

$$M_2 = (S, \Sigma, s_0, \gamma, B),$$

then the product automaton is

$$M_1 \times M_2 = \big(Q \times S, \Sigma, (q_0, s_0), \sigma, C\big)$$

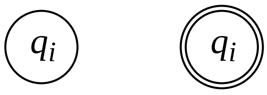
where the set C of terminal states is chosen accordingly. The set of states $Q \times S$ consists of all ordered pairs of states (q_i, s_j) where q_i is in Q and s_j is in S. If $\delta(q_i, a) = q_k$ and $\gamma(s_j, a) = s_\ell$, then we define

$$\sigma\big((q_i, s_j), a\big) = (q_k, s_\ell).$$

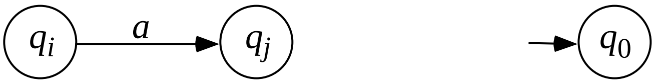
Now, if we want to recognize $L_1 \cup L_2$, we choose C to consist of exactly those pairs (q_i, s_j) where q_i is A or/and s_j is in B, i.e., at least one of the automata is in a terminal state after reading the input word. If, on the other hand, we want to recognize $L_1 \cap L_2$, we take in C all pairs (q_i, s_j) where q_i is in A and s_j is in B, that is, both automata finish their reading in a terminal state. And, if we want to recognize $L_1 - L_2$, we take in C those pairs (q_i, s_j) where q_i is in A and s_j is not in B, so that M_1 finishes in a terminal state after reading the input word but M_2 does not.

(ii) An automaton recognizing the complement $\overline{L}$ is obtained from M simply by changing the set of terminal states to its complement. □

Any finite automaton can be represented graphically as a so-called *state diagram.* A state is then represented by a circle enclosing the symbol of the state, and in particular a terminal state is represented by a double circle:



A state transition $\delta(q_i, a) = q_j$ is represented by an arrow labelled by a, and in particular the initial state is indicated by an incoming arrow:

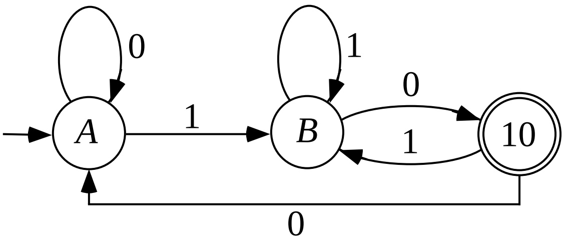


Such a representation is in fact an edge-labelled directed graph, see the course Graph Theory.

Example. *The automaton* $\big(\{A, B, 10\}, \{0, 1\}, A, \delta, \{10\}\big)$ *where* δ *is given by the state transition table*

δ	0	1
A	A	B
B	10	B
10	A	B

is represented by the state transition diagram



The language recognized by the automaton is the regular language $(0 + 1)^*10$.

In general, *the languages recognized by finite automata are exactly all regular languages* (so-called Kleene's Theorem). This will be proved in two parts. The first part[1] can be taken care of immediately, the second part is given later.

Theorem 3. *The language recognized by a finite automaton is regular.*

Proof. Let us consider the finite automaton

$$M = (Q, \Sigma, q_0, \delta, A).$$

A state transition chain of M is a *path* if no state appears in it more than once. Further, a state transition chain is a q_i*-tour* if its first and last state both equal q_i, and q_i appears nowhere else in the chain. A q_i-tour is a q_i*-circuit* if the only state appearing several times in the chain is q_i. Note that there are only a finite number of paths and circuits, but there are infinitely many chains and tours. A state q_i is both a path (a ”null path”) and a q_i-circuit (a ”null circuit”).

Each state transition chain is determined by at least one word, but not infinitely many. Let us denote by R_i the language of words determining exactly all possible q_i-tours. The null circuit corresponds to the language $\{\Lambda\}$.

We show first that R_i is a regular language for each i. We use induction on the number of distinct states appearing in the tour. Let us denote by $R_{S,i}$ the language of

[1] The proof can be transformed to an algorithm in a matrix formalism, the so-called *Kleene Algorithm,* related to the well-known graph-theoretical Floyd–Warshall-type algorithms, cf. the course Graph Theory.

words determining q_i-tours containg only states in the subset S of Q, in particular of course the state q_i. Obviously then $R_i = R_{Q,i}$. The induction is on the cardinality of S, denoted by s, and will prove regularity of each $R_{S,i}$.

Induction Basis, $s = 1$: Now $S = \{q_i\}$, the only possible tours are q_i and q_i, q_i, and the language $R_{S,i}$ is finite and thus regular (indeed, $R_{S,i}$ contains Λ and possibly some of the symbols).

Induction Hypothesis: The claim holds true when $s < h$ where $h \geq 2$.

Induction Statement: The claim holds true when $s = h$.

Proof of the Induction Statement: Each q_i-tour containg only states in S can be expressed—possibly in several ways—in the form

$$q_i, q_{i_1}, K_1, \ldots, q_{i_n}, K_n, q_i$$

where $q_i, q_{i_1}, \ldots, q_{i_n}, q_i$ is a q_i-circuit and q_{i_j}, K_j consists of q_{i_j}-tours containing only states in $S - \{q_i\}$. Let us denote the circuit $q_i, q_{i_1}, \ldots, q_{i_n}, q_i$ itself by $\mathcal{C}$. The set of words

$$a_{j0}a_{j1}\cdots a_{jn} \quad (j = 1, \ldots, \ell)$$

determining the circuit $\mathcal{C}$ as a state transition chain is finite. Now, the language $R_{S-\{q_i\},i_j}$ of all possible words determining q_{i_j}-tours appearing in q_{i_j}, K_j is regular according to the Induction Hypothesis. Let us denote the corresponding regular expression by r_j. Then the language

$$\sum_{j=1}^{\ell} a_{j0}r_1^* a_{j1}r_2^* \cdots r_n^* a_{jn} =_{\text{denote}} r_{\mathcal{C}}$$

of all possible words determining q_i-tours of the given form $q_i, q_{i_1}, K_1, \ldots, q_{i_n}, K_n, q_i$ is regular, too.

Thus, if $\mathcal{C}_1, \ldots, \mathcal{C}_m$ are exactly all q_i-circuits containing only states in S, then the claimed regular language $R_{S,i}$ is $r_{\mathcal{C}_1} + \cdots + r_{\mathcal{C}_m}$.

The proof of the theorem is now very similar to the induction proof above. Any state transition chain leading from the initial state q_0 to a terminal state will either consist of q_0-tours (in case the initial state is a terminal state) or is of the form

$$q_{i_0}, K_0, q_{i_1}, K_1, \ldots, q_{i_n}, K_n$$

where $i_0 = 0$, q_{i_n} is a terminal state, $q_{i_0}, q_{i_1}, \ldots, q_{i_n}$ is a path, and q_{i_j}, K_j consists of q_{i_j}-tours. As above, the language of the corresponding determining words will be regular. □

Note. *Since there often are a lot of arrows in a state diagram, a so-called* partial state diagram *is used, where not all state transitions are indicated. Whenever an automaton, when reading an input word, is in a situation where the diagram does not give a transition, the input is immediately rejected. The corresponding state transition function is a* partial function, *i.e., not defined for all possible arguments. It is fairly easy to see that this does not increase the recognition power of finite automata. Every "partial" finite automaton can be made into an equivalent "total" automaton by adding a new "junk state", and defining all missing state transitions as transitions to the junk state, in particular transitions from the junk state itself.*

A finite automaton can also have idle states that cannot be reached from the initial state. These can be obviously removed.

2.3 Separation of Words. Pumping

The language L *separates* the words w and v if there exists a word u such that one of the words wu and vu is in L and the other one is not. If L does not separate the words w and v, then the words wu and vu are always either both in L or both in $\overline{L}$, depending on u.

There is a connection between the separation power of a language recognized by a finite automaton and the structure of the automaton:

Theorem 4. *If the finite automaton $M = (Q, \Sigma, q_0, \delta, A)$ recognizes the language L and for the words w and v*

$$\delta^*(q_0, w) = \delta^*(q_0, v),$$

then L does not separate w and v.

Proof. As is easily seen, in general

$$\delta^*(q_i, xy) = \delta^*\big(\delta^*(q_i, x), y\big).$$

So

$$\delta^*(q_0, wu) = \delta^*\big(\delta^*(q_0, w), u\big) = \delta^*\big(\delta^*(q_0, v), u\big) = \delta^*(q_0, vu).$$

Thus, depending on whether or not this is a terminal state, the words wu and vu are both in L or both in $\overline{L}$. □

Corollary. *If the language L separates any two of the n words $w_1, \dots, w_n$, then L is not recognized by any finite automaton with less than n states.*

Proof. If the finite automaton $M = (Q, \Sigma, q_0, \delta, A)$ has less than n states then one of the states appears at least twice among the states

$$\delta^*(q_0, w_1)\ , \dots ,\ \delta^*(q_0, w_n).$$ □

The language L_{pal} of all *palindromes* over an alphabet is an example of a language that cannot be recognized using only a finite number of states (assuming that there are at least two symbols in the alphabet). A word w is a palindrome if $\hat{w} = w$. Indeed L_{pal} separates all pairs of words, any two words can be extended to a palindrome and a nonpalindrome. There are numerous languages with a similar property, e.g. the language L_{sqr} of all so-called *square words,* i.e., words of the form w^2.

Separation power is also closely connected with the construction of the smallest finite automaton recognizing a language, measured by the number of states, the so-called *minimization* of a finite automaton. More about this later.

Finally let us consider a situation rather like the one in the above proof where a finite automaton has exactly n states and the word to be accepted is at least of length n:

$$x = a_1 a_2 \cdots a_n y,$$

where $a_1, \dots, a_n$ are input symbols and y is a word. Among the states

$$q_0 = \delta^*(q_0, \Lambda)\ ,\ \delta^*(q_0, a_1)\ ,\ \delta^*(q_0, a_1 a_2)\ , \dots ,\ \delta^*(q_0, a_1 a_2 \cdots a_n)$$

there at least two identical ones, say

$$\delta^*(q_0, a_1 a_2 \cdots a_i) = \delta^*(q_0, a_1 a_2 \cdots a_{i+p}).$$

Let us denote for brevity

$$u = a_1 \cdots a_i \quad , \quad v = a_{i+1} \cdots a_{i+p} \quad \text{and} \quad w = a_{i+p+1} \cdots a_n y.$$

But then the words $uv^m w$ ($m = 0, 1, \dots$) clearly will be accepted as well! This result is known as the

Pumping Lemma ("*uvw*-Lemma"). *If the language L can be recognized by a finite automaton with n states, $x \in L$ and $|x| \geq n$, then x can be written in the form $x = uvw$ where $|uv| \leq n$, $v \neq \Lambda$ and the "pumped" words $uv^m w$ are* all *in L.*

Pumping Lemma is often used to show that a language is not regular, since otherwise the pumping would produce words easily seen not to be in the language.

2.4 Nondeterministic Finite Automata

Nondeterminism means freedom of making some choices, i.e., any of the several possible given alternatives can be chosen. The allowed alternatives must however be clearly defined and (usually) finite in number. Some alternatives may be better than others, that is, the goal can be achieved only through proper choices.

In the case of a finite automaton *nondeterminism* means a choice in state transition, there may be several alternative next states to be chosen from, and there may be several initial states to start with. This is indicated by letting the values of the transition function to be sets of states containing all possible alternatives for the next state. Such a set can be empty, which means that no state transition is possible, cf. the Note on page 8 on partial state diagrams.

Finite automata dealt with before were always deterministic. We now have to mention carefully the type of a finite automaton.

Defined formally a *nondeterministic finite automaton* (*NFA*) is a quintuple $M = (Q, \Sigma, S, \delta, A)$ where

- Q, Σ and A are as for the deterministic finite automaton;
- S is the *set of initial states;*
- δ is the (*state*) *transition function* which maps each pair (q_i, a), where q_i is a state and a is an input symbol, to exactly one subset T of the state set Q: $\delta(q_i, a) = T$.

Note that either S or T (or both) can be empty. The set of all subsets of Q, i.e., the powerset of Q, is usually denoted by 2^Q.

We can immediately extend the state transition function δ in such a way that its first argument is a set of states:

$$\hat{\delta}(\emptyset, a) = \emptyset \quad \text{and} \quad \hat{\delta}(U, a) = \bigcup_{q_i \in U} \delta(q_i, a).$$

We can further define $\hat{\delta}^*$ as δ^* was defined above:

$$\hat{\delta}^*(U, \Lambda) = U \quad \text{and} \quad \hat{\delta}^*(U, ua) = \hat{\delta}\big(\hat{\delta}^*(U, u), a\big).$$

M accepts a word w if there is at least one terminal state in the set of states $\hat{\delta}^*(S, w)$. Λ is accepted if there is at least one terminal state in S. The set of exactly all words accepted by M is the language $L(M)$ *recognized* by M.

The nondeterministic finite automaton may be thought of as a generalization of the deterministic finite automaton, obtained by identifying in the latter each state q_i by the corresponding singleton set $\{q_i\}$. It is however no more powerful in recognition ability:

Theorem 5. *If a language can be recognized by a nondeterministic finite automaton, then it can be recognized by deterministic finite automaton, too, and is thus regular.*

Proof. Consider a language L recognized by the nondeterministic finite automaton $M = (Q, \Sigma, S, \delta, A)$. The equivalent deterministic finite automaton is then $M_1 = (Q_1, \Sigma, q_0, \delta_1, A_1)$ where

$$Q_1 = 2^Q \quad , \quad q_0 = S \quad , \quad \delta_1 = \hat{\delta} \ ,$$

and A_1 consists of exactly all sets of states having a nonempty intersection with A. The states of M_1 are thus all sets of states of M.

We clearly have $\delta_1^*(q_0, w) = \hat{\delta}^*(S, w)$, so M and M_1 accept exactly the same words, and M_1 recognizes the language L. □

A somewhat different kind of nondeterminism is obtained when in addition so-called Λ-*transitions* are allowed. The state transition function δ of a nondeterministic finite automaton is then extended to all pairs (q_i, Λ) where q_i is a state. The resulting automaton is a *nondeterministic finite automaton with* Λ-*transitions* (Λ-*NFA*). The state transition $\delta(q_i, \Lambda) = T$ is interpreted as allowing the automaton to move from the state q_i to any of the states in T, without reading a new input symbol. If $\delta(q_i, \Lambda) = \emptyset$ or $\delta(q_i, \Lambda) = \{q_i\}$, then there is no Λ-transition from q_i to any other state.

For transitions other than Λ-transitions δ can be extended to sets of states exactly as before. For Λ-transitions the extension is analogous:

$$\hat{\delta}(\emptyset, \Lambda) = \emptyset \quad \text{and} \quad \hat{\delta}(U, \Lambda) = \bigcup_{q_i \in U} \delta(q_i, \Lambda).$$

Further, we can extend δ to the "star function" for the Λ-transitions": $\hat{\delta}^*(U, \Lambda) = V$ if

- states in U are also in V;
- states in $\hat{\delta}(V, \Lambda)$ are also in V;
- each state in V is either a state in U or then it can be achieved by repeated Λ-transitions starting from some state in U.

And finally we can extend δ for transitions other than the Λ-transitions:

$$\hat{\delta}^*(U, ua) = \hat{\delta}^*\big(\hat{\delta}\big(\hat{\delta}^*(U, u), a\big), \Lambda\big).$$

Note in particular that for an input symbol a

$$\hat{\delta}^*(U, a) = \hat{\delta}^*\big(\hat{\delta}\big(\hat{\delta}^*(U, \Lambda), a\big), \Lambda\big),$$

i.e., first there are Λ-transitions, then the "proper" state transition determined by a, and finally again Λ-transitions.

The words accepted and the language recognized by a Λ-NFA are defined as before. But still there will be no more recognition power:

Theorem 6. *If a language can be recognized by a* Λ-*NFA, then it can also be recognized by a nondeterministic finite automaton without* Λ-*transitions, and is thus again regular.*

Proof. Consider a language L recognized by the Λ-NFA $M = (Q, \Sigma, S, \delta, A)$. The equivalent nondeterministic finite automaton (without Λ-transitions) is then $M_1 = (Q, \Sigma, S_1, \delta_1, A)$ where

$$S_1 = \hat{\delta}^*(S, \Lambda) \quad \text{and} \quad \delta_1(q_i, a) = \hat{\delta}^*\big(\{q_i\}, a\big).$$

We clearly have $\hat{\delta}_1^*(S_1, w) = \hat{\delta}^*(S, w)$, so M and M_1 accept exactly the same words, and M_1 recognizes the language L. Note especially that if M accepts Λ, then it is possible to get from some state in S to some terminal state using only Λ-transitions, and the terminal state is then in S_1. □

Also nonterministic automata—with or without Λ-transitions—can be given using state diagrams in an obvious fashion. If there are several parallel arrows connecting a state to another state (or itself), then they are often replaced by one arrow labelled by the list of labels of the original arrows.

2.5 Kleene's Theorem

In Theorem 3 above it was proved that a language recognized by a deterministic finite automaton is always regular, and later this was shown for nondeterministic automata, too. The converse holds true also.

Kleene's Theorem. *Regular languages are exactly all languages recognized by finite automata.*

Proof. What remains to be shown is that every regular language can be recognized by a finite automaton. Having the structure of a regular expression in mind, we need to show first that the "atomic" languages $\emptyset$, $\{\Lambda\}$ and $\{a\}$, where a is a symbol, can be recognized by finite automata. This is quite easy. Second, we need to show that if the languages L_1 and L_2 can be recognized by finite automata, then so can the languages $L_1 \cup L_2$ and L_1L_2. For union this was done in Theorem 2. And third, we need to show that if the language L is recognized by a finite automaton, then so is L^+, and consequently also $L^* = L^+ \cup \{\Lambda\}$.

Let us then assume that the languages L_1 and L_2 are recognized by the nondeterministic finite automata

$$M_1 = (Q_1, \Sigma_1, S_1, \delta_1, A_1) \quad \text{and} \quad M_2 = (Q_2, \Sigma_2, S_2, \delta_2, A_2),$$

respectively. It may be assumed that $\Sigma_1 = \Sigma_2 =_{\text{denote}} \Sigma$ (just add null transitions). And further, it may be assumed that the sets of states Q_1 and Q_2 are disjoint. The new finite automaton recognizing L_1L_2 is now

$$M = (Q, \Sigma, S_1, \delta, A_2)$$

where $Q = Q_1 \cup Q_2$ and δ is defined by

$$\delta(q, a) = \begin{cases} \delta_1(q, a) \text{ if } q \in Q_1 \\ \delta_2(q, a) \text{ if } q \in Q_2 \end{cases} \quad \text{and} \quad \delta(q, \Lambda) = \begin{cases} \delta_1(q, \Lambda) & \text{if } q \in Q_1 - A_1 \\ \delta_1(q, \Lambda) \cup S_2 & \text{if } q \in A_1 \\ \delta_2(q, \Lambda) & \text{if } q \in Q_2. \end{cases}$$

A terminal state of M can be reached only by first moving using a Λ-transition from a terminal state of M_1 to an initial state of M_2, and this takes place when M_1 accepted the prefix of the input word then read. To reach the terminal state after that, the remaining suffix must be in L_2.

Finally consider the case where the language L is recognized by the nondeterministic finite automaton

$$M = (Q, \Sigma, S, \delta, A).$$

Then L^+ is recognized by the finite automaton

$$M' = (Q, \Sigma, S, \delta', A)$$

where

$$\delta'(q, a) = \delta(q, a) \quad \text{and} \quad \delta'(q, \Lambda) = \begin{cases} \delta(q, \Lambda) & \text{if } q \notin A \\ \delta(q, \Lambda) \cup S & \text{if } q \in A. \end{cases}$$

It is always possible to move from a terminal state to an initial state using a Λ-transition. This makes possible repeated concatenation. If the input word is divided into subwords according to where these Λ-transitions take place, then the subwords are all in the language L. □

Kleene's Theorem and other theorems above give characterizations for regular languages both via regular expressions and as languages recognized by finite automata of various kinds (DFA, NFA and Λ-NFA). These characterizations are different in nature and useful in different situations. Where a regular expression is easy to use, a finite automaton can be a quite difficult tool to deal with. On the other hand, finite automata can make easy many things which would be very tedious using regular expressions. This is seen in the proofs above, too, just think how difficult it would be to show that the intersection of two regular languages is again regular, by directly using regular expressions.

2.6 Minimization of Automata

There are many finite automata recognizing the same regular language L. A deterministic finite automaton recognizing L with the smallest possible number of states is a *minimal finite automaton.* Such a minimal automaton can be found by studying the structure of the language L. To start with, L must then of course be regular and specified somehow. Let us however consider this first in a quite general context. The alphabet is Σ.

In Section 2.3 separation of words by the language L was discussed. Let us now denote $w \not\equiv_L v$ if the language L separates the words w and v, and correspondingly $w \equiv_L v$ if L does not separate w and v. In the latter case we say that the words w and v are *L-equivalent.* We may obviously agree that always $w \equiv_L w$, and clearly, if $w \equiv_L v$ then also $v \equiv_L w$.

Lemma. *If $w \equiv_L v$ and $v \equiv_L u$, then also $w \equiv_L u$. (That is, $\equiv_L$ is* transitive.*)*

Proof. If $w \equiv_L v$ and $v \equiv_L u$, and z is a word, then there are two alternatives. If vz is in L, then so are wz and uz. On the other hand, if vz is not in L, then neither are wz and uz. We deduce thus that $w \equiv_L u$. □

As a consequence, the words in Σ^* are partitioned into so-called *L-equivalence classes:* Words w and v are in the same class if and only if they are L-equivalent. The class containg the word w is denoted by $[w]$. The "representative" can be any other word v in the class: If $w \equiv_L v$, then $[w] = [v]$. Note that if $w \not\equiv_L u$, then the classes $[w]$ and $[u]$

do not intersect, since a common word v would mean $w \equiv_L v$ and $v \equiv_L u$ and, by the Lemma, $w \equiv_L u$.

The number of all L-equivalence classes is called the *index* of the language L. In general it can be infinite, Theorem 4 however immediately implies

Theorem 7. *If a language is recognized by a deterministic finite automaton with n states, then the index of the language is at most n.*

On the other hand,

Theorem 8. *If the index of the language L is n, then L can be recognized by a deterministic finite automaton with n states.*

Proof. Consider a language L of index n, and its n different equivalence classes

$$[x_0], [x_1], \dots, [x_{n-1}]$$

where in particular $x_0 = \Lambda$.

A deterministic finite automaton $M = \big(Q, \Sigma, q_0, \delta, A\big)$ recognizing L is then obtained by taking

$$Q = \big\{[x_0], [x_1], \dots, [x_{n-1}]\big\} \quad \text{and} \quad q_0 = [x_0] = [\Lambda],$$

letting A consist of exactly those equivalence classes that contain words in L, and defining

$$\delta\big([x_i], a\big) = [x_i a].$$

δ is then well-defined because if $x \equiv_L y$ then obviously also $xa \equiv_L ya$. The corresponding δ^* is also immediate:

$$\delta^*\big([x_i], y\big) = [x_i y].$$

$L(M)$ will then consist of exactly those words w for which

$$\delta^*\big([\Lambda], w\big) = [\Lambda w] = [w]$$

is a terminal state of M, i.e., contains words of L.

Apparently $L \subseteq L(M)$, because if $w \in L$ then $[w]$ is a terminal state of M. On the other hand, if there is a word v of L in $[w]$ then w itself is in L, otherwise we would have $w\Lambda \notin L$ and $v\Lambda \in L$ and L would thus separate w and v. In other words, if $w \in L$ then $[w] \subseteq L$. So $L(M) = L$. □

Corollary. *The number of states of a minimal automaton recognizing the language L is the value of the index of L.*

Corollary (Myhill–Nerode Theorem). *A language is regular if and only if it has a finite index.*

If a regular language L is defined by a deterministic finite automaton $M = (Q, \Sigma, q_0, \delta, A)$ recognizing it, then the minimization naturally starts from M. The first step is to remove all idle states of M, i.e., states that cannot be reached from the initial state. After this we may assume that all states of M can expressed as $\delta^*(q_0, w)$ for some word w.

For the minimization the states of M are partitioned into *M-equivalence classes* as follows. The states q_i and q_j are not M-equivalent if there is a word u such that one of the states $\delta^*(q_i, u)$ and $\delta^*(q_j, u)$ is terminal and the other one is not, denoted by $q_i \not\equiv_M q_j$. If there is no such word u, then q_i and q_j are *M-equivalent,* denoted by $q_i \equiv_M q_j$. We may obviously assume $q_i \equiv_M q_i$. Furthermore, if $q_i \equiv_M q_j$, then also $q_j \equiv_M q_i$, and if $q_i \equiv_M q_j$ and $q_j \equiv_M q_k$ it follows that $q_i \equiv_M q_k$. Each equivalence class consists of mutually M-equivalent states, and the classes are disjoint. (Cf. the L-equivalence classes and the equivalence relation $\equiv_L$.) Let us denote the M-equivalence class represented by the state q_i by $\langle q_i \rangle$. Note that it does not matter which of the M-equivalent states is chosen as the representative of the class. Let us then denote the set of all M-equivalence classes by $\mathcal{Q}$.

M-equivalence and L-equivalence are related since $\big\langle \delta^*(q_0, w) \big\rangle = \big\langle \delta^*(q_0, v) \big\rangle$ if and only if $[w] = [v]$. Because now all states can be reached from the initial state, there are as many M-equivalence classes as there are L-equivalence classes, i.e., the number given by the index of L. Moreover, M-equivalence classes and L-equivalence classes are in a 1–1 correspondence:

$$\big\langle \delta^*(q_0, w) \big\rangle \rightleftharpoons [w],$$

in particular $\langle q_0 \rangle \rightleftharpoons [\Lambda]$.

The minimal automaton corresponding to the construct in the proof of Theorem 8 is now

$$M_{\min} = \big(\mathcal{Q}, \Sigma, \langle q_0 \rangle, \delta_{\min}, \mathcal{A}\big)$$

where $\mathcal{A}$ consists of those M-equivalence classes that contain at least one terminal state, and $\delta_{\min}$ is given by

$$\delta_{\min}\big(\langle q_i \rangle, a\big) = \big\langle \delta(q_i, a) \big\rangle.$$

Note that if an M-equivalence class contains a terminal state, then all its states are terminal. Note also that if $q_i \equiv_M q_j$, then $\delta(q_i, a) \equiv_M \delta(q_j, a)$, so that $\delta_{\min}$ is well-defined.

A somewhat similar construction can be started from a nondeterministic finite automaton, with or without Λ-transitions.

2.7 Decidability Problems

Nearly every characterization problem is algorithmically decidable for regular languages. The most common ones are the following (where L or L_1 and L_2 are given regular languages):

- *Emptiness Problem:* Is the language L empty (i.e., does it equal $\emptyset$)?

  It is fairly easy to check for a given finite automaton recognizing L, whether or not there is a state transition chain from an initial state to a terminal state.

- *Inclusion Problem:* Is the language L_1 included in the language L_2?

  Clearly $L_1 \subseteq L_2$ if and only if $L_1 - L_2 = \emptyset$.

- *Equivalence Problem:* Is $L_1 = L_2$?

  Clearly $L_1 = L_2$ if and only if $L_1 \subseteq L_2$ and $L_2 \subseteq L_1$.

- *Finiteness Problem:* Is L a finite language?

  It is fairly easy to check for a given finite automaton recognizing L, whether or not it has arbitrarily long state transition chains from an initial state to a terminal state. Cf. the proof of Theorem 3.

- *Membership Problem:* Is the given word w in the language L or not?

  Using a given finite automaton recognizing L it is easy to check whether or not it accepts the given input word w.

2.8 Sequential Machines and Tranducers (A Brief Overview)

A sequential machine is simply a deterministic finite automaton equipped with output. Formally a *sequential machine* (*SM*) is a sextuple

$$S = (Q, \Sigma, \Delta, q_0, \delta, \tau)$$

where Q, Σ, q_0 and δ as in a deterministic finite automaton, Δ is the *output alphabet* and τ is the *output function* mapping each pair (q_i, a) to a symbol in Δ. Terminal states will not be needed.

δ is extended to the corresponding "star function" δ^* in the usual fashion. The extension of τ is given by the following:

1. $\tau^*(q_i, \Lambda) = \Lambda$

2. For a word $w = ua$ where a is a symbol,

$$\tau^*(q_i, ua) = \tau^*(q_i, u)\tau\big(\delta^*(q_i, u), a\big).$$

The output word corresponding to the input word w is then $\tau^*(q_0, w)$. The sequential machine S maps the language L to the language

$$S(L) = \big\{\tau^*(q_0, w) \bigm| w \in L\big\}.$$

Using an automaton construct it is fairly simple to show that a sequential machine always maps a regular language to a regular language.

A *generalized sequential machine* (*GSM*)[2] is as a sequential machine except that values of the output function are words over Δ. Again it is not difficult to see that a generalized sequential machine always maps a regular language to a regular language.

If a generalized sequential machine has only one state, then the mapping of words (or languages) defined by it is called a *morphism.* Since there is only one state, it is not necessary to write it down explicitly:

$$\tau^*(\Lambda) = \Lambda \quad \text{and} \quad \tau^*(ua) = \tau^*(u)\tau(a).$$

We then have for all words u and v over Σ the morphic equality

$$\tau^*(uv) = \tau^*(u)\tau^*(v).$$

It is particularly easy to see that a morphism maps a regular language to a regular language: Just map the corresponding regular expression using the morphism.

There are nondeterministic versions of sequential machines and generalized sequential machines. A more general concept however is a so-called transducer. Formally a *transducer* is a quintuple

$$T = (Q, \Sigma, \Delta, S, \delta)$$

where Q, Σ and Δ are as for sequential machines, S is a set of initial states and δ is the *transition-output function* that maps each pair (q_i, a) to a finite set of pairs of the form (q_j, u). This is interpreted as follows: When reading the input symbol a in state q_i the

[2] Sometimes GSM's do have terminal states, too, they then map only words leading to a terminal state.

transducer T can move to any state q_j outputting the word u, provided that the pair (q_j, u) is in $\delta(q_i, a)$.

Definition of the corresponding ”hat-star function” $\hat{\delta}^*$ is now a bit tedious (omitted here), anyway the *transduction* of the language L by T is

$$T(L) = \bigcup_{w \in L} \big\{ u \bigm| (q_i, u) \in \hat{\delta}^*(S, w) \text{ for some state } q_i \big\}.$$

In this case, too, it is the case that a transducer always maps a regular language to a regular language, i.e., transduction preserves regularity.

The mapping given by a transducer with only one state is often called a *finite substitution.* As for morphisms, it is simple to see that a finite substitution preserves regularity: Just map the corresponding regular expression by the finite substitution.

> ”If our experiences are in any sense typical,
> we are certain that the mere mention
> of the word ”grammar” will produce in you
> an immediate feeling of severe nausea.”
>
> (John T. Grinder & Suzette Haden Elgin:
> *Guide to Transformational Grammar*)

Chapter 3

GRAMMARS

3.1 Rewriting Systems

A rewriting system, and a grammar in particular, gives rules endless repetition of which produces all words of a language, starting from a given initial word. Often only words of a certain type will be allowed in the language. This kind of operation is in a sense dual to that of an automaton recognizing a language.

Definition. *A* rewriting system[1] *(*RWS*) is a pair $R = (\Sigma, P)$ where*

- *Σ is an alphabet;*
- *$P = \big\{(p_1, q_1), \dots, (p_n, q_n)\big\}$ is a finite set of ordered pairs of words over Σ, so-called* productions. *A production (p_i, q_i) is usually written in the form $p_i \to q_i$.*

The word v is *directly derived* by R from the word w if $w = rp_is$ and $v = rq_is$ for some production (p_i, q_i), this is denoted by

$$w \Rightarrow_R v.$$

From $\Rightarrow_R$ the corresponding ”star relation”[2] $\Rightarrow_R^*$ is obtained as follows (cf. extension of a transition function to a ”star function”):

1. $w \Rightarrow_R^* w$ for all words w over Σ.
2. If $w \Rightarrow_R v$, it follows that $w \Rightarrow_R^* v$.
3. If $w \Rightarrow_R^* v$ and $v \Rightarrow_R^* u$, it follows that $w \Rightarrow_R^* u$.
4. $w \Rightarrow_R^* v$ only if this follows from items 1.–3.

If then $w \Rightarrow_R^* v$, we say that v is *derived* from w by R. This means that either $v = w$ or/and there is a chain of direct derivations

$$w = w_0 \Rightarrow_R w_1 \Rightarrow_R \cdots \Rightarrow_R w_\ell = v,$$

a so-called *derivation* of v from w. ℓ is the *length* of the derivation.

[1] Rewriting systems are also called *semi-Thue systems.* In a proper *Thue system* there is the additional requirement that if $p \to q$ is a production then so is $q \to p$, i.e., each production $p \leftrightarrow q$ is two-way.

[2] Called the *reflexive-transitive closure* of $\Rightarrow_R$.

As such the only thing an RWS R does is to derive words from other words. However, if a set A of initial words, so-called *axioms,* is fixed, then the *language generated by* R is defined as

$$L_{\text{gen}}(R, A) = \{v \mid w \Rightarrow_R^* v \text{ for some word } w \in A\}.$$

Usually this A contains only one word, or it is finite or at least regular. Such an RWS is "grammar-like".

An RWS can also be made "automaton-like" by specifying a language T of allowed *terminal words.* Then the *language recognized by* the RWS R is

$$L_{\text{rec}}(R, T) = \{w \mid w \Rightarrow_R^* v \text{ for some word } v \in T\}.$$

This T is usually regular, in fact a common choice is $T = \Delta^*$ for some subalphabet Δ of Σ. The symbols of Δ are then called *terminal symbols* (or terminals) and the symbols in $\Sigma - \Delta$ *nonterminal symbols* (or nonterminals).

Example. *A deterministic finite automaton $M = (Q, \Sigma, q_0, \delta, B)$ can be transformed to an RWS in (at least) two ways. It will be assumed here that the intersection $\Sigma \cap Q$ is empty.*

The first way is to take the RWS $R_1 = (\Omega, P_1)$ where $\Omega = \Sigma \cup Q$ and P_1 contains exactly all productions

$$q_i a \to q_j \quad \textit{where} \quad \delta(q_i, a) = q_j,$$

and the productions

$$a \to q_0 a \quad \textit{where} \quad a \in \Sigma.$$

Taking T to be the language $B + \Lambda$ or B, depending on whether or not Λ is in $L(M)$, we have then

$$L_{\text{rec}}(R_1, T) \cap \Sigma^* = L(M).$$

A typical derivation accepting the word $w = a_1 \cdots a_m$ is of the form

$$w \Rightarrow_{R_1} q_0 w \Rightarrow_{R_1} q_{i_1} a_2 \cdots a_m \Rightarrow_{R_1}^* q_{i_m}$$

where q_{i_m} is a terminal state. Finite automata are thus essentially rewriting systems!

Another way to transform M to an equivalent RWS is to take $R_2 = (\Omega,\ P_2)$ where P_2 contains exactly all productions

$$q_i \to a q_j \quad \textit{where} \quad \delta(q_i, a) = q_j,$$

and the production $q_i \to \Lambda$ for each terminal state q_i. Then

$$L_{\text{gen}}\big(R_2, \{q_0\}\big) \cap \Sigma^* = L(M).$$

An automaton is thus essentially transformed to a grammar!

There are numerous ways to vary the generating/recognizing mechanism of a RWS.

Example. (**Markov's normal algorithm**) *Here the productions of an RWS are given as an ordered list*

$$P : p_1 \to q_1, \ldots, p_n \to q_n,$$

and a subset F of P is specified, the so-called terminal productions. *In a derivation it is required that always the first applicable production in the list is used, and it is used in the first applicable position in the word to be rewritten. Thus, if $p_i \to q_i$ is the first* *applicable production in the list, then it has to be applied to the leftmost subword p_i of the word to be rewritten. The derivation* halts *when no applicable production exists or when a terminal production is applied. Starting from a word w the normal algorithm either halts and generates a unique word v, or then it does not stop at all. In the former case the word v is interpreted as the* output produced by the input w, *in the latter case there is no output. Normal algorithms have a universal computing power, that is, everything that can be computed can be computed by normal algorithms. They can also be used for recognition of languages: An input word is recognized when the derivation starting from the word halts. Normal algorithms have a universal recognition power, too.*

3.2 Grammars

A grammar is a rewriting system of a special type where the alphabet is partitioned into two sets of symbols, the so-called *terminal symbols* (terminals) or *constants* and the so-called *nonterminal symbols* (nonterminals) or *variables,* and one of the nonterminals is specified as the axiom (cf. above).

Definition. *A* grammar[3] *is a quadruple $G = (\Sigma_{\mathrm{N}}, \Sigma_{\mathrm{T}}, X_0, P)$ where Σ_{N} is the nonterminal alphabet, Σ_{T} is the terminal alphabet, $X_0 \in \Sigma_{\mathrm{N}}$ is the axiom, and P consists of productions $p_i \to q_i$ such that at least one nonterminal symbol appears in p_i.*

If G is a grammar, then $(\Sigma_{\mathrm{N}} \cup \Sigma_{\mathrm{T}}, P)$ is an RWS, the so-called *RWS induced by* G. We denote $\Sigma = \Sigma_{\mathrm{N}} \cup \Sigma_{\mathrm{T}}$ in the sequel. It is customary to denote terminals by small letters $(a, b, c, \dots,$ etc.) and nonterminals by capital letters $(X, Y, Z, \dots,$ etc.). The relations $\Rightarrow$ and $\Rightarrow^*$, obtained from the RWS induced by G, give the corresponding relations $\Rightarrow_G$ and $\Rightarrow_G^*$ for G. The *language generated by* G is then

$$L(G) = \{w \mid X_0 \Rightarrow_G^* w \text{ and } w \in \Sigma_{\mathrm{T}}^*\}.$$

A grammar G is

- *context-free* or *CF* if in each production $p_i \to q_i$ the left hand side p_i is a single nonterminal. Rewriting then does not depend on which ”context” the nonterminal appears in.

- *linear* if it is CF and the right hand side of each production contains at most one nonterminal. A CF grammar that is not linear is *nonlinear.*

- *context-sensitive* or *CS*[4] if each production is of the form $p_i \to q_i$ where

  $$p_i = u_i X_i v_i \quad \text{and} \quad q_i = u_i w_i v_i,$$

  for some $u_i, v_i \in \Sigma^*$, $X_i \in \Sigma_{\mathrm{N}}$ and $w_i \in \Sigma^+$. The only possible exception is the production $X_0 \to \Lambda$, provided that X_0 does not appear in the right hand side of any of the productions. This exception makes it possible to include the empty word in the generated language $L(G)$, when needed. Rewriting now depends on the ”context” or neighborhood the nonterminal X_i occurs in.

[3] To be exact, a so-called *generative* grammar. There is also a so-called *analytical* grammar that works in a dual automaton-like fashion.

[4] Sometimes a CS grammar is simply defined as a length-increasing grammar. This does not affect the family of languages generated.

- *length-increasing* if each production $p_i \to q_i$ satisfies $|p_i| \leq |q_i|$, again with the possible exception of the production $X_0 \to \Lambda$, provided that X_0 does not appear in the right hand side of any of the productions.

Example. *The linear grammar*

$$G = \big(\{X\}, \{a, b\}, X, \{X \to \Lambda, X \to a, X \to b, X \to aXa, X \to bXb\}\big)$$

generates the language L_{pal} of palindromes over the alphabet $\{a, b\}$. (Recall that a palindrome is a word w such that $\hat{w} = w$.) This grammar is not length increasing (why not?).

Example. *The grammar*

$$\begin{aligned} G = &\big(\{X_0, \$, X, Y\}, \{a\}, X_0, \{X_0 \to \$X\$, \$X \to \$Y, YX \to XXY, \\ &\qquad\qquad\qquad\qquad Y\$ \to XX\$, X \to a, \$ \to \Lambda\}\big) \end{aligned}$$

generates the language $\{a^{2^n} \mid n \geq 0\}$. \$ is an endmarker and Y "moves" from left to right squaring each X. If the productions $X \to a$ and $\$ \to \Lambda$ are applied prematurely, it is not possible to get rid of the Y thereafter, and the derivation will not terminate. The grammar is neither CF, CS nor length-increasing.

3.3 Chomsky's Hierarchy

In *Chomsky's hierachy* grammars are divided into four types:

- *Type 0:* No restrictions.
- *Type 1:* CS grammars.
- *Type 2:* CF grammars.
- *Type 3:* Linear grammars having productions of the form $X_i \to wX_j$ or $X_j \to w$ where X_i and X_j are nonterminals and $w \in \Sigma_{\mathrm{T}}^*$, the so-called *right-linear grammars.*[5]

Grammars of Types 1 and 2 generate the so-called *CS-languages* and *CF-languages,* respectively, the corresponding families of languages are denoted by $\mathcal{CS}$ and $\mathcal{CF}$.

Languages generated by Type 0 grammars are called *computably enumerable languages* (*CE-languages*), the corresponding family is denoted by $\mathcal{CE}$. The name comes from the fact that words in a CE-language can be listed algorithmically, i.e., there is an algorithm which running indefinitely outputs exactly all words of the language one by one. Such an algorithm is in fact obtained via the derivation mechanism of the grammar. On the other hand, languages other than CE-languages cannot be listed algorithmically this way. This is because of the formal and generally accepted definition of algorithm!

Languages generated by Type 3 grammars are familiar:

Theorem 9. *The family of languages generated by Type 3 grammars is the family $\mathcal{R}$ of regular languages.*

[5] There is of course the corresponding *left-linear grammar* where productions are of the form $X_i \to X_j w$ and $X_j \to w$. Type 3 could equally well be defined using this.

Proof. This is essentially the first example in Section 3.1. To get a right-linear grammar just take the axiom q_0. On the other hand, to show that a right-linear grammar generates a regular language, a Λ-NFA simulating the grammar is used (this is left to the reader as an exercise). □

Chomsky's hierarchy may thus be thought of as a hierarchy of families of languages as well:

$$\mathcal{R} \subset \mathcal{CF} \subset \mathcal{CS} \subset \mathcal{CE}.$$

As noted above, the language L_{pal} of all palindromes over an alphabet containg at least two symbols is CF but not regular, showing that the first inclusion is proper. The other inclusions are proper, too, as will be seen later.

Regular languages are closed under many operations on languages, i.e., operating on regular languages always produces a regular language. Such operations include e.g. set-theoretic operations, concatenation, concatenation closure, and mirror image. Other families of languages in Chomsky's hierarchy are closed under quite a few language operations, too. This in fact makes them natural units of classification: A larger family always contains languages somehow radically different, not only languages obtained from the ones in the smaller family by some common operation. Families other than $\mathcal{R}$ are however not closed even under all operations above, in particular intersection and complementation are troublesome.

Lemma. *A grammar can always be replaced by a grammar of the same type that generates the same language and has no terminals on left hand sides of productions.*

Proof. If the initial grammar is $G = (\Sigma_{\mathrm{N}}, \Sigma_{\mathrm{T}}, X_0, P)$, then the new grammar is $G' = (\Sigma'_{\mathrm{N}}, \Sigma_{\mathrm{T}}, X_0, P')$ where

$$\Sigma'_{\mathrm{N}} = \Sigma_{\mathrm{N}} \cup \Sigma'_{\mathrm{T}} \quad , \quad \Sigma'_{\mathrm{T}} = \{a' \mid a \in \Sigma_{\mathrm{T}}\}$$

(Σ'_{T} is a disjoint "shadow alphabet" of Σ_{T}), and P' is obtained from P by changing each terminal symbol a in each production to the corresponding "primed" symbol a', and adding the terminating productions $a' \to a$. □

Theorem 10. *Each family in the Chomsky hierarchy is closed under the operations* $\cup$*, concatenation,* * *and* $^+$*.*

Proof. The case of the family $\mathcal{R}$ was already dealt with. If the languages L and L' are generated by grammars

$$G = (\Sigma_{\mathrm{N}}, \Sigma_{\mathrm{T}}, X_0, P) \quad \text{and} \quad G' = (\Sigma'_{\mathrm{N}}, \Sigma'_{\mathrm{T}}, X'_0, P')$$

of the same type, then it may be assumed first that $\Sigma_{\mathrm{N}} \cap \Sigma'_{\mathrm{N}} = \emptyset$, and second that left hand sides of productions do not contain terminals (by the Lemma above).

$L \cup L'$ is then generated by the grammar

$$H = (\Delta_{\mathrm{N}}, \Delta_{\mathrm{T}}, Y_0, Q)$$

of the same type where

$$\Delta_{\mathrm{N}} = \Sigma_{\mathrm{N}} \cup \Sigma'_{\mathrm{N}} \cup \{Y_0\} \quad , \quad \Delta_{\mathrm{T}} = \Sigma_{\mathrm{T}} \cup \Sigma'_{\mathrm{T}},$$

Y_0 is a new nonterminal, and Q is obtained in the following way:

1. Take all productions in P and P'.

2. If the type is Type 1, remove the productions $X_0 \to \Lambda$ and $X_0' \to \Lambda$ (if any).

3. Add the productions $Y_0 \to X_0$ and $Y_0 \to X_0'$.

4. If the type is Type 1 and Λ is in L or L', add the production $Y_0 \to \Lambda$.

LL' in turn is generated by the grammar H when items 3. and 4. are replaced by

3'. Add the production $Y_0 \to X_0X_0'$. If the type is Type 1 and Λ is in L (resp. L'), add the production $Y_0 \to X_0'$ (resp. $Y_0 \to X_0$).

4'. If the type is Type 1 and Λ appears in both L and L', add the production $Y_0 \to \Lambda$.

The type of the grammar is again preserved. Note how very important it is to make the above two assumptions, so that adjacent derivations do not disturb each other for grammars of Types 0 and 1.

If G is of Type 2, then L^* is generated by the grammar

$$K = \big(\Sigma_{\rm N} \cup \{Y_0\}, \Sigma_{\rm T}, Y_0, Q\big)$$

where Q is obtained from P by adding the productions

$$Y_0 \to \Lambda \quad \text{and} \quad Y_0 \to Y_0X_0.$$

L^+ is generated if the production $Y_0 \to \Lambda$ is replaced by $Y_0 \to X_0$.

For Type 1 the construct is a bit more involved. If G is of Type 1, another new nonterminal Y_1 is added, and Q is obtained as follows: Remove from P the (possible) production $X_0 \to \Lambda$, and add the productions

$$Y_0 \to \Lambda \quad , \quad Y_0 \to X_0 \quad \text{and} \quad Y_0 \to Y_1X_0.$$

Then, for each terminal a, add the productions

$$Y_1a \to Y_1X_0a \quad \text{and} \quad Y_1a \to X_0a.$$

L^+ in turn is generated if the production $Y_0 \to \Lambda$ is omitted (whenever necessary). Note how important it is again for terminals to not appear on left hand sides of productions, to prevent adjacent derivations from interfering with each other. Indeed, a new derivation can only be started when the next one already begins with a terminal.

For Type 0 the construct is quite similar to that for Type 1. □

An additional fairly easily seen closure result is that each family in the Chomsky hierarchy is closed under mirror image of languages.

There are families of languages other than the ones in Chomsky's hierarchy related to it, e.g.

- languages generated by linear grammars, so-called *linear languages* (the family $\mathcal{LIN}$),

- complements of CE languages, so-called *co–CE-languages* (the family co$-\mathcal{CE}$), and

- the intersection of $\mathcal{CE}$ and co$-\mathcal{CE}$, so-called *computable languages* (the family $\mathcal{C}$).

Computable languages are precisely those languages whose membership problem is algorithmically decidable, simply by listing words in the language and its complement in turns, and checking which list will contain the given input word.

It is not necessary to include in the above families of languages the family of languages generated by length-increasing grammars, since it equals $\mathcal{CS}$:

Theorem 11. *For each length-increasing grammar there is a CS-grammar generating the same language.*

Proof. Let us first consider the case where in a length-increasing grammar $G = (\Sigma_\mathrm{N}, \Sigma_\mathrm{T}, X_0, P)$ there is only one length-increasing production $p \to q$ not of the allowed form, i.e., the grammar

$$G' = \big(\Sigma_\mathrm{N}, \Sigma_\mathrm{T}, X_0, P - \{p \to q\}\big)$$

is CS.

By the Lemma above, it may be assumed that there are no terminals in the left hand sides of productions of G. Let us then show how G is transformed to an equivalent CS-grammar $G_1 = (\Delta_\mathrm{N},\ \Sigma_\mathrm{T}, X_0, Q)$. For that we denote

$$p = U_1 \cdots U_m \quad \text{and} \quad q = V_1 \cdots V_n$$

where each U_i and V_j is a nonterminal, and $n \geq m \geq 2$. We take new nonterminals $Z_1, \dots, Z_m$ and let $\Delta_\mathrm{N} = \Sigma_\mathrm{N} \cup \{Z_1, \dots, Z_m\}$. Q then consists of the productions of P, of course excluding $p \to q$, plus new productions taking care of the action of this latter production:

$$\begin{aligned}
\underline{U_1}U_2 \cdots U_m &\to \underline{Z_1}U_2 \cdots U_m, \\
Z_1\underline{U_2}U_3 \cdots U_m &\to Z_1\underline{Z_2}U_3 \cdots U_m, \\
&\ \ \vdots \\
Z_1 \cdots Z_{m-1}\underline{U_m} &\to Z_1 \cdots Z_{m-1}\underline{Z_m V_{m+1} \cdots V_n}, \\
\underline{Z_1}Z_2 \cdots Z_m V_{m+1} \cdots V_n &\to \underline{V_1}Z_2 \cdots Z_m V_{m+1} \cdots V_n, \\
&\ \ \vdots \\
V_1 \cdots V_{m-1}\underline{Z_m}V_{m+1} \cdots V_n &\to V_1 \cdots V_{m-1}\underline{V_m}V_{m+1} \cdots V_n.
\end{aligned}$$

(Here underlining just indicates rewriting.) The resulting grammar G_1 is CS and generates the same language as G. Note how the whole sequence of the new productions should always be applied in the derivation. Indeed, if during this sequence some other productions could be applied, then they could be applied already before the sequence, or after it.

A general length-increasing grammar G is then transformed to an equivalent CS-grammar as follows. We may again restrict ourselves to the case where there are no terminals in the left hand sides of productions. Let us denote by G' the grammar obtained by removing from G all productions not of the allowed form (if any). The removed productions are then added back one by one to G' transforming it each time to an equivalent CS-grammar as described above. The final result is a CS-grammar that generates the same language as G. □

”Computer science is no more
about computers than astronomy
is about telescopes.”

(Attributed to Edsger W. Dijkstra)

Chapter 4

CF-LANGUAGES

4.1 Parsing of Words

We note first that productions of a CF-grammar sharing the same left hand side nonterminal, are customarily written in a joint form. Thus, if the productions having the nonterminal X in the left hand side are

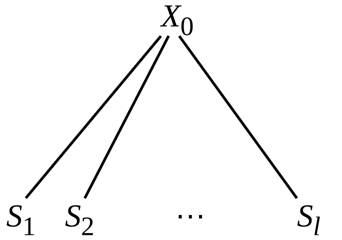
$$X \to w_1 \quad , \dots , \quad X \to w_t ,$$

then these can be written jointly as

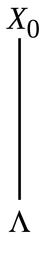
$$X \to w_1 \mid w_2 \mid \cdots \mid w_t .$$

Of course, we should then avoid using the vertical bar | as a symbol of the grammar!

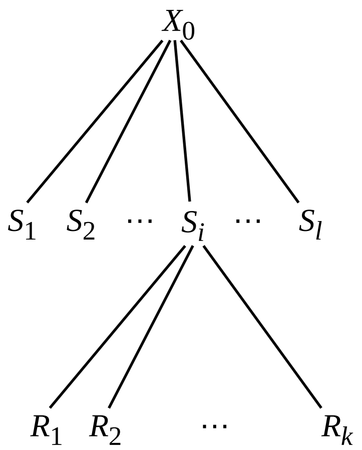
Let us then consider a general CF-grammar $G = (\Sigma_{\mathrm{N}}, \Sigma_{\mathrm{T}}, X_0, P)$, and denote $\Sigma = \Sigma_{\mathrm{N}} \cup \Sigma_{\mathrm{T}}$. To each derivation $X_0 \Rightarrow_G^* w$ a so-called *derivation tree* (or *parse tree*) can always be attached. The vertices of the tree are labelled by symbols in Σ or the empty word Λ. The *root* of the tree is labelled by the axiom X_0. The tree itself is constructed as follows. The starting point is the root vertex. If the first production of the derivation is $X_0 \to S_1 \cdots S_\ell$ where $S_1, \dots, S_\ell \in \Sigma$, then the tree is extended by ℓ vertices labelled from left to right by the symbols $S_1, \dots, S_\ell$:



On the other hand, if the first production is $X_0 \to \Lambda$, then the tree is extended by one vertex labelled by Λ:



Now, if the second production in the derivation is applied to the symbol S_i of the second word, and the production is $S_i \to R_1 \cdots R_k$, then the tree is extended from the corresponding vertex, labelled by S_i, by k vertices, and these are again labelled from left to right by the symbols $R_1, \dots, R_k$ (similarly in the case of $S_i \to \Lambda$):



Construction of the tree is continued in this fashion until the whole derivation is dealt with. Note that the tree can always be extended from any "free" nonterminal, not only those added last. Note also that when a vertex is labelled by a terminal or by Λ, the tree cannot any more be extended from it, such vertices are called *leaves.* The word generated by the derivation can then be read catenating labels of leaves from left to right.

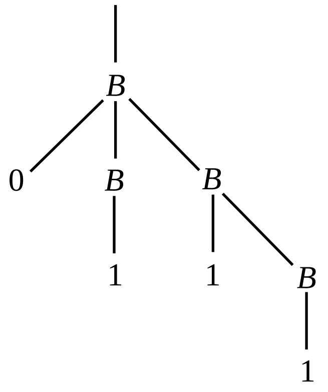
Example. *The derivation*

$$S \Rightarrow B \Rightarrow 0BB \Rightarrow 0B1B \Rightarrow 011B \Rightarrow 0111$$

by the grammar

$$G = \big(\{A, B, S\}, \{0,1\}, S, \{S \to A \mid B, A \to 0 \mid 0A \mid 1AA \mid AA1 \mid A1A, \\ B \to 1 \mid 1B \mid 0BB \mid BB0 \mid B0B\}\big)$$

corresponds to the derivation tree



By the way, this grammar generates exactly all words over $\{0,1\}$ *with nonequal numbers of* 0*'s and* 1*'s.*

Derivation trees call to mind the parsing of sentences, familiar from the grammars of many natural languages, and also the parsing of certain programming languages.

Example. *In the English language a set of simple rules of parsing might be of the form*

$$\begin{aligned}\langle\text{declarative sentence}\rangle &\to \langle\text{subject}\rangle\langle\text{verb}\rangle\langle\text{object}\rangle\\ \langle\text{subject}\rangle &\to \langle\text{proper noun}\rangle\\ \langle\text{proper noun}\rangle &\to \textit{Alice} \mid \textit{Bob}\\ \langle\text{verb}\rangle &\to \textit{reminded}\\ \langle\text{object}\rangle &\to \langle\text{proper noun}\rangle \mid \langle\text{reflexive pronoun}\rangle\\ \langle\text{reflexive pronoun}\rangle &\to \textit{himself} \mid \textit{herself}\end{aligned}$$

where a CF-grammar is immediately identified. The Finnish language is rather more difficult because of inflections, cases, etc.

Example. *In the programming language C a set of simple syntax rules might be*

$$\begin{aligned}\langle\text{statement}\rangle &\to \langle\text{statement}\rangle\langle\text{statement}\rangle \mid \langle\textbf{for}\text{-statement}\rangle \mid \langle\textbf{if}\text{-statement}\rangle \mid \cdots\\ \langle\textbf{for}\text{-statement}\rangle &\to \textbf{for}\ (\ \langle\text{expression}\rangle\ ;\ \langle\text{expression}\rangle\ ;\ \langle\text{expression}\rangle\ )\ \langle\text{statement}\rangle\\ \langle\textbf{if}\text{-statement}\rangle &\to \textbf{if}\ (\ \langle\text{expression}\rangle\ )\ \langle\text{statement}\rangle\\ \langle\text{compound}\rangle &\to \{\ \langle\text{statement}\rangle\ \}\end{aligned}$$

etc., where again the structure of a CF-grammar is identified.

A derivation is a so-called *leftmost derivation* if it is always continued from the leftmost nonterminal. Any derivation can be replaced by a leftmost derivation generating the same word. This should be obvious already by the fact that a derivation tree does not specify the order of application of productions, and a leftmost derivation can always be attached to a derivation tree.

A CF-grammar G is *ambiguous* if some word of $L(G)$ has at least two different leftmost derivations, or equivalently at least two different derivation trees. A CF-grammar that is not ambiguous is *unambiguous.* Grammars corresponding to parsing of sentences of natural languages are typically ambiguous, the exact meaning of the sentence is given by the semantic context. In programming languages ambiguity should be avoided (not always so successfully, it seems).

Ambiguity is more a property of the grammar than that of the language generated. On the other hand, there are CF-languages that cannot be generated by any unambiguous CF-grammar, the so-called *inherently ambiguous languages.*

Example. *The grammar*

$$G = \big(\{S,T,F\},\{a,+,\times,(,)\},S,\{S \to S+T \mid T, T \to T\times F \mid F, F \to (S) \mid a\}\big)$$

generates simple arithmetical formulas. Here a is a "placeholder" for numbers, variables etc. Let us show that G is unambiguous. This is done by induction on the length ℓ of the formula generated.

The basis of the induction is the case $\ell = 1$ which is trivial, since the only way of generating a is

$$S \Rightarrow T \Rightarrow F \Rightarrow a.$$

Let us then make the induction hypothesis, according to which all leftmost derivations of words in $L(G)$ up to the length $p-1$ are unique, and consider a leftmost derivation of a word w of length p in $L(G)$.

Let us take first the case where w has at least one occurrence of the symbol $+$ that is not inside parentheses. Occurrences of $+$ via T and F will be inside parentheses, so that the particular $+$ can only be derived using the initial production $S \to S + T$, where the $+$ is the last occurrence of $+$ in w not inside parentheses. Leftmost derivation of w is then of the form

$$S \Rightarrow S + T \Rightarrow^* u + T \Rightarrow^* u + v = w.$$

Its "subderivations" $S \Rightarrow^* u$ and $T \Rightarrow^* v$ are both leftmost derivations, and thus unique by the induction hypthesis, hence the leftmost derivation of w is also unique. Note that the word v is in the language $L(G)$ and its leftmost derivation $S \Rightarrow T \Rightarrow^* v$ is unique.

The case where there is in w a (last) occurrence of $\times$ not inside parentheses, while all occurrences of $+$ are inside parentheses, is dealt with analogously. The particular $\times$ is then generated via either S or T. The derivation via S starts with $S \Rightarrow T \Rightarrow T \times F$, and the one via T with $T \Rightarrow T \times F$. Again this occurrence of $\times$ is the last one in w not inside parentheses, and its leftmost derivation is of the form

$$S \Rightarrow T \Rightarrow T \times F \Rightarrow^* u \times F \Rightarrow^* u \times v = w,$$

implying, via the induction hypothesis, that w indeed has exactly one leftmost derivation.

Finally there is the case where all occurrences of both $+$ and $\times$ are inside parentheses. The derivation of w must in this case begin with

$$S \Rightarrow T \Rightarrow F \Rightarrow (S),$$

and hence w is of the form (u). Because then u, too, is in $L(G)$, its leftmost derivation is unique by the induction hypothesis, and the same is true for w.

4.2 Normal Forms

The exact form of CF-grammars can be restricted in many ways without reducing the family of languages generated. For instance, as such a general CF-grammar is neither CS nor length-increasing, but it can be replaced by such a CF-grammar:

Theorem 12. *Any CF-language can be generated by a length-increasing CF-grammar.*

Proof. Starting from a CF-grammar $G = (\Sigma_{\mathrm{N}}, \Sigma_{\mathrm{T}}, X_0, P)$ we construct an equivalent length-increasing CF-grammar

$$G' = \big(\Sigma_{\mathrm{N}} \cup \{S\}, \Sigma_{\mathrm{T}}, S, P'\big).$$

If Λ is in $L(G)$, then for S we take the productions $S \to \Lambda \mid X_0$, if not, then only the production $S \to X_0$. To get the other productions we first define recursively the set Δ_Λ of nonterminals of G:

1. If P contains a production $Y \to \Lambda$, then $Y \in \Delta_\Lambda$.
2. If P contains a production $Y \to w$ where $w \in \Delta_\Lambda^+$, then $Y \in \Delta_\Lambda$.
3. A nonterminal is in Δ_Λ only if it is so by items 1. and 2.

Productions of P', other than those for the nonterminal S, are now obtained from productions in P as follows:

(i) Delete all productions of the form $Y \to \Lambda$.

(ii) For each production $Y \to w$, where w contains at least one symbol in Δ_Λ, add in P' all productions obtained from it by deleting in w at least one symbol of Δ_Λ but not all of its symbols.

It should be obvious that now $L(G') \subseteq L(G)$ since, for each derivation of G', symbols of Δ_Λ in the corresponding derivation of G can always be erased if needed. On the other hand, for each derivation of G there is an equivalent derivation of G'. The case of the (possible) derivation of Λ is clear, so let us consider the derivation of the nonempty word v. Again the case is clear if the productions used are all in P'. In the remaining cases we show how a derivation tree $\mathcal{T}$ of the word v for G is transformed to its derivation tree $\mathcal{T}'$ for G'. Now $\mathcal{T}$ must have leaves labelled by Λ. A vertex of $\mathcal{T}$ that only has branches ending in leaves labelled by Λ, is called a Λ-vertex. Starting from some leaf labelled by Λ let us traverse the tree up as far as only Λ-vertices are met. In this way it is not possible to reach the axiom, otherwise the derivation would be that of Λ. We then remove from the tree $\mathcal{T}$ all vertices traversed in this way starting from all leaves labelled by Λ. The remaining tree is a derivation tree $\mathcal{T}'$ for G' of the word v. □

Before proceeding, we point out an immediate consequence of the above theorem and Theorem 11, which is of central importance to Chomsky's hierarchy:

Corollary. $\mathcal{CF} \subseteq \mathcal{CS}$

To continue, we say that a production $X \to Y$ is a *unit production* if Y is a nonterminal. Using a deduction very similar to the one used above we can then prove

Theorem 13. *Any CF-language can be generated by a CF-grammar without unit productions. In addition, it may be assumed that the grammar is length-increasing.*

Proof. Let us just indicate some main points of the proof. We denote by Δ_X the set of all nonterminals ($\neq X$) obtained from the nonterminal X using only unit productions. A grammar $G = (\Sigma_{\mathrm{N}}, \Sigma_{\mathrm{T}}, X_0, P)$ can then be replaced by an equivalent CF-grammar

$$G' = (\Sigma_{\mathrm{N}}, \Sigma_{\mathrm{T}}, X_0, P')$$

without unit productions, where P' is obtained from P in the following way:

1. For each nonterminal X of G find Δ_X.

2. Remove all unit productions.

3. If $Y \in \Delta_X$ and there is in P a production $Y \to w$ (not a unit production), then add the production $X \to w$.

It is apparent that if G is length-increasing, then so is G'. □

A CF-grammar is in *Chomsky's normal form* if its productions are all of the form

$$X \to YZ \quad \text{or} \quad X \to a$$

where X, Y and Z are nonterminals and a is a terminal, the only possible exception being the production $X_0 \to \Lambda$, provided that the axiom X_0 does not appear in the right hand sides of productions.

Transforming a CF-grammar to an equivalent one in Chomsky's normal form is started by transforming it to a length-increasing CF-grammar without unit productions (Theorem 13). Next the grammar is transformed, again keeping it equivalent, to one where the only productions containg terminals are of the form $X \to a$ where a is a terminal, cf. the Lemma in Section 3.2 and its proof. After these operations productions of the grammar are either of the indicated form $X \to a$, or the form

$$X \to Y_1 \cdots Y_k$$

where $Y_1, \ldots, Y_k$ are nonterminals (excepting the possible production $X_0 \to \Lambda$). The latter production $X \to Y_1 \cdots Y_k$ is removed and its action is taken care of by several new productions in normal form:

$$\begin{aligned} X &\to Y_1 Z_1 \\ Z_1 &\to Y_2 Z_2 \\ &\vdots \\ Z_{k-3} &\to Y_{k-2} Z_{k-2} \\ Z_{k-2} &\to Y_{k-1} Y_k \end{aligned}$$

where $Z_1, \ldots, Z_{k-2}$ are new nonterminals to be used only for this production. We thus get

Theorem 14. *Any CF-language can be generated by a CF-grammar in Chomsky's normal form.*

Another classical normal form is *Greibach's normal form.* A CF-grammar is in Greibach's normal form if its productions are of the form

$$X \to aw$$

where a is a terminal and w is either empty or consists only of nonterminals. Again there is the one possible exception, the production $X_0 \to \Lambda$, assuming that the axiom X_0 does not appear in the right hand side of any production. Any CF-grammar can be transformed to Greibach's normal form, too, but proving this is rather more difficult, cf. e.g. the nice presentation of the proof in SIMOVICI & TENNEY.

A grammar in Greibach's normal form resembles a right-linear grammar in that it generates words in leftmost derivations terminal by terminal from left to right. As such a right-linear grammar is however not necessarily in Greibach's normal form.

4.3 Pushdown Automaton

Languages having an infinite index cannot be recognized by finite automata. On the other hand, it is decidedly difficult to deal with an infinite memory structure—indeed, this would lead to a quite different theory—so it is customary to introduce the easier to handle *potentially infinite memory.* In a potentially infinite memory only a certain finite part is in use at any time, the remaining parts containing a constant symbol ("blank"). Depending on how new parts of the memory are brought into use, and exactly how it is used, several types of automata can be defined.

There are CF-languages with an infinite index—e.g. languages of palindromes over nonunary alphabets—so recognition of CF-languages does require automata with infinitely many states. The memory structure is a special one, called *pushdown memory,* and it is of course only potentially infinite. The contents of a pushdown memory may be thought of as a word where only the first symbol can be read and deleted or rewritten, this is called a *stack.* In the beginning the stack contains only one of the specified *initial stack symbols* or *bottom symbols.* In addition to the pushdown memory, the automata also have the "usual" kind of finite memory, used as for Λ-NFA.

Definition. *A* pushdown automaton *(*PDA*) is a septuple* $M = (Q, \Sigma, \Gamma, S, Z, \delta, A)$ *where*

- $Q = \{q_1, \ldots, q_m\}$ *is a finite* set of states, *the elements if which are called* states;
- Σ *is the* input alphabet, *the alphabet of the language;*
- Γ *is the finite* stack alphabet*, i.e., the set of symbols appearing in the stack;*
- $S \subseteq Q$ *is the* set of initial states;
- $Z \subseteq \Gamma$ *is the set of* bottom symbols *of the stack;*
- δ *is the* transition function *which maps each triple* (q_i, a, X)*, where* q_i *is a state,* a *is an input symbol or* Λ *and* X *is a stack symbol, to exactly one finite set* $T = \delta(q_i, a, X)$ *(possibly empty) of pairs* (q, α) *where* q *is a state and* α *is a word over the stack alphabet; cf. the transition function of a* Λ*-NFA;*
- $A \subseteq Q$ *is the* set of terminal states.

In order to define the way a PDA handles its memory structure, we introduce the triples (q_i, x, α) where q_i is a state, x is the unread part (suffix) of the input word and α is the contents of the stack, given as a word with the "topmost" symbol at left. These triples are called *configurations* of M.

It is now not so easy to define and use a "hat function" and a "star function" as was done for Λ-NFA's, because the memory contents is in two parts, the state and the stack. This difficulty is avoided by using the configurations. The configuration (q_j, y, β) is said to be a *direct successor* of the configuration (q_i, x, α), denoted

$$(q_i, x, \alpha) \vdash_M (q_j, y, \beta),$$

if

$$x = ay \quad , \quad \alpha = X\gamma \quad , \quad \beta = \epsilon\gamma \quad \text{and} \quad (q_j, \epsilon) \in \delta(q_i, a, X).$$

Note that here a can be either an input symbol or Λ. We can then define the corresponding "star relation" $\vdash_M^*$ as follows:

1. $(q_i, x, \alpha) \vdash_M^* (q_i, x, \alpha)$
2. If $(q_i, x, \alpha) \vdash_M (q_j, y, \beta)$ then also $(q_i, x, \alpha) \vdash_M^* (q_j, y, \beta)$.
3. If $(q_i, x, \alpha) \vdash_M^* (q_j, y, \beta)$ and $(q_j, y, \beta) \vdash_M (q_k, z, \gamma)$ then also $(q_i, x, \alpha) \vdash_M^* (q_k, z, \gamma)$.
4. $(q_i, x, \alpha) \vdash_M^* (q_j, y, \beta)$ only if this follows from items 1.–3. above.

If $(q_i, x, \alpha) \vdash_M^* (q_j, y, \beta)$, we say that (q_j, y, β) is a *successor* of (q_i, x, α).

A PDA M *accepts*[1] the input word w if

$$(q_i, w, X) \vdash_M^* (q_j, \Lambda, \alpha),$$

for some initial state $q_i \in S$, bottom symbol $X \in Z$, terminal state $q_j \in A$ and stack α. The language $L(M)$ *recognized* by M consists of exactly all words accepted by M.

The pushdown automaton defined above is nondeterministic by nature. In general there will then be multiple choices for the transitions. In particular, it is possible that there is no transition, indicated by an empty value of the transition function or an empty stack, and the automaton halts. Unless the state then is one of the terminal states and the whole input word is read, this means that the input is rejected.

Theorem 15. *Any CF-language can be recognized by a PDA. Moreover, it may be assumed that the PDA then has only three states, an initial state, an intermediate state and a terminal state, and only one bottom symbol.*

Proof. To make matters simpler we assume that the CF-language is generated by a CF-grammar $G = (\Sigma_\mathrm{N}, \Sigma_\mathrm{T}, X_0, P)$ which in Chomsky's normal form.[2] The recognizing PDA is

$$M = \big(\{A, V, T\}, \Sigma_\mathrm{T}, \Sigma_\mathrm{N} \cup \{U\}, \{A\}, \{U\}, \delta, \{T\}\big)$$

where δ is defined by the following rules:

- If $X \to YZ$ is a production of G, then $(V, YZ) \in \delta(V, \Lambda, X)$.
- If $X \to a$ is a production of G such that $a \in \Sigma_\mathrm{T}$ or $a = \Lambda$, then $(V, \Lambda) \in \delta(V, a, X)$.
- The initial transition is given by $\delta(A, \Lambda, U) = \big\{(V, X_0U)\big\}$, and the final transition by $\delta(V, \Lambda, U) = \big\{(T, \Lambda)\big\}$.

The stack symbols are thus the nonterminals of G plus the bottom symbol. Leftmost derivations of G and computations by M correspond exactly to each other: Whenever G, in its leftmost derivation of the word $w = uv$, is rewriting the word $u\alpha$ where $u \in \Sigma_\mathrm{T}^*$ and $\alpha \in \Sigma_\mathrm{N}^+$, the corresponding configuration of M is $(V, v, \alpha U)$. The terminal configuration corresponding to the word w itself is (T, Λ, Λ). □

The converse of this theorem holds true, too. To prove that an auxiliary result is needed to transform a PDA to an equivalent PDA more like the one in the above proof.

Lemma. *Any PDA can be transformed to an equivalent PDA with the property that the stack is empty exactly when the state is terminal.*

Proof. If a PDA $M = (Q, \Sigma, \Gamma, S, Z, \delta, A)$ does not have the required property, some changes in its structure are made. First, a new bottom symbol U is taken, and the new transitions

$$(q_i, XU) \in \delta(q_i, \Lambda, U) \quad (q_i \in S \text{ and } X \in Z)$$

[1] This is the so-called *acceptance by terminal state.* Contents of the stack then does not matter. There is another customary mode of acceptance, *acceptance by empty stack.* An input word w is then accepted if $(q_i, w, X) \vdash_M^* (q_j, \Lambda, \Lambda)$, for some initial state q_i, bottom symbol X and state q_j. No terminal states need to be specified in this mode. It is not at all difficult to see that these two modes of acceptance lead to the same family of recognized languages. Cf. the proof of the Lemma below.

[2] It would in fact be sufficient to assume that if the right hand side of a production of G contains terminals, then there is exactly one of them and it is the first symbol. Starting with a CF-grammar in Greibach's normal would result in a PDA with only two states and no Λ-transitions.

are defined for it. Second, new states V and T are added, and the new transitions

$$(V, X) \in \delta(q_i, \Lambda, X) \quad (q_i \in A \text{ and } X \in \Gamma),$$

$$\delta(V, \Lambda, X) = \{(V, \Lambda)\} \quad (X \in \Gamma)$$

and

$$\delta(V, \Lambda, U) = \{(T, \Lambda)\}$$

are defined. Finally we define the new set of stack symbols to be $\{U\}$ and the new set of terminal states to be $\{T\}$. □

Theorem 16. *For any PDA the language recognized by it is a CF-language.*

Proof. Let us consider a PDA $M = (Q, \Sigma, \Gamma, S, Z, \delta, A)$, and show that the language $L(M)$ is CF. We may assume that M is of the form given by the Lemma above. Thus M accepts an input if and only if its stack is empty after the input is read through. The idea of the construct of the corresponding CF-grammar is to simulate M, and incorporate the state somehow in the leftmost nonterminal of the word being rewritten. The new nonterminals would thus be something like $[X, q_i]$ where $X \in \Gamma$ and $q_i \in Q$. The state can then be updated via the rewriting. The problem with this approach however comes when the topmost stack symbol is erased (replaced by Λ), the state can then not be updated. To remedy this "predicting" the next state q_j is incorporated, too, and the new nonterminals will be triples

$$[q_i, X, q_j]$$

where $X \in \Gamma$ and $q_i, q_j \in Q$. Denote then

$$\Delta = \{[q_i, X, q_j] \mid q_i, q_j \in Q \text{ and } X \in \Gamma\}.$$

Productions of the grammar are given by the following rules where a is either an input symbol or Λ:

- If $(q_j, Y_1 \cdots Y_\ell) \in \delta(q_i, a, X)$ where $\ell \geq 2$ and $Y_1, \dots, Y_\ell \in \Gamma$, then the corresponding productions are

  $$[q_i, X, p_\ell] \to a[q_j, Y_1, p_1][p_1, Y_2, p_2] \cdots [p_{\ell-1}, Y_\ell, p_\ell],$$

  for all choices of $p_1, \dots, p_\ell$ from Q. Note how the third component of a triple always equals the first component of the next triple. Many of these "predicted" states will of course be "misses".

- If $(q_j, Y) \in \delta(q_i, a, X)$, where $Y \in \Gamma$, then the corresponding productions are

  $$[q_i, X, p] \to a[q_j, Y, p],$$

  for all choices of p from Q.

- If $(q_j, \Lambda) \in \delta(q_i, a, X)$, then the corresponding production is

  $$[q_i, X, q_j] \to a.$$

  The topmost stack symbol X can then be erased during the simulation only if the predicted next state q_j is correct, otherwise the leftmost derivation will stop.

- Finally, for the axiom X_0 (assumed not to be in Δ) there are the productions

  $$X_0 \to [q_i, X, q_j]$$

  where $q_i \in S$, $q_j \in A$ and $X \in Z$.

A configuration chain of M accepting the word w (and ending with an empty stack) then corresponds to a leftmost derivation of w by the CF-grammar[3]

$$G = \big(\Delta \cup \{X_0\}, \Sigma, X_0, P\big)$$

where the productions P are given above. Conversely, a leftmost derivation of the word w by G corresponds to a chain of configurations of M accepting w. □

Stack operations of PDA are often restricted. A stack operation, i.e., the stack part of a transition, is of type

- **pop** if it is of the form $(q_j, \Lambda) \in \delta(q_i, a, X)$.
- **push** if it is of the form $(q_j, YX) \in \delta(q_i, a, X)$ where Y is a stack symbol.
- **unit** if it is of the form $(q_j, Y) \in \delta(q_i, a, X)$ where Y is a stack symbol.

Theorem 17. *Any PDA can be replaced by an equivalent PDA where the stack operations are of types* **pop**, **push** *and* **unit**.

Proof. The problematic transitions are of the form

$$(q_j, Y_1 \cdots Y_\ell) \in \delta(q_i, a, X)$$

where $Y_1, \ldots, Y_\ell \in \Gamma$ and $\ell \geq 2$. Other transitions are of the allowed types **pop** or **unit**. To deal with these problematic transitions, certain states of the form $\langle q_j, Y_1 \cdots Y_i \rangle$ are introduced and transitions for these defined. First, the problematic transition is removed and replaced by the transition

$$\big(\langle q_j, Y_1 \cdots Y_{\ell-1} \rangle, Y_\ell\big) \in \delta(q_i, a, X)$$

of type **unit**. Second, the transitions

$$\delta\big(\langle q_j, Y_1 \cdots Y_i \rangle, \Lambda, Y_{i+1}\big) = \big\{\big(\langle q_j, Y_1 \cdots Y_{i-1} \rangle, Y_i Y_{i+1}\big)\big\} \quad (i = 2, \ldots, \ell - 1)$$

of type **push** are added, and finally the transition

$$\delta\big(\langle q_j, Y_1 \rangle, \Lambda, Y_2\big) = \big\{(q_j, Y_1 Y_2)\big\}.$$

One transition is thus replaced by several transitions of the allowed types. □

There is a deterministic variant of the PDA. Four additional conditions are then required to make a PDA a *deterministic pushdown automaton* (*DPDA*):

- The set of initial states contains only one state or is empty.
- There is only one bottom symbol.

[3] If M has no Λ-transitions, it is easy to transform G to Greibach's normal form.

- $\delta(q_i, a, X)$ always contains only one element, or is empty, i.e., there is always at most one possible transition. Here a is an input symbol or Λ.

- If $\delta(q_i, \Lambda, X)$ is not empty, then $\delta(q_i, a, X)$ is empty for all $a \in \Sigma$, that is, if there is a Λ-transition, then there are no other transitions.

Deterministic pushdown automata cannot recognize all CF-languages, the languages recognized by them are called *deterministic CF-languages* (*DCF-languages*). For instance, the language of palindromes over a nonunary alphabet is not a DCF-language. DCF-languages can be generated by unambiguous CF-grammars, this in fact follows by the proof of Theorem 16.

Without its stack a PDA is a lot like a transducer: The symbol read is a pair formed of an input symbol (or Λ) and a stack symbol, and the output is a word replacing the topmost symbol of the stack. Therefore transducers are an important tool in the more advanced theory of CF-languages. (And yes, there are pushdown transducers, too!)

4.4 Parsing Algorithms (A Brief Overview)

What the PDA in the proof of Theorem 15 essentially does is a *top-down parse* of the input word. In other words, it finds a sequence of productions for the derivation of the word generated. Unfortunately though, a PDA is nondeterministic by nature, and a parsing algorithm cannot be that. To get a useful parser this nonterminism should be removed somehow. So, instead of just accepting or rejecting the input word, a PDA should here also "output" sufficient data for the parse.

In many cases the nondeterminism can be removed by *look-ahead,* i.e., by reading more of the input before giving the next step of the parse. A CF-grammar is an LL(k)*-grammar* if in the top-down parsing it suffices to look at the next k symbols to find out the next parse step of the PDA. Formally, an LL(k)-grammar[4] is a CF-grammar satisfying the following look-ahead condition, where $(w)_k$ is the prefix of length k of the word w, and $\Rightarrow_{\text{left}}$ denotes a leftmost direct derivation step: If

$$X_0 \Rightarrow_{\text{left}}^{*} uXv \Rightarrow_{\text{left}} uwv \ \Rightarrow_{\text{left}}^{*} uz \quad \text{and}$$

$$X_0 \Rightarrow_{\text{left}}^{*} uXv' \Rightarrow_{\text{left}} uw'v' \Rightarrow_{\text{left}}^{*} uz'$$

and

$$(z)_k = (z')_k,$$

then

$$w = w'.$$

In the so-called *bottom-up parsing* a word is reduced by replacing an occurrence of the right hand side of a production as a subword by the left hand side nonterminal of the production. Reduction is repeated and data collected for the parse, until the axiom is reached. This type of parsing can also be done using PDA's.

Fast parsing is a much investigated area. A popular and still useful reference is the two-volume book Sippu, S. & Soisalon-Soininen, E.: *Parsing Theory. Volume I: Languages and Parsing.* Springer–Verlag (1988) and *Volume II:* LR(k) *and* LL(k) *Parsing* (1990) by Finnish experts. The classical reference is definitely the "dragon book" Aho, A.V. & Sethi, R. & Ullman, J.D.: *Compilers: Principles, Techniques, and Tools.* Addison–Wesley (1985), the latest edition from 2006 is updated by Monica Lam.

[4] There is also the corresponding concept for rightmost derivations, the so-called LR(k)*-grammar.*

4.5 Pumping

We recall that in sufficiently long words of a regular language one subword can be pumped. Now, there are CF-languages, other than the regular ones, having this property, too. It is not, however, a general property of CF-languages. All CF-languages do have a pumping property, but generally then two subwords must be pumped in synchrony.

The pumping property is easiest to derive starting from a CF-grammar in Chomsky's normal form. This of course in no way restricts the case, since pumping is a property of the language, not of the grammar. The derivation tree of a CF-grammar in Chomsky's normal form is a *binary tree,* i.e., each vertex is extended by at most two new ones. We define the *height* of a (derivation) tree to be the length of the longest path from the root to a leaf.

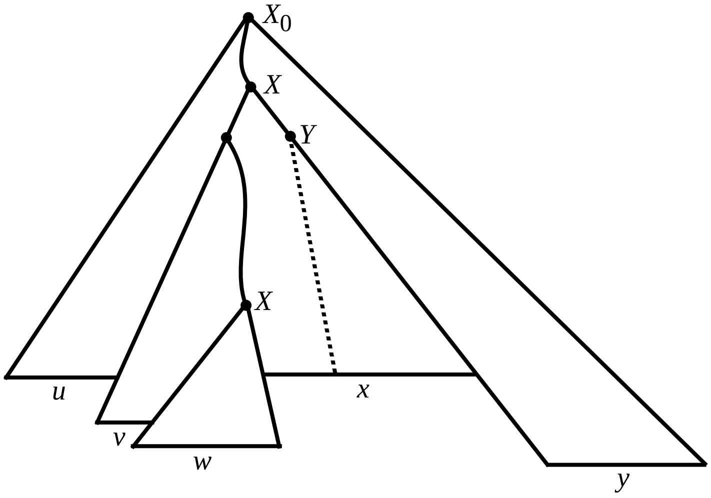
Lemma. *If a binary tree has more than 2^h leaves, then its height is at least $h+1$.*

Proof. This is definitely true when $h = 0$. We proceed by induction on h. According to the induction hypothesis then, the lemma is true when $h \leq \ell$, and the induction statement says that it is true also when $h = \ell + 1 \geq 1$. Whenever the tree has at least two leaves, it may be divided into two binary trees via the first branching, plus a number of preceding vertices (always including the root). At least one of these binary subtrees has more than $2^{\ell+1}/2 = 2^\ell$ leaves and its height is thus at least $\ell + 1$ (by the induction hypothesis). The height of the whole binary tree is then at least $\ell + 2$. □

The basic pumping result is the

Pumping Lemma (”*uvwxy*-Lemma”). *If a CF-language L can be generated by a grammar in Chomsky's normal form having p nonterminals, $z \in L$ and $|z| \geq 2^{p+1}$, then z may be written in the form $z = uvwxy$ where $|vwx| \leq 2^{p+1}$, $vx \neq \Lambda$, $w \neq \Lambda$, and the words uv^nwx^ny are all in L.*

Proof. The height of the derivation tree of the word z is at least $p + 1$ by the Lemma above. Consider then a longest path from the root to a leaf. In addition to the leaf, the path has at least $p + 1$ vertices, and they are labelled by nonterminals. We take the lower $p + 1$ occurrences of such vertices. Since there are only p nonterminals, some nonterminal X appears at least twice as a label. We choose two such occurrences of X. The lower occurrence of X starts a subtree, and its leaves give a word w ($\neq \Lambda$). The upper occurrence of X then starts a subtree the leaves of which give some word vwx, and we can write z in the form $z = uvwxy$. See the schematic picture below.



We may interpret the subtree started from the upper occurrence of X as a (binary) derivation tree of vwx. Its height is then at most $p+1$, and by the Lemma it has at most 2^{p+1} leaves, and hence $|vwx| \leq 2^{p+1}$. The upper occurrence of X has two descendants, one of them is the ancestor of the lower occurrence of X, and the other one is not. The label of the latter vertex is some nonterminal Y. The subtree started from this vertex is the derivation tree of some nonempty subword of v or x, depending on which side of the upper occurrence of X Y is in. So $v \neq \Lambda$ or/and $x \neq \Lambda$.

A leftmost derivation of the word z is of the form

$$X_0 \Rightarrow^* uXy' \Rightarrow^* uvXx'y' \Rightarrow^* uvwxy.$$

We thus conclude that

$$X_0 \Rightarrow^* uXy' \Rightarrow^* uwy,$$

$$X_0 \Rightarrow^* uvXx'y' \Rightarrow^* uv^2Xx'^2y' \Rightarrow^* uv^2wx^2y$$

and so on, are leftmost derivations, too. □

The case where pumping of one subword is possible corresponds to the situation where either $v = \Lambda$ or $x = \Lambda$.

Using pumping it is often easy to show that a language is not CF.

Example. *The language $L = \{a^{2^n} \mid n \geq 0\}$ is a CS-language, as is fairly easy to show (left as an exercise). It is not however CF. To prove this, assume the contrary. Then L can be generated by a grammar in Chomsky's normal form with, say, p nonterminals, and sufficiently long words can be pumped. This is not possible, since otherwise taking $n = p+3$ we can write $2^{p+3} = m_1 + m_2$, where $0 < m_2 \leq 2^{p+1}$, and the word a^{m_1} is in the language L (just choose $m_2 = |vx|$). On the other hand,*

$$m_1 \geq 2^{p+3} - 2^{p+1} > 2^{p+3} - 2^{p+2} = 2^{p+2},$$

and no word of L has length in the interval $2^{p+2}+1, \ldots, 2^{p+3}-1$.

A somewhat stronger pumping result can be proved by strengthening the above proof a bit:

Ogden's Lemma. *If a CF-language L can be generated by a grammar in Chomsky's normal form having p nonterminals, $z \in L$ and $|z| \geq 2^{p+1}$, and at least 2^{p+1} symbols of z are marked, then z can be written in the form $z = uvwxy$ where v and x have together at least one marked symbol, vwx has at most 2^{p+1} marked symbols, w has at least one marked symbol, and the words $uv^n wx^n y$ are all in L.*

4.6 Intersections and Complements of CF-Languages

The family of CF-languages is not closed under intersection. For instance, it is easily seen that the languages

$$L_1 = \{a^i b^i a^j \mid i \geq 0 \text{ and } j \geq 0\} \quad \text{and} \quad L_2 = \{a^i b^j a^j \mid i \geq 0 \text{ and } j \geq 0\}$$

are CF while their intersection

$$L_1 \cap L_2 = \{a^i b^i a^i \mid i \geq 0\}$$

is not (by the Pumping Lemma). (The intersection is CS, though.)

By the rules of set theory

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}.$$

So, since CF-languages are closed under union, it follows that *CF-languages are not closed under complementation.* Indeed, otherwise the languages $\overline{L_1}$ and $\overline{L_2}$ are CF, and so is their union and $L_1 \cap L_2$. On the other hand, it can be shown that *DCF-languages are closed under complementation.*

A state pair construct, very similar to the one in the proof of Theorem 2, proves

Theorem 18. *The intersection of a CF-language and a regular language is CF.*

Proof. From the PDA $M = (Q, \Sigma, \Gamma, S, Z, \delta, A)$ recognizing the CF-language L_1 and the deterministic finite automaton $M' = (Q', \Sigma, q_0', \delta', A')$ recognizing the regular language L_2 a new PDA

$$M'' = (Q'', \Sigma, \Gamma, S'', Z, \delta'', A'')$$

is constructed. We choose

$$\begin{aligned} Q'' &= \{(q_i, q_j') \mid q_i \in Q \text{ ja } q_j' \in Q'\}, \\ S'' &= \{(q_i, q_0') \mid q_i \in S\}, \quad \text{and} \\ A'' &= \{(q_i, q_j') \mid q_i \in A \text{ ja } q_j' \in A'\}, \end{aligned}$$

and define δ'' by the following rules:

- If $(q_j, \alpha) \in \delta(q_i, a, X)$ and $\delta'(q_k', a) = q_\ell'$, then

  $$\big((q_j, q_\ell'), \alpha\big) \in \delta''\big((q_i, q_k'), a, X\big),$$

  i.e., reading the input symbol a results in the correct state transition in both automata.

- If $(q_j, \alpha) \in \delta(q_i, \Lambda, X)$, then

  $$\big((q_j, q_k'), \alpha\big) \in \delta''\big((q_i, q_k'), \Lambda, X\big),$$

  i.e., in a Λ-transition state transition takes place only in the PDA.

This M'' recognizes the intersection $L_1 \cap L_2$, as is immediately verified □

4.7 Decidability Problems. Post's Correspondence Problem

For regular languages just about any characterization problem is algorithmically solvable. This is not any more true for CF-languages. Let us start with some problems that are algorithmically decidable.

- *Membership Problem:* Is a given word w in the CF-language L?

  To solve this problem, the language L is generated by a grammar in Chomsky's normal form. It is then trivial to check whether or not Λ is in the language L. There exist only finitely many possible leftmost derivations of w, since in each derivation step either a new terminal appears or the length is increased by one. These possible derivations are checked.

  Membership and parsing are of course related, if we only think about finding a derivation tree or checking that there is no parse. Many fast methods for this purpose are known, e.g. the so-called *Earley Algorithm.*

- *Emptiness Problem:* Is a given CF-language L empty $(= \emptyset)$?

  Using the Pumping Lemma it is quite easy to see that if L is not empty, then it has a word of length at most $2^{p+1} - 1$.

- *Finiteness Problem:* Is a given CF-language L finite?

  If a CF-language is infinite, it can be pumped. Using the Pumping Lemma it is then easy to see that it has a word of length in the interval $2^{p+1}, \dots, 2^{p+2} - 1$.

- *DCF-equivalence Problem:* Are two given DCF-languages L_1 and L_2 the same?

  This problem was for a long time a very famous open problem. It was solved only comparatively recently by the French mathematician Géraud Sénizergues. This solution by the way is extremely complicated.[5]

There are many algorithmically undecidable problems for CF-languages. Most of these can be reduced more or less easily to the algorithmic undecidability of a single problem, the so-called *Post correspondence problem* (*PCP*). In order to be able to deal with algorithmic unsolvability, the concept of an algorithm must be exactly defined. A prevalent definition uses so-called *Turing machines,* which in turn are closely related to Type 0 grammars.[6] We will return to this topic later.

The input of Post's correspondence problem consists of two alphabets $\Sigma = \{a_1, \dots, a_n\}$ and Δ, and two morphisms $\sigma_1, \sigma_2 : \Sigma^* \to \Delta^*$, cf. Section 2.8. The morphisms are given by listing the images of the symbols of Σ (these are nonempty words[7] over the alphabet Δ):

$$\sigma_1(a_i) = \alpha_i \quad , \quad \sigma_2(a_i) = \beta_i \quad (i = 1, \dots, n).$$

[5] It is presented in the very long journal article Sénizergues, G.: $L(A) = L(B)$? Decidability Results from Complete Formal Systems. *Theoretical Computer Science* **251** (2001), 1–166. A much shorter variant of the solution appears in the article Stirling, C.: Decidability of DPDA Equivalence. *Theoretical Computer Science* **255** (2001), 1–31.

[6] Should there be an algorithm for deciding PCP, then there would also be an algorithm for deciding the halting problem for Turing machines, cf. Theorem 30. This implication is somewhat difficult to prove, see e.g. Martin.

[7] Unlike here, the empty word is sometimes allowed as an image. This is not a significant difference.

The problem to be solved is to find out whether or not there is a word $w \in \Sigma^+$ such that $\sigma_1(w) = \sigma_2(w)$. The output is the answer "yes" or "no". The word w itself is called the *solution* of the problem.

In connection with a given Post's correspondence problem we will need the languages

$$L_1 = \{\sigma_1(w)\#\hat{w} \mid w \in \Sigma^+\} \quad \text{and} \quad L_2 = \{\sigma_2(w)\#\hat{w} \mid w \in \Sigma^+\}$$

over the alphabet $\Sigma \cup \Delta \cup \{\#\}$ where $\#$ is a new symbol. It is fairly simple to check that these languages as well as the complements $\overline{L_1}$ and $\overline{L_2}$ are all CF. Thus, as unions of CF-languages, the languages

$$L_3 = L_1 \cup L_2 \quad \text{and} \quad L_4 = \overline{L_1} \cup \overline{L_2}$$

are CF, too. The given Post's correspondence problem then does not have a solution if and only if the intersection $L_1 \cap L_2$ is empty, i.e., if and only if L_4 consists of all words over $\Sigma \cup \Delta \cup \{\#\}$. The latter fact follows because set-theoretically $L_4 = \overline{L_1 \cap L_2}$.

We may also note that the language L_4 is regular if and only if $L_1 \cap L_2$ is regular, and that this happens if and only if $L_1 \cap L_2 = \emptyset$. Indeed, if $L_1 \cap L_2$ is regular and has a word

$$\sigma_1(w)\#\hat{w} = \sigma_2(w)\#\hat{w},$$

then $w \neq \Lambda$ and $\sigma_1(w) \neq \Lambda$, and the words

$$\sigma_1(w^n)\#\widehat{w^n} = \sigma_2(w^n)\#\widehat{w^n} \quad (n = 2, 3, \dots)$$

will all be in $L_1 \cap L_2$, too. For large enough n the Pumping Lemma of regular languages is applicable, and pumping produces words clearly not in L_1 and L_2.

The following problems are thus algorithmically undecidable by the above:

- *Emptiness of Intersection Problem:* Is the intersection of two given CF-languages empty?

- *Universality Problem:* Does a given CF-language contain all words over its alphabet?

- *Equivalence Problem:* Are two given CF-languages the same?

  This is reduced to the Universality Problem since the language of all words over an alphabet is of course CF.

- *Regularity Problem* Is a given CF-language regular?

With a small additional device the algorithmic undecidability of ambiguity can be proved, too:

- *Ambiguity Problem:* Is a given CF-grammar ambiguous?

  Consider the CF-grammar

  $$G = \big(\{X_0, X_1, X_2\}, \Sigma \cup \Delta \cup \{\#\}, X_0, P\big),$$

  generating the language L_3 above, where P contains the productions $X_0 \to X_1 \mid X_2$ and

  $$X_1 \to \alpha_i X_1 a_i \mid \alpha_i \# a_i \quad \text{and} \quad X_2 \to \beta_i X_2 a_i \mid \beta_i \# a_i \quad (i = 1, \dots, n).$$

  If now $\sigma_1(w) = \sigma_2(w)$ for some word $w \in \Sigma^+$, then the word

  $$\sigma_1(w)\#\hat{w} = \sigma_2(w)\#\hat{w}$$

  of L_3 has two different leftmost derivations by G. If, on the other hand, some word $v\#\hat{w}$ of L_3 has two different leftmost derivations, then one of them must begin with $X_0 \Rightarrow_G X_1$ and the other with $X_0 \Rightarrow_G X_2$, and $v = \sigma_1(w) = \sigma_2(w)$.

Inherent ambiguity of CF-languages is also among the algorithmically undecidable problems. Moreover, not only are CF-languages not closed under intersection and complementation, possible closure is not algorithmic. Indeed, whether or not the intersection of two given CF-languages is CF, and whether or not the complement of a given CF-language is CF, are both algorithmically undecidable problems. Proofs of these undecidabilities are however a lot more difficult than the ones above, see e.g. HOPCROFT & ULLMAN.

”Almost any language one can think of
is context-sensitive.”

(John E. Hopcroft & Jeffrey D. Ullman:
*Introduction to Automata Theory,
Languages, and Computation*)

Chapter 5

CS-LANGUAGES

5.1 Linear-Bounded Automaton

Reading and writing in a pushdown memory takes place on top of the stack, and there is no direct access to other parts of the stack. Ratio of the height of the stack and the length of the input word read in may in principle be arbitrarily large because of Λ-transitions. On the other hand, Λ-transitions can be removed totally.[1] Thus we might assume the height of the stack to be proportionate to the length of the input word.

Changing the philosophy of pushdown automaton by allowing reading and writing ”everywhere in the stack”—writing meaning replacing a symbol by another one—and also allowing reading the input word everywhere all the time, we get essentially an automaton called a linear-bounded automaton (LBA). The amount (or length) of memory used is not allowed to exceed the length of the input. Since it is possible to use more symbols in the memory than in the input alphabet, the information stored in the memory may occasionally be larger than what can be contained by the input, remaining however proportionate to it. In other words, there is a coefficient C such that

$$\text{information in memory} \leq C \times \text{information in input.}$$

This in fact is the basis of the name ”linear-bounded”.

In practise, an LBA is usually defined in a somewhat different way, vis-à-vis the Turing machine.

Definition. *A* linear-bounded automaton (LBA) *is an octuple*

$$M = (Q, \Sigma, \Gamma, S, X_{\mathrm{L}}, X_{\mathrm{R}}, \delta, A)$$

where

- $Q = \{q_1, \dots, q_m\}$ *is the finite* set of states, *the elements are called* states;
- Σ *is the* input alphabet *(alphabet of the language);*
- Γ *is the* tape alphabet ($\Sigma \subseteq \Gamma$);
- $S \subseteq Q$ *is the* set of initial states;
- $X_{\mathrm{L}} \in \Gamma - \Sigma$ *is the* left endmarker;

[1] Cf. footnotes in the proofs of Theorems 15 and 16. Basically, this corresponds to transforming a grammar to Greibach’s normal form.

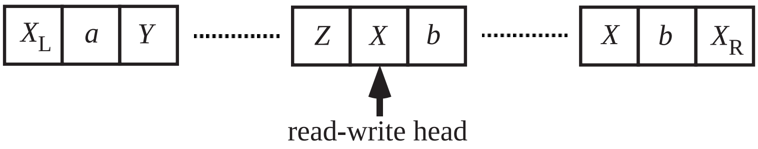
- $X_\mathrm{R} \in \Gamma - \Sigma$ *is the* right endmarker ($X_\mathrm{R} \neq X_\mathrm{L}$);
- δ *is the* transition function *which maps each pair* (q_i, X)*, where* q_i *is a state and* X *is a tape symbol, to a set* $\delta(q_i, X)$ *of triples of the form* (q_j, Y, s) *where* q_j *is a state,* Y *is a tape symbol and* s *is one of the numbers* 0, $+1$ *and* -1*; in addition, it is assumed that*
  - *if* $q_i \in A$*, then* $\delta(q_i, X) = \emptyset$*,*
  - *if* $(q_j, Y, s) \in \delta(q_i, X_\mathrm{L})$*, then* $Y = X_\mathrm{L}$ *and* $s \neq -1$*, and*
  - *if* $(q_j, Y, s) \in \delta(q_i, X_\mathrm{R})$*, then* $Y = X_\mathrm{R}$ *and* $s \neq +1$*;*
- $A \subseteq Q$ *is the* set of terminal states.

The memory of an LBA, the so-called *tape,* may be thought of as a word of the form

$$X_\mathrm{L}\alpha X_\mathrm{R}$$

where $\alpha \in \Gamma^*$. In the beginning the tape is $X_\mathrm{L}wX_\mathrm{R}$ where w is the input word. Moreover, $|\alpha| = |w|$, i.e., the length of the tape is $|w| + 2$ during the whole computation.

At any time an LBA is in exactly one state q_i, *reading* exactly one symbol X in its tape. In the beginning the LBA reads the left endmarker X_L. One may illustrate the situation by imagining that an LBA has a "read-write head" reading the tape, and writing on it:



The transition function δ then tells the next move of the LBA. If

$$(q_j, Y, s) \in \delta(q_i, X)$$

and the tape symbol X under scan is the ℓ^th symbol, then the LBA changes its state from q_i to q_j, replaces the tape symbol X it reads by the symbol Y (Y may be X), and moves on to read the $(\ell + s)^\mathrm{th}$ symbol in the tape. This operation amounts to one *transition* of the LBA. Note that an LBA is nondeterministic by nature: it may be possible to choose a transition from several available ones, or that there is no transition available.

As for the PDA, memory contents are given using configurations. A *configuration* is a quadruple (q_i, α, X, β) where q_i is a state, $\alpha X\beta$ is the tape and X is the tape symbol under scan. Above then $\ell = |\alpha X|$. The *length* of the configuration (q_i, α, X, β) is $|\alpha X\beta|$. A configuration $(q_j, \alpha', Y, \beta')$ is the *direct successor* of the configuration (q_i, α, X, β) if it is obtained from (q_i, α, X, β) via one transition of the LBA M, this is denoted by

$$(q_i, \alpha, X, \beta) \vdash_M (q_j, \alpha', Y, \beta').$$

Thus, corresponding to the transition $(q_j, Y, -1) \in \delta(q_i, X)$, we get e.g.

$$(q_i, \alpha Z, X, \beta) \vdash_M (q_j, \alpha, Z, Y\beta)$$

where $Z \in \Gamma$. The "star relation" $\vdash_M^*$ is defined exactly as it was for the PDA.

In the beginning the tape contains the input word w (plus the endmarkers), and the corresponding initial configuration is $(q_i, \Lambda, X_{\mathrm{L}}, wX_{\mathrm{R}})$ where q_i is an initial state. The LBA M *accepts* the input w if

$$(q_i, \Lambda, X_{\mathrm{L}}, wX_{\mathrm{R}}) \vdash_M^* (q_j, \alpha, Y, \beta)$$

where q_i is an initial state and q_j is a terminal state. Note that there is no transition from a terminal state, i.e., the LBA *halts.* An LBA may also halt in a nonterminal state, the input is then rejected. The language of all words accepted by an LBA M is the language *recognized* by it, denoted by $L(M)$.

Since the complete definition of even a fairly simple LBA may be quite long, it is customary to just describe the basic idea of working of the LBA, yet with sufficient detail to leave no doubt about the correctness of the working.[2]

Example. *The language* $L = \{a^{2^n} \mid n \geq 0\}$ *is recognized by the LBA*

$$M = \big(Q, \{a\}, \Gamma, \{q_0\}, @, \#, \delta, \{q_1\}\big),$$

the working of which may be described briefly as follows. In addition to the symbol a*,* Γ *contains also the symbol* a'*.* M *first changes the leftmost occurrence of* a *to* a'*. After that* M *repeatedly doubles the number of occurrences of the symbol* a'*, using the previous occurrences of* a' *as a counter. If, during such doubling process,* M *hits the right endmarker #, it halts in a nonterminal state* ($\neq q_1$)*. If, on the other hand, it hits # after just finishing a doubling round, it moves to the terminal state* q_1*. Obviously, many more tape symbols and states will be needed to implement such a working.*

Theorem 19. *Each CS-language can be recognized by an LBA.*

Proof. We take an arbitrary CS-grammar $G = (\Sigma_{\mathrm{N}}, \Sigma_{\mathrm{T}}, X_0, P)$, and show how the language it generates is recognized by an LBA

$$M = (Q, \Sigma_{\mathrm{T}}, \Gamma, S, X_{\mathrm{L}}, X_{\mathrm{R}}, \delta, A).$$

In addition to the terminals of G, Γ also contains the symbols

$$[x, a] \quad (x \in \Sigma_{\mathrm{N}} \cup \Sigma_{\mathrm{T}} \text{ and } a \in \Sigma_{\mathrm{T}}),$$

and the symbols

$$[\Lambda, a] \quad (a \in \Sigma_{\mathrm{T}}).$$

M first changes each symbol a of the input to the symbol $[\Lambda, a]$, except for the leftmost symbol b of the input, which it changes to $[X_0, b]$. In this way M stores the input in the second ("lower") components of the tape symbols, and uses the first ("higher") components to simulate the derivation by G. (Often these "upper" and "lower" parts of the tape are called "tracks", and there may be more than two of them.) One simulation step goes as follows:

1. M traverses the tape from left to right, using its finite state memory to search for a subword equal to the left hand side of some production of G. For this, a number of last read tape symbols remain stored in the state memory of M.

[2] "Designing Turing machines by writing out a complete set of states and a next-move function is a noticeably unrewarding task." (HOPCROFT & ULLMAN).

2. After finding in this way a subword equal to left hand side of a production, M decides whether or not it will use this production in the simulation (nondeterminism). If not, M continues its search.

3. If M decides to use in the simulation the production it found, it next conducts the simulation of the corresponding direct derivation by G. If this leads to lengthening of the word, i.e., the right hand side of the production is longer than the left hand side, a suffix of the word already derived must be moved to the right respectively, if possible. This takes a lot of transitions. If there is not sufficient space available, that is, the derived word is too long and there would be an overflow, M halts in a nonterminal state. After simulating the direct derivation step, M again starts its search from the left endmarker.

4. If there is no way of continuing the simulated derivation, that is, no applicable production is found, M checks whether or not the derived word equals the input, stored in the ”lower track” for this purpose, and in the positive case halts in a terminal state. In the negative case, M halts in a nonterminal state.

In addition to the given tape symbols, many more are clearly needed as markers etc., and also a lot of states. □

Theorem 20. *The language recognized by an LBA is CS.*

Proof. Consider an arbitrary LBA

$$M = (Q, \Sigma, \Gamma, S, \$, \#, \delta, A).$$

The length-increasing grammar $G = (\Sigma_\mathrm{N}, \Sigma, X_0, P)$ generating $L(M)$, is obtained as followsi (cf. Theorem 11). First the nonterminals

$$[X, a] \quad (\text{for } X \in \Gamma \text{ and } a \in \Sigma),$$

are needed, as well as the endmarkers

$$[\$, X, a] \quad \text{and} \quad [\#, X, a] \quad (\text{for } X \in \Gamma \text{ and } a \in \Sigma).$$

To simulate a state transition, more nonterminals are needed:

$$[X, q_i, a] \quad \text{and} \quad [X, q_i, s, a] \quad (\text{for } X \in \Gamma, q_i \in Q, a \in \Sigma \text{ and } s = 0, \pm 1),$$

as well as the corresponding endmarkers

$$\begin{aligned} [\$, q_i, X, a] \quad &\text{and} \quad [\#, q_i, X, a], \\ [\$, X, q_i, a] \quad &\text{and} \quad [\#, X, q_i, a], \\ [\$, X, q_i, s, a] \quad &\text{and} \quad [\#, X, q_i, s, a] \quad (\text{for } X \in \Gamma, q_i \in Q, a \in \Sigma \text{ and } s = 0, \pm 1). \end{aligned}$$

Finally, one nonterminal Y is still needed.

G generates an arbitrary input word of length at least 3, stored in the last component of the nonterminals. (Words shorter than that are taken care of separately.) For this the productions

$$\begin{aligned} &X_0 \to Y[\#, a, a] \quad (\text{for } a \in \Sigma) \text{ and} \\ &Y \to Y[a, a] \mid [\$, q_i, a, a][b, b] \quad (\text{for } a, b \in \Sigma \text{ and } q_i \in S) \end{aligned}$$

are needed. After that G simulates the working of M using the first components of the nonterminals. Note how the endmarkers of M must be placed inside the endmost symbols because a length-increasing grammar cannot erase them.

A transition

$$(q_j, V, 0) \in \delta(q_i, U),$$

where the read-write head does not move, is simulated using the productions

$$\begin{aligned}
[U, q_i, a] &\to [V, q_j, a], \\
[\$, q_i, X, a] &\to [\$, q_j, X, a] \quad (\text{for } U = V = \$), \\
[\$, U, q_i, a] &\to [\$, V, q_j, a], \\
[\#, q_i, X, a] &\to [\#, q_j, X, a] \quad (\text{for } U = V = \#) \text{ and} \\
[\#, U, q_i, a] &\to [\#, V, q_j, a].
\end{aligned}$$

A transition

$$(q_j, V, +1) \in \delta(q_i, U),$$

where the read-write head moves to the right, is in turn simulated by the productions

$$\begin{aligned}
[U, q_i, a] &\to [V, q_j, +1, a] \quad \text{and} \\
[\$, U, q_i, a] &\to [\$, V, q_j, +1, a],
\end{aligned}$$

”declaring” that a move to the right is coming, and the productions

$$\begin{aligned}
[V, q_j, +1, a][X, b] &\to [V, a][X, q_j, b], \\
[V, q_j, +1, a][\#, X, b] &\to [V, a][\#, X, q_j, b], \\
[\#, U, q_i, a] &\to [\#, q_j, V, a], \\
[\$, q_i, X, a] &\to [\$, X, q_j, a] \quad (\text{for } U = V = \$) \text{ and} \\
[\$, V, q_j, +1, a][X, b] &\to [\$, V, a][X, q_j, b],
\end{aligned}$$

taking care of the move itself. A transition, where the read-write head moves to the left, is simulated by analogous productions (left to the reader).

Finally the terminating productions

$$\begin{aligned}
&[X, q_i, a] \to a\ , \\
&[\$, q_i, X, a] \to a\ , \qquad [\$, X, q_i, a] \to a\ , \\
&[\#, q_i, X, a] \to a\ , \qquad [\#, X, q_i, a] \to a\ ,
\end{aligned}$$

where $q_i \in A$, and

$$[X, a] \to a\ , \quad [\$, X, a] \to a\ , \quad [\#, X, a] \to a$$

are needed.

A word w in $L(M)$ of length ≤ 2, is included using the production $X_0 \to w$. □

As a consequence of Theorems 19 and 20, *CS-languages are exactly all languages recognized by linear-bounded automata.*

An LBA is *deterministic* if $\delta(q_i, X)$ always contains at most one element, i.e., either there is no available transition or then there is only one, and there is at most one initial state. Languages recognized by deterministic LBA's are called *deterministic CS-languages* or *DCS-languages.* It is a long-standing and famous open problem, whether or not all CS-languages are deterministic.

5.2 Normal Forms

A length-increasing grammar is in *Kuroda's normal form* if its productions are of the form

$$X \to a\ , \quad X \to Y\ , \quad X \to YZ \quad \text{and} \quad XY \to UV$$

where a is a terminal and X, Y, Z, U and V are nonterminals. Again there is the exception, the production $X_0 \to \Lambda$, assuming the axiom X_0 does not appear in the right hand side of any production.

Theorem 21. *Each CS-language can be generated by a length-increasing grammar in Kuroda's normal form.*

Proof. The CS-language is first recognized by an LBA, and then the LBA is transformed to a length-increasing grammar as in the proof of Theorem 20. The result is a grammar in Kuroda's normal form, as is easily verified. □

A CS-grammar is in *Penttonen's normal form*[3] if its productions are of the form

$$X \to a\ , \quad X \to YZ \quad \text{and} \quad XY \to XZ$$

where a is a terminal and X, Y and Z are nonterminals. (With the above mentioned exception here, too, of course.) Each CS-grammar can be transformed to Penttonen's normal form, the proof of this is, however, quite difficult.

5.3 Properties of CS-Languages

CS-languages are computable, that is, their membership problem is algorithmically decidable. This can be seen fairly easily: To find out whether or not a given word is in the language recognized by the LBA M, just see through the steps taken by M with input w, until either M halts or there is a repeating configuration. This should be done for all alternative transitions (nondeterminism). On the other hand,

Theorem 22. *There is a computable language over the unary alphabet $\{a\}$ that is not CS.*

Proof. Let us enumerate all possible CS-grammars with terminal alphabet $\{a\}$: $G_1, G_2, \ldots$ This can be done e.g. as follows. First replace the i^{th} nonterminal of the grammar everywhere by $\#b_i$ where b_i is the binary representation of i. After this, the symbols used are

$$a\ 0\ 1\ ,\ \#\ (\ )\ \{\ \}\ \to$$

and grammars can be ordered lexicographically, first according to length and then within each length alphabetically.

Consider then the language

$$L = \big\{a^n \bigm| a^n \notin L(G_n)\big\}$$

[3] This is the so-called left normal form. There naturally is also the corresponding right normal form. The original article reference is Penttonen, M.: One-Sided and Two-Sided Context in Formal Grammars. *Information and Control* **25** (1974), 371–392. Prof. Martti Penttonen is a well-known Finnish computer scientist.

over the alphabet $\{a\}$. L is computable because checking membership of the word a^m in L can be done as follows: (1) Find the grammar G_m. (2) Since membership is algorithmically decidable for G_m, continue by checking whether or not a^m is in the language $L(G_m)$.

On the other hand L is not CS. Otherwise it would be one of the languages $L(G_1)$, $L(G_2), \ldots$, say $L = L(G_k)$. But then the word a^k is in L if and only if it is not in L! □

The proof above is another example of the diagonal method, cf. the proof of Theorem 1. As a matter of fact, it is difficult to find "natural" examples of non-CS languages, without using the diagonal method or methods of computational complexity theory. Many (most?) programming languages have features which in fact imply that they really are not CF-languages. Still, just about every one of them is CS.

In Theorem 10 it was stated, among other things, that CS-languages are closed under union, concatenation, concatenation closure, and mirror image. Unlike CF-languages, CS-languages are closed under intersection and complementation, too.

Theorem 23. *$\mathcal{CS}$ is closed under intersection.*

Proof. Starting from LBA's M_1 and M_2 recognizing the CS-languages L_1 and L_2, respectively, it is not difficult to construct a third LBA M recognizing the intersection $L_1 \cap L_2$. M will then simulate both M_1 and M_2 simultaneously in turns, storing the tapes, positions of read-write heads and states of M_1 and M_2 in its tape (using tracks and special tape symbols). M accepts the input word if in this simulation both M_1 and M_2 accept it. □

Closure of $\mathcal{CS}$ under complementation was a celebrated result in discrete mathematics in the 1980's. After being open for a long time, it was proved at exactly the same time but independently by Neil Immerman and Róbert Szelepcsényi.[4] In fact they both proved a rather more general result in space complexity theory.

Immerman–Szelepcsényi Theorem. *$\mathcal{CS}$ is closed under complementation.*

Proof. Let us take an LBA M, and construct another LBA M_C recognizing $\overline{L(M)}$.

After receiving the input w, the first thing M_C does is to count the number N of those configurations of M which are successors of an initial configuration corresponding to w. The length of these configurations is $|w| + 2 = n$. Let us denote by N_j the number of all configurations that can be reached from an initial configuration corresponding to the input w by at most j transitions. When, for such a j, the first value $j = j_{\max}$ is found such that $N_{j_{\max}} = N_{j_{\max}+1}$, then obviously $N = N_{j_{\max}}$. The starting N_0 is the number of initial states of M. As is easily verified, the number N can be stored in the tape using, say, a decimal expansion with several decimals encoded in one tape symbol, if necessary. Similarly, several numbers of the same size can be stored in the tape. In what follows, an initial configuration is always one corresponding to the input w.

After getting the number N_j, M_C computes the next number N_{j+1}. For that M_C needs to store only the number N_j, but not the preceding numbers. M_C goes through all configurations of M of length n in an alphabetical order; for this it needs to remember only the current configuration and its alphabetical predecessor, if needed, but no others. After finishing the investigation of the configuration κ_ℓ, M_C uses it to find the alphabetically

[4] The original article references are Immerman, N.: Nondeterministic Space is Closed under Complementation. *SIAM Journal of Computing* **17** (1988), 275–303 and Szelepcsényi, R.: The Method of Forced Enumeration for Nondeterministic Automata. *Acta Informatica* **26** (1988), 279–284.

succeeding configuration $\kappa = \kappa_{\ell+1}$. The aim is to find out, whether the configuration κ could be reached from an initial configuration by at most $j+1$ transitions.

M_{C} maintains in its tape two counters λ_1 and λ_2. Each time a configuration κ is found that can be reached by at most $j+1$ transitions from an initial configuration, the counter λ_1 is increased by one.

To investigate a configuration κ, M_{C} searches through configurations κ' of length n for configurations that can be reached from an initial configuration by at most j transitions. M_{C} retains the number of such configurations in the counter λ_2.

For each configuration κ', M_{C} first guesses nondeterministically whether or not this configuration is among those N_j configurations that can be reached by at most j transitions from some initial configurations. If the guess is "no", then M_{C} moves on to the next configuration to be investigated without changing the counter λ_2. If, on the other hand, the guess is "yes", M_{C} continues by guessing a chain of configurations leading to the configuration κ' from some initial configuration by at most j steps. This guessing is done nondeterministically step by step. In case the configurations κ' are all checked and $\lambda_2 < N_j$, then M_{C} halts in a nonterminal state without accepting w.

M_{C} checks whether or not the guessed chain of configurations ending in κ' is correct, i.e., consists of successive configurations. There are two alternatives.

(1) If the chain of configurations is correct, M_{C} checks whether $\kappa = \kappa'$ or whether $\kappa' \vdash_M \kappa$, and in the affirmative case increases the counter λ_1 by one, and moves on to investigate the configuration $\kappa_{\ell+2}$. If the counter λ_2 hits its maximum value N_j and a searched for configuration is not found, M_{C} concludes that the configuration κ cannot be reached by at most $j+1$ transitions from an initial configuration, and moves on to investigate the configuration $\kappa_{\ell+2}$ without changing the counter λ_1.

(2) If the guessed chain of configurations is not correct, then M_{C} halts in a nonterminal state without accepting the input w.

Finally M_{C} has checked all possible configurations κ_ℓ, and found all numbers N_j, and thus also the number N. Note that always, when moving on to investigate a new configuration κ_ℓ, the LBA M_{C} resets the counter λ_2, and always, when starting to compute the next number N_j, it resets the counter λ_1.

The second stage of the working of M_{C} is to again go through all configurations of length n maintaining a counter λ. For each configuration κ, M_{C} first guesses whether or not it is a successor of an initial configuration. If the guess is "no", M_{C} moves on to the next configuration in the list without touching the counter λ. If, on the other hand, the guess is "yes", then M_{C} continues by guessing a chain of configurations leading to the configuration κ from some initial configuration in at most $j_{\max}$ steps. There are now several possible cases:

1. If the guessed chain of configurations is not correct (see above), or it is correct but κ is an accepting terminal configuration of M, then M_{C} halts in a nonterminal state without accepting the input w.

2. If the guessed chain of configurations is correct and κ is not an accepting terminal configuration of M and λ has not reached its maximum value N, then M_{C} increases the counter λ by one and moves on the next configuration in the list.

3. If the guessed chain of configurations is correct and κ is not an accepting terminal configuration of M and the counter λ hits its maximum value N, then M_{C} accepts the input w by making a transition to a terminal state. □

Nearly every characterization problem is algorithmically undecidable for CS-languages. Anyway, as noted, the membership problem is algorithmically decidable for them. In particular, the *emptiness, finiteness, universality, equivalence and regularity problems are all undecidable.* Universality, equivalence and regularity were undecidable already for CF-languages, and hence for CS-languages, too, and since CS-languages are closed under complementation, emptiness is undecidable as well.

To show algorithmic undecidability of finiteness, an additional device is needed, say, via the Post correspondence problem introduced in Section 4.7. For each pair of PCP input morphisms

$$\sigma_1 : \Sigma^* \to \Delta^* \quad \text{and} \quad \sigma_2 : \Sigma^* \to \Delta^*$$

an LBA M with input alphabet $\{a\}$ is constructed, which, after receiving the input a^n, checks whether or not there is a word $w \in \Sigma^+$ such that $|w| \leq n$ and $\sigma_1(w) = \sigma_2(w)$. (Clearly this is possible.) If such a word w exists, then M rejects the input a^n, otherwise it accepts a^n. Thus $L(M)$ is finite if and only if there exists a word $w \in \Sigma^+$ such that $\sigma_1(w) = \sigma_2(w)$.

Using Post's correspondence problem it is possible to show undecidability of emptiness "directly", without using the hard-to-prove Immerman–Szelepcsényi Theorem. Indeed, it is easy to see that

$$\{w \mid w \in \Sigma^+ \text{ and } \sigma_1(w) = \sigma_2(w)\}$$

is a CS-language.

”No, I don’t have a solution,
but I certainly admire the problem.”

(Ashleigh Brilliant)

Chapter 6

CE-LANGUAGES

6.1 Turing Machine

A *Turing machine (TM)* is like an LBA except that there are no endmarkers, and the tape is potentially infinite in both directions. In the tape alphabet there is a special symbol B, the so-called *blank.* At each time, only finitely many tape symbols are different from the blank. In the beginning the tape contains the input word, flanked by blanks, and the read-write head reads the first symbol of the input. In case the input is empty, the read-write head reads one of the blanks.

Formally a Turing machine is a septuple

$$M = (Q, \Sigma, \Gamma, S, B, \delta, A)$$

where Q, Σ, Γ, S, δ and A as for the LBA, and B is the blank. A transition is also defined as for the LBA, there are no restrictions for the movement of the read-write head since there are no endmarkers. The only additional condition is that if $(q_j, Y, s) \in \delta(q_i, X)$, then $Y \neq B$. The Turing machine defined in this way is nondeterministic by nature. The *deterministic Turing Machine* (*DTM*) is defined as the deterministic LBA.

Definition of a configuration for a Turing machine M is a bit different from that for the LBA. If the read-write head is not reading a blank, then the corresponding *configuration* is a quadruple (q_i, α, X, β) where q_i is a state, $\alpha X \beta$ is the longest subword in the tape not containing any blanks, and X is the tape symbol read by M in this subword. If, on the other hand, the read-write head is reading a blank, then the corresponding configuration is (q_i, Λ, B, β) (resp. $(q_i, \alpha, B, \Lambda)$) where β (resp. α) is the longest subword in the tape not containing blanks. In particular, if the tape contains only blanks, i.e., the tape is empty, then the corresponding configuration is $(q_i, \Lambda, B, \Lambda)$.

Acceptance of an input word and recognition of a language are defined as for the LBA.

Theorem 24. *Each CE-language can be recognized by a Turing machine.*

Proof. The proof is very similar to that of Theorem 19. The only real difference is that a Type 0 grammar can erase symbols, and a word may be shortened when rewritten. The corresponding suffix of the derived word must then be moved to the left. □

Theorem 25. *Languages recognized by Turing machines are all CE.*

Proof. The proof is more or less as that of Theorem 20. Let us just mention a few details. There are two blanks used, one in each end of the word to be rewritten. These will be erased in the end. Symbols containing the stored input will be terminated in the end, while all other symbols will be erased. This process begins only when the TM being simulated is in a terminal state, and proceeds ”as a wave” in both directions. □

Languages recognized by Turing machines are thus exactly all CE-languages.

Thinking of recognizing languages, we could as well restrict ourselves to deterministic Turing machines. Indeed, a Turing machine M can always be simulated by a DTM M' recognizing the same language. In the i^{th} stage of its working, the simulating DTM M' has in its tape somehow encoded all those configurations M that can be reached from an initial configuration corresponding to the input w in at most i transitions, one after the other. In the 0^{th} stage (beginning) M' computes in its tape all initial configurations of M corresponding to the input w, as many as there are initial states. When moving from stage i to stage $i+1$ the DTM M' simply goes through all last computed configurations of M continuing each of them in all possible ways (nondeterminism) by one transition of M, and stores the results in its tape. M' halts when it finds a terminal configuration of M.

It was already noted that the family $\mathcal{CE}$ is closed under union, concatenation, concatenation closure, and mirror image. In exactly the same way as for Theorem 23, we can prove

Theorem 26. *$\mathcal{CE}$ is closed under intersection.*

Despite the fact that all languages in $\mathcal{CE}$ can be recognized by deterministic Turing machines, $\mathcal{CE}$ is not closed under complementation. The reason for this is that a DTM may take infinitely many steps without halting at all. This is, by the way, true for the LBA, too, but can then be easily prevented by a straightforward use of counters (left as an exercise for the reader). For a more detailed treatment of the matter, the so-called universal Turing machine is introduced. It will suffice to restrict ourselves to Turing machines with input alphabet $\{0,1\}$.[1] Such Turing machines can be encoded as binary numbers. This can be done e.g. by first presenting δ as a set P of quintuples:

$$(q_j, Y, s, q_i, X) \in P$$

if and only if $(q_j, Y, s) \in \delta(q_i, X)$. After that the symbols used are encoded in binary: The symbols

$$(\ )\ \{\ \}\ ,\ 0\ 1\ +\ -$$

are encoded, in this order, as the binary words 10^i $(i = 1, \dots, 9)$. The i^{th} state is then encoded as the binary word 110^i, and the j^{th} tape symbol as the word 1110^j. After these encodings a Turing machine M can be given as a binary number $\beta(M)$.

A *universal Turing machine* (*UTM*) is a Turing machine U which, after receiving an input[2] $w\$\beta(M)$, where $w \in \{0,1\}^*$, simulates the Turing machine M on input w. In particular, U accepts its input if M accepts the input w. All other inputs are rejected by U.

Theorem 27. *There exists a universal Turing machine U.*

Proof. Without going into any further details, the idea of working of U is the following. First U checks that the input is of the correct form $w\$\beta(M)$. In the positive case U "decodes" $\beta(M)$ to be able to use M's transition function δ. After that U simulates M on input w transition by transition, fetching in between the transition rules it needs from the decoded presentation of δ. When M enters a terminal state, then so does U. □

[1] This alphabet could in fact be of any kind whatsoever, even unary!

[2] The language formed of exactly all inputs of this form can be recognized by an LBA, as one can see fairly easily.

Theorem 28. *$\mathcal{CE}$ is not closed under complementation.*

Proof. The ”diagonal language”

$$D = \{w \mid w\$w \in L(U)\}$$

is CE, as is seen quite easily—just make a copy of the w and then simulate U. D then contains those encoded Turing machines that ”accept themselves”. On the other hand, the complement $\overline{D}$ is not CE. In the opposite case it would be recognized by a Turing machine M, and the word $\beta(M)$ would be in the language D if and only if it is not! □

The above proof again uses the diagonal method, cf. the proofs of Theorems 1 and 22. The language $\overline{D}$ in the proof is an example of a language with a definitely finite description, and yet not CE.

$\overline{D}$ is a co–CE-language that is not CE. Correspondingly, D is CE-language that is not co–CE. The intersection of the families $\mathcal{CE}$ and co–$\mathcal{CE}$ is $\mathcal{C}$ (computable languages), and it is thus properly included in both families.

6.2 Algorithmic Solvability

A deterministic Turing machine can be equipped with *output.* When the machine enters a terminal state, the output is in the tape properly indicated and separated from the rest of the tape contents. Note that occasionally there might be no output, the machine may halt in a nonterminal state or not halt at all.

According to the prevailing conception, *algorithms* are methods that can be realized using Turing machines, and only those. This is generally known as the *Church–Turing thesis.* Problems that can be solved algorithmically are thus exactly those problems that can be solved by Turing machines, at least in principle if not in practise.

One consequence of the Church–Turing thesis is that there clearly are finitely defined problems that cannot be solved algorithmically.

Theorem 29. *There exists a CE-language whose membership problem is not algorithmically decidable. Recall that the algorithm should then always output the answer yes/no depending on whether or not the input word is in the language.*

Proof. The diagonal language D in the proof of Theorem 28 is a CE-language whose membership problem is not algorithmically decidable. Indeed, otherwise the membership problem could be decided for the language $\overline{D}$ by a deterministic Turing machine, too, and $\overline{D}$ would thus be CE. □

The *Halting problem* for a deterministic Turing machine M asks whether or not M halts after receiving an input word w (also the input of the problem).

Theorem 30. *There is a Turing machine whose Halting problem is algorithmically undecidable.*

Proof. It is easy to transform the Turing machine to one halting only when in a terminal state, by adding, if needed, a behavior which leaves the machine in an infinite loop when the original machine is about to halt in a nonterminal state. The theorem now follows from the one above. □

Just about all characterization problems for CE-languages are algorithmically undecidable. Indeed, every nontrivial property is undecidable. Here, a property is called nontrivial if some CE-languages have it but not all of them.

Rice's Theorem. *If O is a nontrivial property of CE-languages, then it is algorithmically undecidable. The input here is a CE-language, given via a Turing machine recognizing it.*

Proof. The empty language $\emptyset$ either has the property O or it does not. By interchanging, if needed, the property O with its negation, we may assume that that the empty language does not have the property O. Since the property O is nontrivial, there is a CE-language L_1 that has O, and a Turing machine M_1 recognizing L_1.

Let us then fix a CE-language L with an algorithmically undecidable membership problem, and a Turing machine M recognizing L.

For each word w over the alphabet of the language L, we define a Turing machine M_w as follows. The input alphabet of M_w is the same as that of M_1. After receiving an input v, M_w starts by trying to find out whether w is in the language L by simulating the Turing machine M. In the affirmative case M_w continues by simulating the Turing machine M_1 on input v, which it carefully stored for this purpose, and accepts the input if and only if M_1 does. In the negative case M_w does not accept anything. Note especially that even if the simulation of M does not halt, the result is correct! Note also that even though no Turing machine can find out whether or not M halts on input w, the Turing machine M_w can still be algorithmically constructed.

With this a word w is transformed to a Turing machine M_w with the following property: The language $L(M_w)$ has the property O if and only if $w \in L$. Since the latter membership is algorithmically undecidable, then so is the property O. □

As an immediate consequence of Rice's Theorem, the following problems are algorithmically undecidable for CE-languages:

- Emptiness problem
- Finiteness problem
- Universality problem
- Regularity problem
- *Computability problem:* Is a given language computable?

Of course then e.g. the Equivalence problem is undecidable for CE-languages since already the "weaker" Universality problem is so. And many of these problems were undecidable already for CF- and CS-languages.

There are numerous variants of the Turing machine, machines with several tapes or many-dimensional tapes, etc. None of these, nor any of the other definitions of an algorithm, lead to a family of recognized languages larger than $\mathcal{CE}$.

On the other hand, CE-languages can be recognized by Turing machines of even more restricted nature than the determistic ones. A DTM M is called a *reversible Turing machine* (*RTM*) if there is another DTM M' with the property that if $\kappa_1 \vdash_M \kappa_2$ for the configurations κ_1 and κ_2, then (with some insignificant technical changes) $\kappa_2 \vdash_{M'} \kappa_1$. The DTM M' thus works as the time reverse of M! Yves Lecerf proved already in 1962 that each CE-language can be recognized by an RTM.[3] Similarly, any algorithm can be replaced by an equivalent reversible one. This result has now become very important e.g. in the theory of quantum computing.

6.3 Time Complexity Classes (A Brief Overview)

For automata Chomsky's hierarcy is mostly about the effects various limitations on memory usage have. Similar limitations can be set for *time,* i.e., the number of computational steps (transitions) taken.

Rather than setting strict limits for the number of steps, it is usually better to set them asymptotically, that is, only for input words of sufficient length and modulo a constant coefficient. Indeed, a Turing machine can easily be "accelerated linearly", and short inputs quickly dealt with separately.

The *time complexity class* $\mathcal{TIME}\big(f(n)\big)$ then is the family of those languages L that can be recognized by a deterministic Turing machine M with the following property: There are constants C and n_0 (depending on L) such that M uses at most $Cf(n)$ steps for any input word of length $n \geq n_0$. The time complexity class $\mathcal{NTIME}\big(f(n)\big)$ is defined similarly, using nondeterministic Turing machines. Choosing different functions $f(n)$, a complicated infinite hierarchy inside $\mathcal{C}$ is thus created.

Thinking about practical realization of algorithms, for instance the time complexity classes $\mathcal{TIME}(2^n)$ and $\mathcal{NTIME}(2^n)$ are unrealistic, computations are just too time-consuming. On the other hand, the classes $\mathcal{TIME}(n^d)$ (deterministic polynomial-time languages) are much better realizable in this sense. Indeed, the family of all deterministic polynomial-time languages

$$\mathcal{P} = \bigcup_{d \geq 1} \mathcal{TIME}(n^d)$$

is generally thought to be the family of tractably recognizable languages.

The classes $\mathcal{NTIME}(n^d)$ are not similarly clearly practically realizable as $\mathcal{P}$ is. The family

$$\mathcal{NP} = \bigcup_{d \geq 1} \mathcal{NTIME}(n^d)$$

can of course be defined, it is however a long-standing and very famous open problem whether or not $\mathcal{P} = \mathcal{NP}$! Moreover, the class $\mathcal{NP}$ has turned out to contain numerous "universal" languages, the so-called *$\mathcal{NP}$-complete languages,* having the property that if any one of them is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

[3] The original article reference is LECERF, M.Y.: Machines de Turing réversibles. Récursive insolubilité en $n \in N$ de l'équation $u = \theta^n u$, où θ est un "isomorphisme de codes". *Comptes Rendus* **257** (1963), 2597–2600.

> ”Indeed, the entire proof that Huffman’s algorithm works is comparatively trivial in the binary case. Most texts prove only the $s = 2$ case and leave the (nontrivial!) generalization as a problem.”
>
> (Robert J. McEliece: *The Theory of Information and Coding*)

Chapter 7

CODES

7.1 Code. Schützenberger’s Criterium

A language L is a *code* if it is nonempty and the words of L^* can be uniquely expanded as a concatenation of words of L. This expansion is called *decoding.* To be more exact, if $u_1, \dots, u_m$ and $v_1, \dots, v_n$ are words in L and

$$u_1 \cdots u_m = v_1 \cdots v_n,$$

then $u_1 = v_1$, which, sufficiently many times repeated, quarantees uniqueness of decoding.

Error-correcting codes (see the course Coding Theory) and some cryptographic codes (see the course Mathematical Cryptology) are codes in this sense, too, even if rather specialized such. On the other hand, codes used in data compression (see the course Information Theory) are of a fairly general type of codes.

Some general properties may be noticed immediately:

- A code does not contain the empty word.
- Any nonempty sublanguage of a code is also a code, a so-called *subcode.*
- If the alphabet of the code is unary (contains only one symbol), then the code consists of exactly one nonempty word.

Since codes over a unary alphabet are so very simple, it will be assumed in the sequel that alphabets contain at least two symbols. Codes do not have many noticeable closure properties. The intersection of two codes is a code if it is nonempty. Mirror images and powers of codes are codes, too.

A code can be defined and characterized by various conditions on its words. A language L is said to be *catenatively independent* if

$$L \cap LL^+ = \emptyset,$$

i.e., no word of the language L can be given as a concatenation of several words of L. Clearly a code is always catenatively independent, but the converse is not true in general. Therefore additional conditions are needed to specify a code. One such is given by

Schützenberger’s Criterium. *A catenatively independent language L over the alphabet Σ is a code if and only if the following condition is valid for all words $w \in \Sigma^*$:*

(∗) *If there exist words t, x, y and z in L^* such that $wt = x$ and $yw = z$, then $w \in L^*$.*

Proof. Let us first assume that the condition (∗) holds true and show that then L is a code. Suppose the contrary is true: L is not a code. Then there are words $u_1, \dots, u_m$ and $v_1, \dots, v_n$ in L such that

$$u_1 \cdots u_m = v_1 \cdots v_n,$$

but $u_1 \neq v_1$. One of the words u_1 and v_1 is a proper prefix of the other, say $v_1 = u_1 w$ where $w \neq \Lambda$. Now w is a suffix of v_1 and a prefix of $u_2 \cdots u_m$, so that according to the condition (∗) w is in L^+. This however means that L is catenatively dependent, a contradiction.

Assume second that L is a code and show validity of condition (∗). Let us again suppose the contrary is true: There exist words t, x, y and z in L^* such that $wt = x$ and $yw = z$, but $w \notin L^*$. None of the words t, x, y, z and w can then be empty. Writing the words y and z as concatenations of words in L as

$$y = u_1 \cdots u_m \quad \text{and} \quad z = v_1 \cdots v_n,$$

it follows that

$$u_1 \cdots u_m w = v_1 \cdots v_n.$$

Obviously it may be assumed that $u_1 \neq v_1$. But now

$$v_1 \cdots v_n t = u_1 \cdots u_m w t = u_1 \cdots u_m x$$

and $u_1 \neq v_1$, and L cannot possibly be a code, a contradiction again. □

7.2 The Sardinas–Patterson Algorithm

As such Schützenberger's criterium does not give an algorithm for deciding whether or not a given finite language L is a code. It is, however, possible to use it to derive such algorithms[1], e.g. the classical *Sardinas–Patterson algorithm:*

1. Set $i \leftarrow 0$ and $L_i \leftarrow L$.

2. Set $i \leftarrow i + 1$ and

   $$L_i \leftarrow \{w \mid w \neq \Lambda, \text{ and } xw = y \text{ or } yw = x \text{ for some words } x \in L \text{ and } y \in L_{i-1}\}.$$

3. If $L_i \cap L \neq \emptyset$, then return "no" and quit.

4. If $L_i = L_j$ for an index j, $1 \leq j < i$, then return "yes" and quit. Otherwise go to item 2.

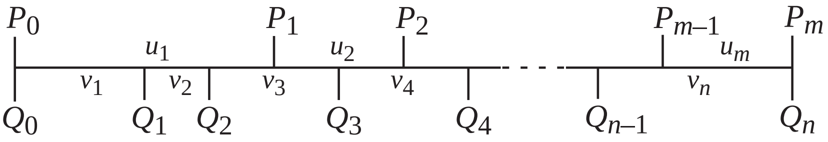
Theorem 31. *The Sardinas–Patterson algorithm gives the correct answer.*

Proof. Let us first show that if L is not a code, then one of the languages $L_1, L_2, \dots$ will contain a word in L. We assume thus that L is not a code. The case where $\Lambda \in L$ is immediately clear, because then $L_1 \cap L \neq \emptyset$, and we move on to the other cases. There are thus words $u_1, \dots, u_m$ and $v_1, \dots, v_n$ in L such that

$$u_1 \cdots u_m = v_1 \cdots v_n = w,$$

but $u_1 \neq v_1$. This situation can be illustrated graphically:

[1] Cf. e.g. Berstel & Perrin & Reutenauer.



In the figure subwords are indicated as line segments and their ends by points in the line. Thus for example always

$$P_{i-1}P_i = u_i \quad \text{and} \quad Q_{j-1}Q_j = v_j.$$

The initial point is $P_0 = Q_0$ and the terminal point $P_m = Q_n$. We may assume that these are the only cases where the points P_i and Q_j coincide, otherwise we simply replace w by one of its prefixes. Let us then expand w to a concatenation of subwords using all the points $P_1, \dots, P_m, Q_1, \dots, Q_n$. Subwords (line segments) of the form P_iQ_j or Q_jP_i, where P_i and Q_j are consequtive and $i, j \geq 1$, are now in the union $L_1 \cup L_2 \cup \cdots$, as is easily seen from the description of the algorithm, starting from the beginning of the word w.

Now, if the point immediately preceding the terminal point $P_m = Q_n$ is P_{m-1}, as it is in the above figure, then the subword $Q_{n-1}P_{m-j}$, where P_{m-j} is the point immediately succeeding Q_{n-1}, is in one of the languages L_i $(i \geq 1)$. But this implies that $u_m = P_{m-1}P_m$ is in the language L_{i+j}. The case where the point immediately preceding the terminal point $P_m = Q_n$ is Q_{n-1}, is dealt with analogously. This concludes the first part of the proof.

Second we show that if L is a code, then none of the languages $L_1, L_2, \dots$ contains words in L. For this, Schützenberger's criterium will be needed.

We begin by showing that for each word $w \in L_i$ we have $L^*w \cap L^* \neq \emptyset$, in other words, there are words $s, t \in L^*$ such that $sw = t$. This is trivial in case $i = 0$. We continue from this by induction on i, and set the induction hypothesis according to which each word in L_{i-1} is a suffix of some word in L^*, as indicated. Consider then a word w in L_i. The way L_i is defined by the algorithm implies that there are words $x \in L$ and $y \in L_{i-1}$ such that

$$xw = y \quad \text{or} \quad yw = x.$$

On the other hand, the induction hypothesis implies that there words $s, t \in L^*$ such that $sy = t$. We deduce that

$$sxw = sy = t \quad \text{or} \quad sx = syw = tw,$$

and the claimed result is valid for L_i, too.

To proceed with the second part of the proof, we assume the contrary, i.e., that there actually is a word $w \in L \cap L_i$ for some index value $i \geq 1$, and derive a contradiction. Following the algorithm there then are words $x \in L$ and $y \in L_{i-1}$ such that

$$yw = x \quad \text{or} \quad xw = y.$$

If $yw = x$, it follows that $L^*y \cap L^* \neq \emptyset$ (by the above) and $L^* \cap yL^* \neq \emptyset$, and, as a consequence of the Schützenberger criterium, that $y \in L^*$, a contradiction. (Note that the case $i = 1$ is clear anyway.) The other alternative $xw = y$ then must be true, and consequently $i \geq 2$ (why?) and $y \in L^*$. Again by the algorithm there are words $x' \in L$ and $y' \in L_{i-2}$ such that

$$y'y = x' \quad \text{or} \quad x'y = y'.$$

The former alternative is again excluded using the Schützenberger criterium as above. The latter alternative $x'y = y'$ then must hold true, and consequently $i \geq 3$ (why?) and $y' \in L^*$.

We may continue in this way indefinitely. Thus a contradiction arises for each value of i.

Finally we note that if L is a code, then the algorithm eventually stops in item 4. There are only finitely many possible languages L_i because L is finite and lengths of words cannot increase indefinitely. □

For infinite languages the case is more complicated. Deciding whether or not a given regular language L is a code can be done algorithmically as follows. First, the case where L contains the empty word is clear, of course. Excluding that, it is fairly easy to transform a DFA recognizing L to a Λ-NFA M, which accepts exactly all words that can be written as a concatenation of words in L in at least two different ways (cf. the proof of Kleene's Theorem). The problem is then reduced to deciding emptiness of the regular language $L(M)$.[2]

On the other hand, there is no general algorithm for deciding whether or not a given CF-language is a code. This can be shown using e.g. the Post correspondence problem introduced in Section 4.7. For each pair of PCP input morphisms

$$\sigma_1 : \Sigma^* \to \Delta^* \quad \text{and} \quad \sigma_2 : \Sigma^* \to \Delta^*$$

we just construct the language

$$L = \{\sigma_1(w)\#\hat{w}\$ \mid w \in \Sigma^+\} \cup \{\sigma_2(w)\#\hat{w}\$\sigma_1(v)\#\hat{v}\$ \mid w, v \in \Sigma^+\}$$

over the alphabet $\Sigma \cup \Delta \cup \{\#, \$\}$. It is not difficult to see that L is CF, and that it is a code if and only if there is no word $w \in \Sigma^+$ such that $\sigma_1(w) = \sigma_2(w)$.

7.3 Indicator Sums. Prefix Codes

Various numerical sums make it possible to arithmetize the theory of codes, and form an important tool—especially for error-correcting codes but in general as well, cf. the course Coding Theory.

One such sum is the *indicator sum* of a language L over the alphabet Σ

$$\text{is}(L) = \sum_{w \in L} M^{-|w|}$$

where M is the cardinality of Σ. In general, the indicator sum may be infinite, e.g.

$$\text{is}(\Sigma^*) = \sum_{n=0}^{\infty} M^n M^{-n} = \infty.$$

In codes, on the other hand, words appear sparsely and the situation is different:

Markov–McMillan Theorem. *For every code L (finite or infinite)* $\text{is}(L) \leq 1$.

Proof. Let us first consider a finite code $\{w_1, w_2, \dots, w_k\}$ such that the lengths of the words are

$$l_1 \leq l_2 \leq \cdots \leq l_k.$$

[2] This idea works of course especially for finite languages, but Sardinas–Patterson is much better!

We take an arbitrary positive integer r (and finally the limit $r \to \infty$). Then

$$\Big(\sum_{n=1}^{k} M^{-l_n}\Big)^r = \underbrace{\Big(\sum_{n=1}^{k} M^{-l_n}\Big)\cdots\Big(\sum_{n=1}^{k} M^{-l_n}\Big)}_{r \text{ copies}} = \sum_{n_1=1}^{k}\sum_{n_2=1}^{k}\cdots\sum_{n_r=1}^{k} M^{-l_{n_1}-l_{n_2}-\cdots-l_{n_r}}.$$

The sum $l_{n_1} + l_{n_2} + \cdots + l_{n_r}$ is the length of the concatenation of the r code words $w_{n_1}w_{n_2}\cdots w_{n_r}$. When the indices $n_1, n_2, \dots, n_r$ independently run through the values $1, 2, \dots, k$, all possible concatenations of r code words will be included, in particular possible multiple occurrences.

Let us then denote by s_j the number of all words of length j obtained by concatenating r code words, including possible multiple occurrences. It is not however possible to get the same word by different concatenations of code words, so $s_j \le M^j$. The possible values of j are clearly

$$rl_1\ ,\ rl_1 + 1\ ,\dots,\ rl_k.$$

Hence

$$\Big(\sum_{n=1}^{k} M^{-l_n}\Big)^r = \sum_{j=rl_1}^{rl_k} s_j M^{-j} \le \sum_{j=rl_1}^{rl_k} M^j M^{-j} = rl_k - rl_1 + 1 \le rl_k.$$

Extracting the r^{th} root on both sides and letting $r \to \infty$ we get

$$\sum_{n=1}^{k} M^{-l_n} \le \sqrt[r]{rl_k} \to 1,$$

since

$$\lim_{r\to\infty} \ln \sqrt[r]{rl_k} = \lim_{r\to\infty} \frac{\ln r + \ln l_k}{r} = 0.$$

The left hand side of this inequality does not depend on r, and thus the inequality must hold true in the limit $r \to \infty$ as well.

For infinite codes the result follows from the above. Indeed, if it is not true for some infinite code, this will be so for some of its finite subcodes, too. □

The special case where $\mathrm{is}(L) = 1$ has to do with so-called maximal codes. A code L is *maximal* if it is not a proper subcode of another code, that is, it is not possible to add even one word to L without losing unique decodability.[3] Indeed,

Corollary. *If* $\mathrm{is}(L) = 1$ *for a code L, then the code is maximal.*

It may be noted that the converse is true for finite codes, but not for infinite codes in general.

The converse of the Markov–McMillan Theorem is not true in general, because there are noncodes with $\mathrm{is}(L) \le 1$ (can you find one?). Some sort of a converse is, however, valid:

Kraft's Theorem. *If $\sum_{n=1,2,\dots} M^{-l_n} \le 1$ and $l_1 > 0$, then there is a code (finite or infinite) over an alphabet of cardinality M such that the lengths of its words are $l_1, l_2, \dots$ In addition the code may be chosen to be a* prefix code, *i.e., a code where no code word is a prefix of another code word.*[4]

[3] In the literature this is sometimes also called a *complete code,* and ”maximality” is reserved for other purposes.

[4] Equally well it could be chosen to be a *suffix code* where no code word is a suffix of another code word.

Proof. First, it may be noted that the prefix condition alone guarantees that the language is a code. That is, if no word is a prefix of another word in a language ($\neq \{\Lambda\}$), then the language is a code. Second, it is noted that the sum condition implies

$$(*) \qquad \sum_{n=1}^{j} M^{l_j - l_n} \leq M^{l_j} \quad (j = 1, 2, \dots).$$

Apparently we may assume $1 \leq l_1 \leq l_2 \leq \cdots$

The following process, iterated sufficiently many times, produces the desired prefix code.

1. Set $L \leftarrow \emptyset$, $i \leftarrow 1$ and
$$W_j \leftarrow \Sigma^{l_j} \quad (j = 1, 2, \dots).$$
Note that W_j then contains M^{l_j} words.

2. Choose a word w_i in W_i (any word) and set
$$W_j \leftarrow W_j - w_i \Sigma^{l_j - l_i} \quad (j = i, i + 1, \dots).$$
Before this operation, altogether
$$M^{l_j - l_1} + M^{l_j - l_2} + \cdots + M^{l_j - l_{i-1}}$$
words were removed from the original W_j. According to the inequality $(*)$, W_i may then be empty, but $W_{i+1}, W_{i+2}, \dots$ are not empty.

3. Set $L \leftarrow L \cup \{w_i\}$. Assuming that there still are given lengths of words not dealt with, set $i \leftarrow i + 1$ and go to item 2. Otherwise return L and stop. In case the number of given lengths of words is infinite, the process runs indefinitely producing the prefix code L in the limit. □

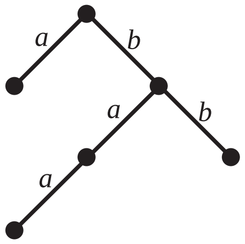
Corollary. *Any code can be replaced by a prefix code over the same alphabet preserving all lengths of code words.*

This result is very important because a prefix code is extremely easily decoded (can you see how?).

Prefix codes are closely connected with certain tree structures, cf. the derivation tree in Section 4.1. A tree T is a *prefix tree* over the alphabet Σ if

- the branches (edges) of T are labelled by symbols in Σ,
- branches leaving a vertex all have different label symbols, and
- T has vertices other than the root (T may even be infinite).

The following tree is an example of a prefix tree over the alphabet $\{a, b\}$:



Any prefix tree T determines a language $L(T)$, obtained by traversing all paths from the root to the leaves collecting and concatenating labels of branches. In the example above $L(T) = \{a, bb, baa\}$. Because of its construction, for any prefix tree T the language $L(T)$ is a prefix code. The converse result holds true, too: For each prefix code L there is a prefix tree T such that $L = L(T)$. This prefix tree is obtained as follows:

1. First set T to the root vertex.

2. Sort the words of L lexicographically, i.e., first according to length and then alphabetically inside each length: $w_1, w_2, \ldots$ Set $i \leftarrow 1$.

3. Find the longest word u in T, as traversed from the root to a vertex V, that is a prefix of w_i and write $w_i = uv$. Extend then the tree T by annexing a path starting from the vertex V and with branches labelled in order by the symbols of v. In the beginning V is the root.

4. If L is finite and its words are all dealt with, then return T and stop. Otherwise set $i \leftarrow i + 1$ and go to item 3. If L is infinite, then the corresponding prefix tree is obtained in the limit of infinitely many iterations.

7.4 Bounded-Delay Codes

It is clearly not possible to decode an infinite code using a generalized sequential machine (GSM, cf. Section 2.8). On the other hand, there are finite codes that cannot be decoded by a GSM either. For example, for the code $\{a, ab, bb\}$, not one code word of the word ab^n can be decoded until it is read through.

A finite code L has *decoding delay* p if $u_1 = v_1$ whenever

$$u_1, \ldots, u_p \in L \quad \text{and} \quad v_1, \ldots, v_n \in L$$

and $u_1 \cdots u_p$ is a prefix of $v_1 \cdots v_n$. This means that a look-ahead of p code words is needed for decoding one code word. A code that has a decoding delay p for some p, is a so-called *bounded-delay code.* Note in particular that codes of decoding delay 1 are the prefix codes in the previous section.

A finite code L of decoding delay p over the alphabet Σ can be decoded using a GSM S as follows:

1. If m is the length of the longest word in L^p, then the states of S are $\langle w \rangle$ where w goes through all words of length at most m. The initial state is $\langle \Lambda \rangle$.

2. The input/output alphabet of S is $\Sigma \cup \{\#\}$. The special symbol # marks the end of the input word so that S will know when to empty its memory. The decoding is indicated by putting # between code words of L.

3. If S is in the state $\langle w \rangle$ and $w \notin L^p$, then on input $a \in \Sigma$ it moves to the state $\langle wa \rangle$ outputting Λ.

4. If S is in a state $\langle u_1 u_2 \cdots u_p \rangle$ where $u_1, u_2, \ldots, u_p$ are code words, then on input $a \in \Sigma$ it moves to the state $\langle u_2 \cdots u_p a \rangle$ outputting $u_1\#$. (Especially, if $p = 1$, then S just moves to the state $\langle a \rangle$.)

5. If S is in a state $\langle u_1 \cdots u_k \rangle$ where $k \geq 2$ and $u_1, \dots, u_k$ are code words, then on input # it moves to the state $\langle \Lambda \rangle$ outputting $u_1\# \cdots \#u_k$.

6. If S is in a state $\langle u \rangle$ where $u \in L$ or $u = \Lambda$, then on input # it moves to the state $\langle \Lambda \rangle$ outputting u.

For the sake of completeness, state transitions and outputs of S dealing with words outside L^* should be added.

Prefix codes are especially easy to decode in this way. Thus, if what is important about code words is their lengths, say, getting as short code words as possible, it is viable to use prefix codes. This is always possible, by the Markov–McMillan Theorem and Kraft's Theorem.

7.5 Optimal Codes and Huffman's Algorithm

In optimal coding first the alphabet $\Sigma = \{c_1, c_2, \dots, c_M\}$ is fixed, and then the number k and the weights $P_1, P_2, \dots, P_k$ of the code words. The weights are nonnegative real numbers summing to 1. When relevant, they may be interpreted as probabilities, frequencies, etc. In what follows it will be assumed that the weights are indexed in nonincreasing order:

$$P_1 \geq P_2 \geq \cdots \geq P_k \ (\geq 0).$$

Thus, the extreme cases are $P_1 = 1, P_2 = \cdots = P_k = 0$ and $P_1 = \cdots = P_k = 1/k$.

The problem then is to find code words $w_1, w_2, \dots, w_k$, corresponding to the given weights, such that the *mean length*

$$P_1|w_1| + P_2|w_2| + \cdots + P_k|w_k|$$

is the smallest possible. The code thus obtained is a so-called *optimal code.* Since mean length depends on the code words only via their lengths, it may be assumed in addition that the code is a prefix code. We will denote $|w_i| = l_i$ $(i = 1, 2, \dots, k)$, and the mean length by $\bar{l}$. Optimal coding may then be given as the following integer optimization problem:

$$\begin{cases} \bar{l} = \displaystyle\sum_{i=1}^{k} P_i l_i = \text{min!} \\ \displaystyle\sum_{i=1}^{k} M^{-l_i} \leq 1 \\ l_1, l_2, \dots, l_k \geq 1. \end{cases}$$

Optimal codes are closely connected with information theory[5] and data compression. Indeed, numerous algorithms were developed for finding them. The best known such is probably Huffman's algorithm. Using the above notation we proceed by first showing that

Lemma. *The code words $w_1, w_2, \dots, w_k$ of an optimal prefix code may be chosen in such a way that*

[5] According to the classical *Shannon Coding Theorem,* $H \leq \bar{l} \leq H + 1$ where

$$H = -P_1 \log_M P_1 - \cdots - P_k \log_M P_k$$

is the so-called *entropy,* cf. the course Information Theory.

- *$l_1 \leq \cdots \leq l_{k-s} \leq l_{k-s+1} = \cdots = l_k$ where $2 \leq s \leq M$ and $s \equiv k \mod M-1$,*[6]
- *$w_{k-s+1}, \ldots, w_k$ differ only in their last symbol, which is c_i for w_{k-s+i}, and*
- *the common prefix of length $l_k - 1$ of $w_{k-s+1}, \ldots, w_k$ is not a prefix of any of the words $w_1, \ldots, w_{k-s}$.*

Proof. Some optimal (prefix) code $\{w'_1, w'_2, \ldots, w'_k\}$ of course exists, with lengths of code words $l'_1, l'_2, \ldots, l'_k$. If now $l'_i > l'_{i+1}$ for some i, then we exchange w'_i and w'_{i+1}. This changes the mean length by

$$P_i l'_{i+1} + P_{i+1} l'_i - (P_i l'_i + P_{i+1} l'_{i+1}) = (P_i - P_{i+1})(l'_{i+1} - l'_i) \leq 0.$$

The code is optimal, so the change is $= 0$, and the code remains optimal. After repeating this operation sufficiently many times we may assume that the lengths satisfy

$$l_1 \leq l_2 \leq \cdots \leq l_k.$$

Even so, there may be several such optimal codes. We choose the code to be used in such way that the sum $l_1 + l_2 + \cdots + l_k$ is the smallest possible, and denote

$$(*) \qquad \Delta = M^{l_k} - \sum_{i=1}^{k} M^{l_k - l_i}.$$

By the Markov–McMillan Theorem, $\Delta \geq 0$. On the other hand, if $\Delta \geq M - 1$, then the lengths $l_1, \ldots, l_{k-1}, l_k - 1$ would satisfy the condition of Kraft's Theorem, which is impossible because $l_1 + l_2 + \cdots + l_k$ is the smallest possible. We deduce that $2 \leq M - \Delta \leq M$.

Let us then denote by r the number of code words of length l_k. We then have $r \geq 2$. Indeed, otherwise removing the last symbol of w_k would not destroy the prefix property of the code, which is again impossible because $l_1 + l_2 + \cdots + l_k$ is the smallest possible. The equality $(*)$ implies

$$\Delta \equiv -r \mod M \ , \quad \text{i.e.} \ , \quad r \equiv M - \Delta \mod M.$$

We know that $2 \leq M - \Delta \leq M$, so r can be written as

$$r = tM + (M - \Delta)$$

where $t \geq 0$. The equality $(*)$ further implies

$$\Delta \equiv 1 - k \mod M - 1$$

since

$$M \equiv 1 \mod M - 1 \ , \quad \text{i.e.} \ , \quad M - \Delta \equiv k \mod M - 1.$$

Hence $M - \Delta$ must be exactly the number s in the lemma, and $r \geq s$.

Reindexing if necessary we may assume that among the code words $w_{k-r+1}, \ldots, w_k$ of length l_k the words $w_{k-t}, \ldots, w_k$ have different prefixes of length $l_k - 1$, denoted by $z_1, \ldots, z_{t+1}$. Recall that $r = tM + s$. Finally we replace the words $w_{k-r+1}, \ldots, w_k$ by the words

$$z_1c_1 \ , \ldots , \ z_1c_M \ , \ z_2c_1 \ , \ldots , \ z_2c_M \ , \ldots , \ z_tc_1 \ , \ldots , \ z_tc_M \ , \ z_{t+1}c_1 \ , \ldots , \ z_{t+1}c_s.$$

This destroys neither the prefix property nor the optimality of the code. □

[6] The congruence or modular equality $a \equiv b \mod m$ means that $a - b$ is divisible by m.

Huffman's algorithm is a recursive algorithm. To define the recursion, consider the words $v_1, \dots, v_{k-s+1}$, obtained from the code $\{w_1, w_2, \dots, w_k\}$ given by the Lemma, as follows:

(a) $v_1 = w_1, \dots, v_{k-s} = w_{k-s}$, and

(b) v_{k-s+1} is obtained by removing the last symbol $(= c_1)$ of w_{k-s+1}.

The words $v_1, \dots, v_{k-s+1}$ then form a prefix code. Taking the corresponding weights to be

$$P_1, \dots, P_{k-s}, P_{k-s+1} + \cdots + P_k$$

gives the mean length

$$\overline{l'} = \sum_{i=1}^{k-s} P_i l_i + \sum_{i=k-s+1}^{k} P_i(l_k - 1) = \bar{l} - \sum_{i=k-s+1}^{k} P_i.$$

This means that the code $\{v_1, \dots, v_{k-s+1}\}$ is optimal, too. Indeed, otherwise there would be a prefix code $\{v'_1, \dots, v'_{k-s+1}\}$ with a smaller mean length, and the mean length of the prefix code

$$\{v'_1, \dots, v'_{k-s}, v'_{k-s+1}c_1, \dots, v'_{k-s+1}c_s\}$$

would be smaller than

$$\overline{l'} + P_{k-s+1} + \cdots + P_k = \bar{l}.$$

On the other hand, if $\{v'_1, \dots, v'_{k-s+1}\}$ is an optimal prefix code, corresponding to the weights

$$P_1, \dots, P_{k-s}, P_{k-s+1} + \cdots + P_k,$$

then its mean length is $\overline{l'}$ and

$$\{v'_1, \dots, v'_{k-s}, v'_{k-s+1}c_1, \dots, v'_{k-s+1}c_s\}$$

is an optimal prefix code corresponding to the weights $P_1, P_2, \dots, P_k$. If this is not so, the Lemma would give an optimal code with mean length less than

$$\overline{l'} + P_{k-s+1} + \cdots + P_k,$$

and this would further give a code even "more optimal" than the code $\{v'_1, \dots, v'_{k-s+1}\}$.

Huffman's algorithm is the following recursion which, as shown above, outputs an optimal prefix code after receiving as input the weights $P_1, P_2, \dots, P_k$ in nonincreasing order. Note in particular that a zero weight is by no means excluded.

1. If $k \leq M$, then return $w_1 = c_1, w_2 = c_2, \dots, w_k = c_k$, and quit.

2. Otherwise the optimal code $\{w_1, w_2, \dots, w_k\}$, corresponding to the input weights $P_1, P_2, \dots, P_k$ is immediately obtained, once the code $\{v_1, v_2, \dots, v_{k-s+1}\}$ is known, by taking

   $$w_1 = v_1 \ , \ w_2 = v_2 \ , \dots , \ w_{k-s} = v_{k-s}$$

   and

   $$w_{k-s+1} = v_{k-s+1}c_1 \ , \dots , \ w_k = v_{k-s+1}c_s.$$

3. To get the code $\{v_1, \dots, v_{k-s+1}\}$ compute first s and set
$$Q_1 \leftarrow P_1\ , \dots,\ Q_{k-s} \leftarrow P_{k-s} \quad \text{and} \quad Q_{k-s+1} \leftarrow P_{k-s+1} + \cdots + P_k.$$
Set further the new values of the weights $P_1, P_2, \dots, P_{k-s+1}$ to be $Q_1, Q_2, \dots, Q_{k-s+1}$ in nonincreasing order, and $k \leftarrow k - s + 1$, and go to item 1.

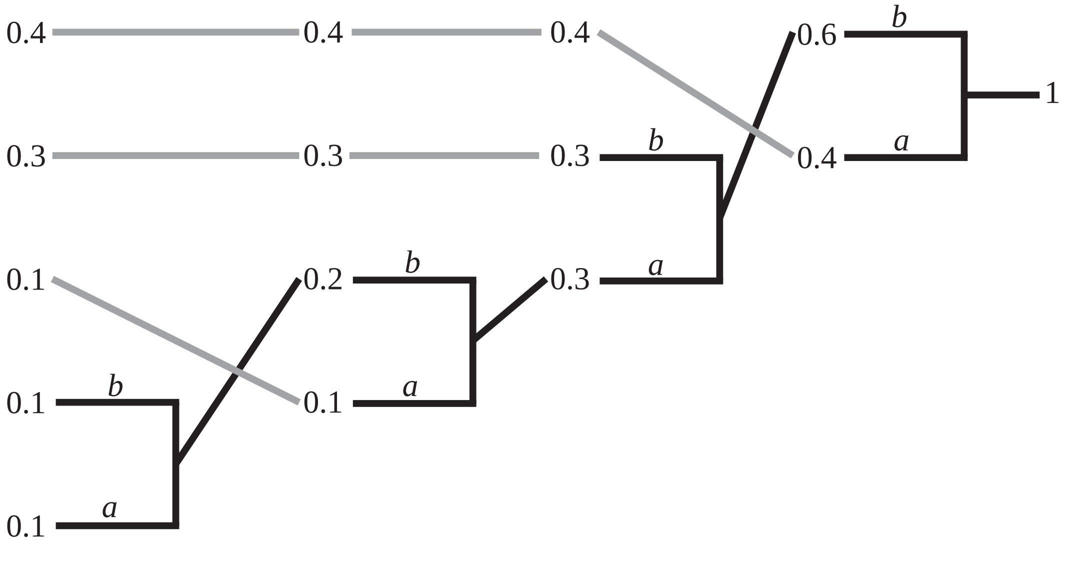
The recursion is finite because the number of weights decreases in each iteration, reaching finally a value $\leq M$.

Note. *The first value of s satisfies $s \equiv k \mod M-1$. The "next k" will then be $k-s+1$ and*
$$k - s + 1 \equiv 1 \mod M - 1.$$
Thus the next value of s is in fact M. Continuing in this way it is further seen that all remaining values of s equal M, hence only the first value of s needs to be computed! Note also that in the case $M = 2$ (the binary Huffman algorithm) *we always have $s = 2$.*

Huffman's algorithm essentially constructs a prefix tree (the so-called *Huffman tree*), corresponding to an optimal prefix code, using the following "bottom-up" method:

- The branches of the tree are labelled by the symbolds of Σ, as was done before. The vertices are labelled by weights.
- In the beginning only the leaves are labelled by the weights $P_1, P_2, \dots, P_k$. The leaves are then also labelled as unfinished.
- At each stage a value of s is computed according to the number of unfinished vertices, cf. the Note above. Then a new vertex is added to the tree and the unfinished vertices corresponding to the s smallest labels (weights) are connected to the new vertex by branches labelled by the symbols $c_1, \dots, c_s$. These s vertices are then labelled as finished, and the new vertex receives a weight label which is the sum of the s smallest weights, and is also labelled as unfinished. The process then continues.
- The tree is ready when the weight label 1 is reached and the root is found.

This procedure offers a "graphical" way of finding an optimal code, for relatively small numbers of words anyway. Below there is an example of such a "graphical" Huffman tree.



The branches of the tree are in black, the grey ones are there only for the sorting of weights. The tree defines the code words $a, bb, baa, babb, baba$ of an optimal prefix code over the alphabet $\{a, b\}$ for the weights $0.4, 0.3, 0.1, 0.1, 0.1$.

The optimal code returned by Huffman's algorithm is not in general unique, owing to occasional occurrences of equal weights. A drawback of the algorithm is the intrinsic trickiness of fast implementations.

An example of a problem that can be dealt with by Huffman's algorithm is the so-called *query system.* The goal is to find out the correct one of the k given alternatives V_1,...,V_k using questions with answers always coming from a fixed set of M choices. (Usually "yes" and "no" when $M = 2$.) The frequencies (probabilities) $P_1, \dots, P_k$ of the possible alternatives V_1,...,V_k as correct solutions are known. How should you then choose your questions so as to minimize their mean number?

"Go out to a nearby field. Pick a flower.
Simulate its development."

(Exercise 18.1 in Herman, G.T. & Rozenberg, G.:
Developmental Systems and Languages)

Chapter 8

LINDENMAYER'S SYSTEMS

8.1 Introduction

All rewriting systems so far dealt with are sequential, that is, only one symbol or (short) subword is rewritten at each step of the derivation. The corresponding parallel rewriting systems have been quite popular, too. The best known of these are *Lindenmayer's systems* or *L-systems.* Originally L-systems were meant to model morphology of plants, and of cellular structures in general. Nowadays their use is almost entirely in computer graphics, as models of plants and to generate fractals for various purposes.

Compared with grammars the standard terminology of Lindenmayer's systems is a bit different. In particular the following acronyms should be mentioned:

- 0: a context-free system
- I: a context-sensitive system
- E: terminal symbols are used
- P: productions are length-increasing

8.2 Context-Free L-Systems

Formally a *0L-system* is a triple $G = (\Sigma, \alpha, P)$ where Σ is the alphabet (of the language), $\alpha \in \Sigma^*$ is the so-called *axiom* and P is the set of productions. In particular, it is required that there is in P at least one production of the form $a \to w$ for each symbol a of Σ.

Derivations are defined as for grammars, except that at each step of the derivation some production must be applied to each symbol of the word (parallelism). Such a production may well be an identity production of the form $a \to a$. The language generated by G is

$$L(G) = \{w \mid \alpha \Rightarrow_G^* w\}.$$

A 0L-system is

- *deterministic* or a *D0L-system* if there is exactly one production $a \to w$ for each symbol a of the alphabet.
- *length-increasing* or *propagating* or a *P0L-system* if in each production $a \to w$ we have $w \neq \Lambda$, a PD0L-system then appearing as a special case.

The corresponding families of languages are denoted by $\mathit{0}\mathcal{L}$, $\mathcal{D}\mathit{0}\mathcal{L}$ etc.

Example. *The language* $\{a^{2^n} \mid n \geq 0\}$ *is generated by the simple PD0L-system* $G = \big(\{a\}, a,\, \{a \to a^2\}\big)$. *Note that this language is not CF but it is CS.*

Theorem 32. $\mathit{0}\mathcal{L} \subset \mathcal{CS}$

Proof. The proof is very similar to that of Theorem 12. For a 0L-system $G = (\Sigma, \alpha, P)$ we denote

$$\Delta = \{a \in \Sigma \mid a \Rightarrow_G^* \Lambda\}.$$

For each symbol $a \in \Delta$ we also define the number

$$d_a = \min_{a \Rightarrow_G^n \Lambda} n.$$

It is fairly easy to construct an LBA M recognizing $L(G)$. M simulates each derivation of G starting from the axiom α. When it meets a symbol $a \in \Delta$ the LBA M decides whether it continues with a derivation where $a \Rightarrow_G^* \Lambda$, or not. In the former case it immediately erases a in its tape moving the remaining suffix to the left, and it must then simulate the derivation of G at least d_a steps. This M remembers using its states. For the simulation M compresses several symbols of Σ into one tape symbol, if needed, so that the symbols of Δ to be erased can be fitted in, too. If the simulation overflows, i.e., uses more than the allowed space, M halts in a nonterminal state. Note how the simulation is especially easy for P0L-systems.

The language $\{a^n b^n \mid n \geq 1\}$ is a CF-language that clearly is not 0L. □

Adding a terminal alphabet $\Sigma_{\mathrm{T}} \subseteq \Sigma$ in a 0L-system $G = (\Sigma, \alpha, P)$ we get an *E0L-system* $G' = (\Sigma, \Sigma_{\mathrm{T}}, \alpha, P)$. The language generated is then

$$L(G') = \{w \mid \alpha \Rightarrow_{G'}^* w \text{ and } w \in \Sigma_{\mathrm{T}}^*\}.$$

Example. *The language* $\{a^n b^n \mid n \geq 1\}$ *is an E0L-language, it is generated by the E0L-system* $G = \big(\{a, b, c\},\, \{a, b\}, c, P\big)$ *where* P *contains the productions*

$$a \to a \quad , \quad b \to b \quad , \quad c \to acb \mid ab.$$

Theorem 33. $\mathcal{CF} \subset \mathcal{E}\mathit{0}\mathcal{L} \subset \mathcal{CS}$

Proof. It is easy to transform a CF-grammar into an equivalent E0L-system, just add identity productions for all symbols. On the other hand, there are D0L-languages that are not CF (the example above). Thus $\mathcal{CF} \subset \mathcal{E}\mathit{0}\mathcal{L}$.

It is an immediate consequence of Theorem 32 that $\mathcal{E}\mathit{0}\mathcal{L} \subseteq \mathcal{CS}$. It is however rather more difficult to find a CS-language that is not E0L. An example of such a language is the so-called *Herman's language*

$$H = \big\{w \bigm| |w|_a = 2^n, n = 0, 1, 2, \dots\big\}$$

over the alphabet $\{a, b\}$. Here $|w|_a$ denotes number of occurrences of the symbol a in the word w. It is not difficult to show that H is CS but a lot more difficult to show that it is not E0L (skipped here). □

In computer graphics symbols are interpreted as graphical operations performed in the order given by the derivation of the word.

Example. *In the beginning the picture window is the square* $0 \leq x \leq 1$, $0 \leq y \leq 1$. *Symbols of the alphabet* $\{a, b, c\}$ *are interpreted as follows:*

- *a: a line segment connecting the points* $(1/3, 1/2)$ *and* $(2/3, 1/2)$.
- *b: a line segment connecting the points* $(0, 0)$ *and* $(1/3, 1/2)$*; scaling to fit into the rectangle* $0 \leq x \leq 1/3$, $0 \leq y \leq 1/2$.
- *c: a line segment connecting the points* $(2/3, 1/2)$ *and* $(1, 1)$*; scaling + translation into the rectangle* $2/3 \leq x \leq 1$, $1/2 \leq y \leq 1$.

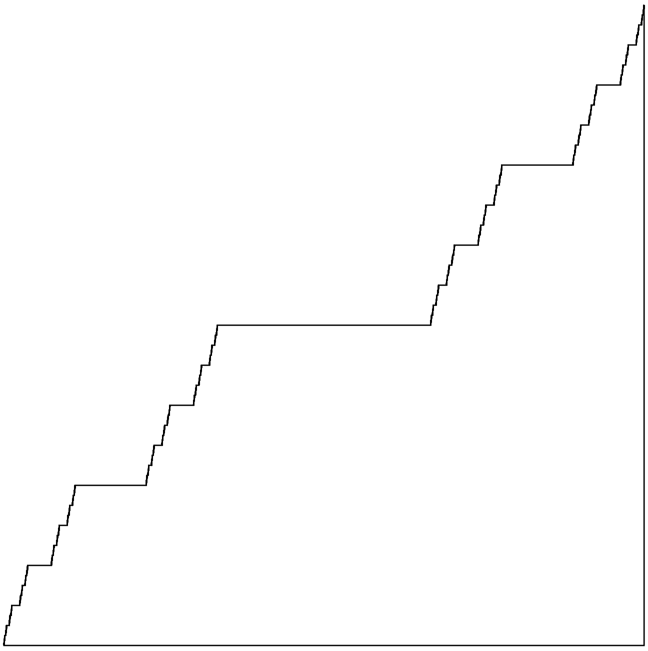
The D0L-system

$$G = \big(\{a, b, c\}, bac, \{a \to a, b \to bac, c \to bac\}\big)$$

then generates the words

$$bac\ ,\ bacabac\ ,\ bacabacabacabac\ , \ldots$$

Interpreted graphically the limiting picture is a fractal, the so-called Devil's staircase:



This is a graph of a continuous function that is almost everywhere differentiable, the derivative however being always $= 0$.

When modelling plants etc. the operations are three-dimensional: parts of trunk, leaves, branches, flowerings, and so on.

8.3 Context-Sensitive L-Systems or L-Systems with Interaction

An *IL-system* is a quintuple $G = (\Sigma, X_{\mathrm{L}}, X_{\mathrm{R}}, \alpha, P)$ where Σ is the alphabet (of the language), $X_{\mathrm{L}}, X_{\mathrm{R}} \notin \Sigma$ are the endmarkers (left and right), $\alpha \in \Sigma^*$ is the so-called *axiom* and P is the set of productions. These productions are of the special type

$$\langle u, a, v\rangle \to w$$

where $a \in \Sigma$ and u is the so-called *left context* and v is the *right context.*

To explain in more detail let us denote by d_L (resp. d_R) the length of the longest occurring left context (resp. right context), and define the sets

$$Y_\mathrm{L} = \big\{X_\mathrm{L}x \mid x \in \Sigma^* \text{ and } |x| < d_\mathrm{L}\big\} \cup \Sigma^{d_\mathrm{L}} \quad \text{and} \quad Y_\mathrm{R} = \big\{yX_\mathrm{R} \mid y \in \Sigma^* \text{ and } |y| < d_\mathrm{R}\big\} \cup \Sigma^{d_\mathrm{R}}.$$

It then will be required that for every symbol $a \in \Sigma$ and every word $u \in Y_\mathrm{L}$ and every word $v \in Y_\mathrm{R}$ there is a suffix u' of u and a prefix v' of v such that for some w the production $\langle u', a, v'\rangle \to w$ is in P. This particular condition guarantees that rewriting is possible in all situations. If there always is exactly one such production, the system is a deterministic IL-system or *DIL-system.*

The production $\langle u, a, v\rangle \to w$ is interpreted as being available for rewriting a only when the occurrence of a in the word being rewritten is in between the subwords u and v, in this order. As for L-systems in general, rewriting is parallel, that is, every symbol of a word must be rewritten simultaneously. If no symbol is ever erased or rewritten as Λ, i.e., $w \neq \Lambda$ in each production $\langle u, a, v\rangle \to w$, then the system is propagating or a so-called *PIL-system.*

The language generated by G is

$$L(G) = \{w \mid X_\mathrm{L}\alpha X_\mathrm{R} \Rightarrow_G^* X_\mathrm{L}wX_\mathrm{R}\}.$$

Note that the endmarkers are never rewritten, they are just there to indicate the boundaries.

Adding a terminal alphabet $\Sigma_\mathrm{T} \subseteq \Sigma$ to an IL-system $G = (\Sigma, X_\mathrm{L}, X_\mathrm{R}, \alpha, P)$ makes it a so-called *EIL-system*: $G' = (\Sigma, \Sigma_\mathrm{T}, X_\mathrm{L}, X_\mathrm{R}, \alpha, P)$. The generated language is then

$$L(G') = \{w \mid X_\mathrm{L}\alpha X_\mathrm{R} \Rightarrow_{G'}^* X_\mathrm{L}wX_\mathrm{R} \text{ and } w \in \Sigma_\mathrm{T}^*\}.$$

Theorem 34. *$\mathcal{EPIL} = \mathcal{CS}$ and $\mathcal{EIL} = \mathcal{CE}$.*

Proof. The proofs of these equalities are very similar to those of Theorems 19 and 20. □

Since context-sensitive L-families thus are the same as the corresponding families in Chomsky's hierarchy, they do not have as significant a role as the context-free ones do. It should be noted, however, that IL-systems form a curious parallel alternative for modelling computation.

It is interesting that for deterministic IL-systems the results are quite similar. The proofs are then however somewhat more complicated than that of Theorem 34.[1]

Theorem 35. *$\mathcal{EPDIL} = \mathcal{DCS}$ and $\mathcal{EDIL} = \mathcal{CE}$.*

This gives deterministic grammatical characterizations for the families $\mathcal{DCS}$ and $\mathcal{CE}$, and also a deterministic model for parallel computation.

An IL-system is structurally so close to a Turing machine that it can be regarded as one of the fairly few parallel-computation variants of the Turing machine. This does not offer much advantage for space complexity, for time complexity the question remains more or less open.

[1] The original reference is the doctoral thesis VITÁNYI, P.M.B.: *Lindenmayer Systems: Structure, Languages, and Growth Functions.* Mathematisch Centrum. Amsterdam (1978).

”Mathematics, that was a mistake.
I should have made the whole
thing a lot easier.”

(God in the movie *Oh, God! Book II*)

Chapter 9

FORMAL POWER SERIES

9.1 Language as a Formal Power Series

The *characteristic function* of the language L over the alphabet Σ is defined by

$$\chi_L(w) = \begin{cases} 1 \text{ if } w \in L \\ 0 \text{ otherwise.} \end{cases}$$

Obviously the characteristic function χ_L completely determines the language L. Moreover, as of operations of languages, we see that

$$\begin{aligned} \chi_{L_1 \cup L_2}(w) &= \max\big(\chi_{L_1}(w), \chi_{L_2}(w)\big), \\ \chi_{L_1 \cap L_2}(w) &= \min\big(\chi_{L_1}(w), \chi_{L_2}(w)\big) = \chi_{L_1}(w)\chi_{L_2}(w), \\ \chi_{L_1 L_2}(w) &= \max_{uv=w}\big(\chi_{L_1}(u)\chi_{L_2}(v)\big). \end{aligned}$$

Thinking of maximization as a kind of sum, these operations remind us of the basic operations of power series, familiar from calculus: *sum* (sum of coefficients), *Hadamard's product* (product of coefficients) and *Cauchy's product* (product of series, convolution). It is then natural to adopt a formal notation

$$\sum_{w \in \Sigma^*} \chi_L(w)w,$$

the so-called *formal power series* of the language L. Here we consider symbols of the alphabet $\Sigma = \{x_1, \dots, x_k\}$ as a set of noncommuting variables.

9.2 Semirings

As such the only novelty in representing a language as a formal power series is in the ”familiarity” of the power series notation. Moreover, the operation of concatenation closure, so central for languages, has no equivalent for formal power series. The importance of the notion of formal power series however becomes obvious when more general coefficients are allowed, since this naturally leads to generalizations of languages via changing the set concept. The coefficients are then assumed to come from a certain class of algebraic structures, the so-called semirings, where addition and multiplication are defined, and the usual laws of arithmetic hold true. Furthermore, semirings have a zero element and an identity element.

A *semiring* is an algebraic structure $R = (C, +, \cdot, \mathbf{0}, \mathbf{1})$ where C is a nonempty set (the elements) and the "arithmetic" operations have the desired properties—in addition to giving unique results and being always defined:

- The operations $+$ (*addition*) and $\cdot$ (*multiplication*) are both associative, i.e.,

  $$a + (b + c) = (a + b) + c \quad \text{and} \quad a \cdot (b \cdot c) = (a \cdot b) \cdot c.$$

  It follows from these that in chained sums and products parentheses can be placed in any proper way whatsoever and the result is always the same, indeed, it is then not necessary to use any parentheses at all:

  $$a_1 + a_2 + \cdots + a_n \quad \text{and} \quad a_1 \cdot a_2 \cdot \cdots \cdot a_n.$$

  In particular, this makes it possible to adopt the usual handy notation for multiples and powers

  $$na = \underbrace{a + a + \cdots + a}_{n \text{ copies}} \quad \text{and} \quad a^n = \underbrace{a \cdot a \cdot \cdots \cdot a}_{n \text{ copies}},$$

  and especially $1a = a$ and $a^1 = a$. The usual rules of calculation apply here:

  $$(n + m)a = (na) + (ma) \quad \text{and} \quad a^{n+m} = a^n \cdot a^m.$$

- Addition is commutative, i.e., $a + b = b + a$.

  Quite often multiplication is commutative, too, i.e., $a \cdot b = b \cdot a$. The semiring is then *commutative* or *Abelian.*

- Multiplication is distributive with respect to addition, i.e.,

  $$a \cdot (b + c) = (a \cdot b) + (a \cdot c) \quad \text{and} \quad (a + b) \cdot c = (a \cdot c) + (b \cdot c).$$

- $\mathbf{0} \in C$ is the so-called *zero element* satisfying $a + \mathbf{0} = \mathbf{0} + a = a$ for all elements a. In the notation for multiples we then agree that $0a = \mathbf{0}$. Note that there can be only one zero element (why?).

- $\mathbf{1} \in C$ is the so-called *identity element* satisfying $\mathbf{1} \cdot a = a \cdot \mathbf{1} = a$ for all elements a. In the power notation we then agree that $a^0 = \mathbf{1}$, and especially that $\mathbf{0}^0 = \mathbf{1}$. It is assumed in addition that $\mathbf{0} \neq \mathbf{1}$. (A semiring thus has at least two elements.) Note that there can be only one identity element (why?).

- $\mathbf{0} \cdot a = a \cdot \mathbf{0} = \mathbf{0}$

Parentheses may be omitted by agreeing that multiplication always precedes addition. The dot symbol for multiplication is often omitted, too, as usual.

Familiar examples of semirings are $N = (\mathbb{N}, +, \cdot, 0, 1)$ (natural numbers and the usual arithmetic operations) and $(\mathbb{R}_+, +, \cdot, 0, 1)$ (nonnegative real numbers and the usual arithmetic operations). Of course, all integers $\mathbb{Z}$ as well as all reals $\mathbb{R}$ equipped with the usual arithmetic operations form semirings, too. In the formal power series of a language the coefficients will be in a semiring, the so-called *Boolean semiring* $B = (\mathbb{B}, \max, \min, 0, 1)$ where $\mathbb{B} = \{0, 1\}$ (bits) and thus the only elements are the zero element and the identity element. Associativity and distributivity are quite easy to verify. All these semirings are in fact commutative. An example of a noncommutative semiring would be the semiring $(\mathbb{N}^{n \times n}, +, \cdot, \mathbf{O}_n, \mathbf{I}_n)$ of $n \times n$-matrices with elements in $\mathbb{N}$ where addition and multiplication are the customary matrix operations, $\mathbf{O}_n$ is the zero matrix and $\mathbf{I}_n$ is the identity matrix, and $n > 1$.

9.3 The General Formal Power Series

Defined analogously to the formal power series of a language, a general *formal power series in the semiring $R = (C, +, \cdot, \mathbf{0}, \mathbf{1})$ over the alphabet $\Sigma = \{x_1, x_2, \ldots, x_k\}$* is a mapping $\kappa : \Sigma^* \longrightarrow C$. It is traditionally denoted by an infinite formal sum (series)

$$\sum_{w \in \Sigma^*} \kappa(w)w$$

where the summands $\kappa(w)w$ are called *terms.* Often terms $\kappa(w)w$ where $\kappa(w) = \mathbf{0}$ will be omitted, and customary shorthand notations are used:

$$\kappa(\Lambda)\Lambda = \kappa(\Lambda) \quad \text{and} \quad \mathbf{1}w = w.$$

The $\kappa(w)$ in the term $\kappa(w)w$ is the *coefficient.* Parentheses closing a coefficient are also often omitted if no confusion can arise. The set of all such power series is denoted by $R\langle\langle\Sigma\rangle\rangle$.

As for the formal power series of languages, the operations $+$ and $\cdot$ of the semiring lead to operations for formal power series. Let

$$S_1 = \sum_{w \in \Sigma^*} \kappa_1(w)w \quad \text{and} \quad S_2 = \sum_{w \in \Sigma^*} \kappa_2(w)w.$$

Then

- $S_1 + S_2 = \sum_{w \in \Sigma^*} \big(\kappa_1(w) + \kappa_2(w)\big)w$ (*sum*).
- $S_1 \otimes S_2 = \sum_{w \in \Sigma^*} \kappa_1(w)\kappa_2(w)w$ (*Hadamard's product*).
- $S_1 S_2 = \sum_{w \in \Sigma^*} \kappa(w)w$ (*Cauchy's product*) where $\kappa(w) = \sum_{uv=w} \kappa_1(u)\kappa_2(v)$.

The products in turn lead to powers, especially Cauchy's product leads to the m^{th} *Cauchy power*

$$S^m = \sum_{w \in \Sigma^*} \kappa'(w)w$$

of the series

$$S = \sum_{w \in \Sigma^*} \kappa(w)w$$

where

$$\kappa'(w) = \sum_{u_1 u_2 \cdots u_m = w} \kappa(u_1)\kappa(u_2) \cdots \kappa(u_m).$$

Note in particular that the first power S^1 of the series is then the series itself, as it should be.

A formal power series where only finitely many coefficients are $\neq \mathbf{0}$ is called a *formal polynomial.* Formal power series of finite languages are exactly all formal polynomials of the Boolean semiring B. Formal polynomials containing only one term, i.e., polynomials of the form aw for some nonzero element $a \in C$ and word $w \in \Sigma^*$, are the so-called *monomials.* Further, polynomials where $\kappa(\Lambda)$ is the only nonzero coefficient are the so-called *constant series* and they are usually identified with elements of the semiring. For a constant series a Cauchy's product is very simple:

$$a\Big(\sum_{w\in\Sigma^*}\kappa(w)w\Big)=\sum_{w\in\Sigma^*}a\kappa(w)w.$$

By convention, the zeroth power S^0 of the formal power series S is defined to be the constant series $\mathbf{1}$. The series where all coefficients are $=\mathbf{0}$ is the so-called *zero series* $\mathbf{0}$.

Series with $\kappa(\Lambda)=\mathbf{0}$, i.e., with a zero constant term, are called *quasi-regular.* A quasi-regular formal power series

$$S=\sum_{w\in\Sigma^*}\kappa(w)w=\sum_{w\in\Sigma^+}\kappa(w)w$$

has the so-called *quasi-inverses*

$$S^+=\sum_{w\in\Sigma^+}\kappa^+(w)w \quad\text{and}\quad S^*=\mathbf{1}+S^+$$

(note that $\kappa^+(\Lambda)=\mathbf{0}$) where

$$\kappa^+(w)=\sum_{m=1}^{|w|}\sum_{u_1u_2\cdots u_m=w}\kappa(u_1)\kappa(u_2)\cdots\kappa(u_m).$$

The basic property of quasi-inverses is then given by

Theorem 36. *The quasi-inverses of a quasi-regular formal power series S satisfy the equalities*

$$S+SS^+=S+S^+S=SS^*=S^*S=S^+.$$

Proof. To give the idea, let us prove the equality $S+SS^+=S^+$. (The other equalities are left as exercises for the reader.) We denote

$$SS^+=\sum_{w\in\Sigma^*}\kappa'(w)w.$$

According to the definitions above, then

$$\begin{aligned}\kappa'(w)&=\sum_{uv=w}\kappa(u)\sum_{m=1}^{|v|}\sum_{u_1\cdots u_m=v}\kappa(u_1)\cdots\kappa(u_m)=\sum_{uv=w}\sum_{m=1}^{|v|}\sum_{u_1\cdots u_m=v}\kappa(u)\kappa(u_1)\cdots\kappa(u_m)\\&=\sum_{k=2}^{|w|}\sum_{u_1u_2\cdots u_k=w}\kappa(u_1)\kappa(u_2)\cdots\kappa(u_k).\end{aligned}$$

The last equality can be verified by checking through the suffixes v of w. We see thus that $\kappa(w)+\kappa'(w)=\kappa^+(w)$. □

The formal power series S of a language L that does not contain the empty word is quasi-regular, and S^+ (resp. S^*) is the formal power series of L^+ (resp. L^*). Quasi-inversion thus is the "missing operation" that corresponds to concatenation closure of languages. It is, however, not defined for all power series, while concatenation closure is defined for all languages. It also behaves a bit differently, as we will see.

Even though elements in a semiring may not have opposite elements (or negatives), it is nevertheless customary to write

$$S^+ = \frac{S}{\mathbf{1} - S} \quad \text{and} \quad S^* = \frac{\mathbf{1}}{\mathbf{1} - S}.$$

Another way of expressing quasi-inverses is to use Cauchy's powers:

$$S^+ = \sum_{m=1}^{\infty} S^m \quad \text{and} \quad S^* = \sum_{m=0}^{\infty} S^m.$$

Example. *The power series of the (single) variable x, familiar from basic courses in calculus, may be thought of as formal power series of the semiring $(\mathbb{Q}, +, \cdot, 0, 1)$ (rational numbers), or $(\mathbb{R}, +, \cdot, 0, 1)$ (reals). E.g. the Maclaurin series of the function xe^x*

$$S = \sum_{m=1}^{\infty} \frac{1}{(m-1)!} x^m$$

can be interpreted as such a formal power series over the alphabet $\{x\}$. Its quasi-inverses S^+ and S^* are the Maclaurin series of the functions $xe^x/(1 - xe^x)$ and $1/(1 - xe^x)$, as one might expect.

But note that now $(S^+)^+ \neq S^+$. For languages $(L^+)^+ = L^+$, so not all rules of calculation for languages are valid for formal power series!

Sum, Cauchy's product and quasi-inversion are the so-called *rational operations* of formal power series. Those formal power series of the semiring R over the alphabet $\Sigma = \{x_1, x_2, \dots, x_k\}$ that can be obtained using rational operations starting from constants and monomials of the form x_l (i.e., variables), are the so-called *R-rational power series* over Σ. The set of all R-rational power series over Σ is denoted by $R_{\mathrm{rat}}\langle\langle\Sigma\rangle\rangle$. Considering the connections to operations of languages, and the definition of regular languages via regular expressions, we see that

Theorem 37. *The formal power series of regular languages over the alphabet Σ form exactly the set $B_{\mathrm{rat}}\langle\langle\Sigma\rangle\rangle$, that is, they are exactly all B-rational power series over Σ.*

Proof. Thinking of defining regular languages using regular expressions in Section 2.1, and quasi-inversion, all we need to show is that in the expressions concatenation closure can always be given in the form $r^* = \Lambda + r_1^+$ where the regular language corresponding to r_1 does not contain the empty word. This follows in a straightforward way from the following rules that "separate" the empty word::

$$\begin{aligned} r_1 + (\Lambda + r_2) &= \Lambda + (r_1 + r_2) \ , \\ (\Lambda + r_1) + (\Lambda + r_2) &= \Lambda + (r_1 + r_2) \ , \\ r_1(\Lambda + r_2) &= r_1 + r_1 r_2 \ , \\ (\Lambda + r_1) r_2 &= r_2 + r_1 r_2 \ , \\ (\Lambda + r_1)(\Lambda + r_2) &= \Lambda + (r_1 + r_2 + r_1 r_2) \ , \\ (\Lambda + r)^* &= r^* = \Lambda + r^+. \end{aligned}$$

Applying these rules, any regular language can be given in an equivalent form $\Lambda + r$ or r where the regular language corresponding to r does not contain the empty word. (Recall that $\Lambda^* = \emptyset^* = \Lambda$.) □

Rational formal power series are thus the counterpart of regular languages.[1]

A formal power series in $R\langle\langle\Sigma\rangle\rangle$

$$S = \sum_{w\in\Sigma^*} \kappa(w)w$$

is never too far from a language, indeed, it always determines the language

$$L(S) = \big\{w \mid \kappa(w) \neq \mathbf{0}\big\},$$

the so-called *support language* of S. The support language of the zero series (all coefficients $= \mathbf{0}$) is the empty language, for a formal polynomial it is a finite language. On the other hand, it is the extra structure in R, when compared to B, that makes formal power series an interesting generalization of languages.

Note. *It is fairly easy to see that* $\big(R\langle\langle\Sigma\rangle\rangle, +, \cdot, \mathbf{0}, \mathbf{1}\big)$*, where* $\cdot$ *is Cauchy's product, is itself a semiring—and it could be used as the coefficient semiring of a formal power series!*

9.4 Recognizable Formal Power Series. Schützenberger's Representation Theorem

For regular languages, definitions using regular expressions on the one hand, and using finite automata on the other, together form a strong machinery for proving properties of the languages. It was noted above that the family $\mathcal{R}_\Sigma$ of regular languages over Σ is respective to $R_{\mathrm{rat}}\langle\langle\Sigma\rangle\rangle$, and the latter was defined via rational operations. This corresponds to use of regular expressions. There is an automata-like characterization, too, the so-called recognizability.

In order to get a grasp of recognizability we first give, as a preamble, a representation of a nondeterministic finite automaton (without Λ-transitions) $M = (Q, \Sigma, S, \delta, A)$ via the Boolean semiring B. The stateset of the automaton is $Q = \{q_1, \dots, q_m\}$. For each symbol $x_l \in \Sigma$ there is a corresponding $m \times m$-bitmatrix $\mathbf{D}_l = (d_{ij}^{(l)})$ where

$$d_{ij}^{(l)} = \begin{cases} 1 \text{ if } q_j \in \delta(q_i, x_l) \\ 0 \text{ otherwise.} \end{cases}$$

Furthermore, for the sets of initial states and terminal states there are corresponding m-vectors (row vectors) $\mathbf{s} = (s_1, \dots, s_m)$ and $\mathbf{a} = (a_1, \dots, a_m)$ where

$$s_i = \begin{cases} 1 \text{ if } q_i \in S \\ 0 \text{ otherwise} \end{cases} \quad \text{and} \quad a_i = \begin{cases} 1 \text{ if } q_i \in A \\ 0 \text{ otherwise.} \end{cases}$$

Then the states that can be reached from some initial state after reading the input symbol x_l, are given by the elements equal to 1 in the vector $\mathbf{s}\mathbf{D}_l$. In general, those states that can be reached from some initial state after reading the input $w = x_{l_1}x_{l_2}\cdots x_{l_k}$, ie., $\hat{\delta}^*(S, w)$, are given by the elements 1 in the vector

$$\mathbf{s}\mathbf{D}_{l_1}\mathbf{D}_{l_2}\cdots\mathbf{D}_{l_k}.$$

[1] The counterpart of CF-languages, the so-called *algebraic formal power series,* can be obtained by allowing algebraic operations (i.e., solutions of polynomial equations).

For brevity, let us now denote

$$\boldsymbol{\mu}(w) = \mathbf{D}_{l_1}\mathbf{D}_{l_2}\cdots\mathbf{D}_{l_k},$$

and especially $\boldsymbol{\mu}(x_l) = \mathbf{D}_l$ and $\boldsymbol{\mu}(\Lambda) = \mathbf{I}_m$ (an identity matrix whose elements are in $\mathbb{B}$). Remember that all calculation is now in the Boolean semiring B. Associativity of the matrix product then follows from the associativity and distributivity of the operations of B.

An input w is now accepted exactly in the case when the set $\hat{\delta}^*(S, w)$ contains at least one terminal state, that is, when

$$\mathbf{s}\boldsymbol{\mu}(w)\mathbf{a}^{\mathsf{T}} = 1.$$

In particular, the empty word Λ is accepted exactly when $\mathbf{s}\mathbf{a}^{\mathsf{T}} = 1$.

The formal power series of the language $L(M)$ is thus

$$\sum_{w\in\Sigma^*} \mathbf{s}\boldsymbol{\mu}(w)\mathbf{a}^{\mathsf{T}}w.$$

Such a formal power series is called *B-recognizable.*

The definition of a recognizable formal power series over the alphabet $\Sigma = \{x_1, \dots, x_k\}$ in the case of a general semiring $R = (C, +, \cdot, \mathbf{0}, \mathbf{1})$ is quite similar. The formal power series

$$S = \sum_{w\in\Sigma^*} \kappa(w)w$$

is *R-recognizable* if there exists a number $m \geq 1$ and $m \times m$-matrices $\mathbf{D}_1, \dots, \mathbf{D}_k$ and m-vectors $\mathbf{s}$ and $\mathbf{a}$ (row vectors) such that

$$\kappa(w) = \mathbf{s}\boldsymbol{\mu}(w)\mathbf{a}^{\mathsf{T}},$$

the so-called *matrix representation*[2] of S where

- $\boldsymbol{\mu}(w)$ is an $m \times m$-matrix over R,
- $\boldsymbol{\mu}(\Lambda) = \mathbf{I}_m$ (an $m \times m$ identity matrix formed using the elements $\mathbf{0}$ and $\mathbf{1}$),
- $\boldsymbol{\mu}(x_l) = \mathbf{D}_l$ ($l = 1, \dots, k$), and
- $\boldsymbol{\mu}$ satisfies the condition

  $$\boldsymbol{\mu}(uv) = \boldsymbol{\mu}(u)\boldsymbol{\mu}(v) \quad (\text{for all } u, v \in \Sigma^*).$$

  Here, too, associativity of the matrix product follows from the associativity and distributivity of the operations of R.

A matrix representation is a kind of automaton that recognizes the series. The set of R-recognizable power series over the alphabet Σ is denoted by $R_{\mathrm{rec}}\langle\langle\Sigma\rangle\rangle$.

All operations of formal power series dealt with above preserve recognizability. Let us start with the rational operations:

[2] Note that here $\mathbf{s}\boldsymbol{\mu}(w)\mathbf{a}^{\mathsf{T}}$ is a linear combination of elements of the matrix $\boldsymbol{\mu}(w)$ with fixed coefficients obtained from the vectors $\mathbf{s}$ and $\mathbf{a}$. Indeed, the matrix representation is sometimes defined as being just a linear combination of elements of the matrices with fixed coefficients. This gives the same concept of recognizability as our definition.

Theorem 38. (i) *If the formal power series S_1 and S_2 are R-recognizable, then so are $S_1 + S_2$ and S_1S_2.*

(ii) *If the quasi-regular formal power series S is R-recognizable, then so are S^+ and S^*.*

Proof. Proofs of these are rather similar to those of Theorem 2 and Kleene's Theorem.

(i) Assume the series S_1 and S_2 are recognizable with matrix representations

$$\kappa_1(w) = \mathbf{s}_1\boldsymbol{\mu}_1(w)\mathbf{a}_1^{\mathsf{T}} \quad \text{and} \quad \kappa_2(w) = \mathbf{s}_2\boldsymbol{\mu}_2(w)\mathbf{a}_2^{\mathsf{T}},$$

respectively. A matrix representation of the sum $S_1 + S_2$

$$\gamma(w) = \mathbf{t}\boldsymbol{\nu}(w)\mathbf{b}^{\mathsf{T}}$$

is obtained by taking

$$\mathbf{t} = \left(\mathbf{s}_1 \,\middle|\, \mathbf{s}_2\right) \quad \text{and} \quad \mathbf{b} = \left(\mathbf{a}_1 \,\middle|\, \mathbf{a}_2\right)$$

and

$$\boldsymbol{\nu}(x_l) = \left(\begin{array}{c|c} \boldsymbol{\mu}_1(x_l) & \mathbf{O} \\ \hline \mathbf{O} & \boldsymbol{\mu}_2(x_l) \end{array}\right) \quad (l = 1, \dots, k)$$

(block matrix). Here the $\mathbf{O}$'s are zero matrices of appropriate sizes, formed using the zero element of R. This follows since

$$\begin{aligned}\boldsymbol{\gamma}(w) &= \left(\mathbf{s}_1 \,\middle|\, \mathbf{s}_2\right) \left(\begin{array}{c|c} \boldsymbol{\mu}_1(w) & \mathbf{O} \\ \hline \mathbf{O} & \boldsymbol{\mu}_2(w) \end{array}\right) \left(\begin{array}{c} \mathbf{a}_1^{\mathsf{T}} \\ \hline \mathbf{a}_2^{\mathsf{T}} \end{array}\right) = \left(\mathbf{s}_1 \,\middle|\, \mathbf{s}_2\right) \left(\begin{array}{c} \boldsymbol{\mu}_1(w)\mathbf{a}_1^{\mathsf{T}} \\ \hline \boldsymbol{\mu}_2(w)\mathbf{a}_2^{\mathsf{T}} \end{array}\right) \\ &= \kappa_1(w) + \kappa_2(w).\end{aligned}$$

The case of the Cauchy product S_1S_2 is somewhat more complicated. Let us denote $\mathbf{C} = \mathbf{a}_1^{\mathsf{T}}\mathbf{s}_2$. We take now

$$\mathbf{t} = \left(\mathbf{s}_1 \,\middle|\, \mathbf{0}\right) \quad \text{and} \quad \mathbf{b} = \left(\mathbf{a}_2\mathbf{C}^{\mathsf{T}} \,\middle|\, \mathbf{a}_2\right),$$

where $\mathbf{0}$ is a zero vector of the same size as $\mathbf{a}_2$, and

$$\boldsymbol{\nu}(x_l) = \left(\begin{array}{c|c} \boldsymbol{\mu}_1(x_l) & \mathbf{C}\boldsymbol{\mu}_2(x_l) \\ \hline \mathbf{O} & \boldsymbol{\mu}_2(x_l) \end{array}\right) \quad (l = 1, \dots, k).$$

Now

$$\boldsymbol{\nu}(w) = \left(\begin{array}{c|c} \boldsymbol{\mu}_1(w) & \boldsymbol{\eta}(w) \\ \hline \mathbf{O} & \boldsymbol{\mu}_2(w) \end{array}\right)$$

where

$$\boldsymbol{\eta}(w) = \sum_{\substack{uv=w \\ v\neq\Lambda}} \boldsymbol{\mu}_1(u)\mathbf{C}\boldsymbol{\mu}_2(v).$$

This is shown by induction on the length of w. Clearly the result is true when $w = \Lambda$ (the sum is empty and $\boldsymbol{\eta}(\Lambda) = \mathbf{O}$). On the other hand, the upper right hand block of the matrix $\boldsymbol{\nu}(wx_l)$ is

$$\begin{aligned}\boldsymbol{\mu}_1(w)\mathbf{C}\boldsymbol{\mu}_2(x_l) + \left(\sum_{\substack{uv=w \\ v\neq\Lambda}} \boldsymbol{\mu}_1(u)\mathbf{C}\boldsymbol{\mu}_2(v)\right)\boldsymbol{\mu}_2(x_l) &= \boldsymbol{\mu}_1(w)\mathbf{C}\boldsymbol{\mu}_2(x_l) + \sum_{\substack{uv=w \\ v\neq\Lambda}} \boldsymbol{\mu}_1(u)\mathbf{C}\boldsymbol{\mu}_2(vx_l) \\ &= \sum_{\substack{uv=wx_l \\ v\neq\Lambda}} \boldsymbol{\mu}_1(u)\mathbf{C}\boldsymbol{\mu}_2(v).\end{aligned}$$

The matrix representation

$$\begin{aligned}\mathbf{t}\boldsymbol{\nu}(w)\mathbf{b}^{\mathsf{T}} &= \mathbf{s}_1\boldsymbol{\mu}_1(w)\mathbf{C}\mathbf{a}_2^{\mathsf{T}} + \sum_{\substack{uv=w\\ v\neq\Lambda}} \mathbf{s}_1\boldsymbol{\mu}_1(u)\mathbf{C}\boldsymbol{\mu}_2(v)\mathbf{a}_2^{\mathsf{T}} = \sum_{uv=w} \mathbf{s}_1\boldsymbol{\mu}_1(u)\mathbf{a}_1^{\mathsf{T}}\mathbf{s}_2\boldsymbol{\mu}_2(v)\mathbf{a}_2^{\mathsf{T}}\\ &= \sum_{uv=w} \kappa_1(u)\kappa_2(v)\end{aligned}$$

is then the matrix representation of the Cauchy product.

(ii) Assume the quasi-regular formal power series S is recognizable with a matrix representation

$$\kappa(w) = \mathbf{s}\boldsymbol{\mu}(w)\mathbf{a}^{\mathsf{T}}$$

where $\mathbf{s}\mathbf{a}^{\mathsf{T}} = \kappa(\Lambda) = \mathbf{0}$. We denote again $\mathbf{C} = \mathbf{a}^{\mathsf{T}}\mathbf{s}$ whence

$$\mathbf{s}\mathbf{C} = \mathbf{0}.$$

A matrix representation for the series S^+ is now

$$\mathbf{s}\boldsymbol{\mu}^+(w)\mathbf{a}^{\mathsf{T}}$$

where

$$\boldsymbol{\mu}^+(x_l) = \boldsymbol{\mu}(x_l) + \mathbf{C}\boldsymbol{\mu}(x_l) \quad (l = 1, \dots, k).$$

This representation is evidently correct when $w = \Lambda$. For a nonempty word $w = x_{l_1}x_{l_2}\cdots x_{l_n}$ we get, multiplying out,

$$\begin{aligned}\mathbf{s}\boldsymbol{\mu}^+(w)\mathbf{a}^{\mathsf{T}} &= \mathbf{s}\big(\boldsymbol{\mu}(x_{l_1}) + \mathbf{C}\boldsymbol{\mu}(x_{l_1})\big)\big(\boldsymbol{\mu}(x_{l_2}) + \mathbf{C}\boldsymbol{\mu}(x_{l_2})\big)\cdots\big(\boldsymbol{\mu}(x_{l_n}) + \mathbf{C}\boldsymbol{\mu}(x_{l_n})\big)\mathbf{a}^{\mathsf{T}}\\ &= \sum_{m=1}^{|w|} \sum_{u_1u_2\cdots u_m=w} \mathbf{s}\boldsymbol{\mu}(u_1)\mathbf{C}\boldsymbol{\mu}(u_2)\mathbf{C}\cdots\mathbf{C}\boldsymbol{\mu}(u_m)\mathbf{a}^{\mathsf{T}}.\end{aligned}$$

Indeed, in the expanded product terms containing $\mathbf{sC}$ will be zero elements and can be omitted, leaving exactly the terms appearing in the sum. Hence

$$\mathbf{s}\boldsymbol{\mu}^+(w)\mathbf{a}^{\mathsf{T}} = \sum_{m=1}^{|w|} \sum_{u_1u_2\cdots u_m=w} \kappa(u_1)\kappa(u_2)\cdots\kappa(u_m) = \kappa^+(w)$$

and the matrix representation of the quasi-inverse S^+ is correct.

The constant series $\mathbf{1}$ is recognizable (see below), and so is then the quasi-inverse $S^* = \mathbf{1} + S^+$. □

As an immediate consequence we get

Corollary. *R-rational formal power series are R-recognizable.*

Proof. By the previous theorem and the definition of rationality all we need to show is that constant series and the monomials x_l are recognizable. For constant series this is trivial, the matrix representation of the constant a is

$$\kappa(w) = a\mu(w)\mathbf{1}$$

where $\mu(x_l) = \mathbf{0}$ $(l = 1, \dots, k)$. Recall that $\mathbf{0}^0 = \mathbf{1}$. The matrix representation of the monomial x_l is

$$\kappa(w) = \begin{pmatrix}\mathbf{1} & \mathbf{0}\end{pmatrix}\boldsymbol{\mu}(w)\begin{pmatrix}\mathbf{0}\\ \mathbf{1}\end{pmatrix}$$

where

$$\boldsymbol{\mu}(x_l) = \begin{pmatrix}\mathbf{0} & \mathbf{1}\\ \mathbf{0} & \mathbf{0}\end{pmatrix} \quad\text{and}\quad \boldsymbol{\mu}(x_i) = \begin{pmatrix}\mathbf{0} & \mathbf{0}\\ \mathbf{0} & \mathbf{0}\end{pmatrix} \quad (\text{for } i \neq l).$$ □

The converse result holds true, too, i.e., R-recognizable formal power series are R-rational. The proof of this is similar to the proof of Theorem 3, and is based on the fact that $m \times m$-matrices over the semiring R form a semiring, too, when the operations are sum and product of matrices, and the zero element and the identity element are the zero matrix $\mathbf{O}_m$ and the identity matrix $\mathbf{I}_m$ over R. This semiring is denoted by $R^{m\times m}$.

Formal power series of $R^{m\times m}\langle\langle\Sigma\rangle\rangle$ may be interpreted either as formal power series with $m \times m$-matrix coefficients (the usual interpretation), or as $m \times m$-matrices whose elements are formal power series in $R\langle\langle\Sigma\rangle\rangle$. The matrix product of matrices with power series elements is then the same as the Cauchy product of the corresponding power series with matrix coefficients (why?).

To prove the converse we need the following technical lemma.

Lemma. *Assume $\mathbf{P} \in R^{m\times m}\langle\langle\Sigma\rangle\rangle$ is quasi-regular and $\mathbf{Z}$ and $\mathbf{Q}$ are m-vectors (row vectors) over $R\langle\langle\Sigma\rangle\rangle$. Then the only solution of the equation*

$$\mathbf{Z} = \mathbf{Q} + \mathbf{Z}\mathbf{P}$$

is $\mathbf{Z} = \mathbf{Q}\mathbf{P}^*$. Furthermore, if the elements of $\mathbf{P}$ and $\mathbf{Q}$ are R-rational, then so are the elements of $\mathbf{Z}$.

Proof. By Theorem 36, $\mathbf{P}^* = \mathbf{I}_m + \mathbf{P}^*\mathbf{P}$ so that $\mathbf{Z} = \mathbf{Q}\mathbf{P}^*$ indeed is a solution of the equation. On the other hand, the solution satisfies also the equations

$$\mathbf{Z} = \mathbf{Q}\sum_{i=0}^{n-1}\mathbf{P}^i + \mathbf{Z}\mathbf{P}^n \quad (n = 1, 2, \dots),$$

obtained by "iteration", where the "remainder term" $\mathbf{Z}\mathbf{P}^n$ contains all terms of (word) length $\geq n$ (remember that $\mathbf{P}$ is quasi-regular). The solution is thus unique.

Let us then assume that the elements of $\mathbf{P}$ and $\mathbf{Q}$ are R-rational power series, and show that so are the elements of the solution $\mathbf{Z}$, using induction on m.

The Induction Basis, the case $m = 1$, is obvious. Let us then assume (Induction Hypothesis) that the claimed result is correct when $m = l - 1$, and consider the case $m = l$ (Induction Statement). For the proof of the Induction Statement let us denote

$$\mathbf{Z} = (Z_1, \dots, Z_l) \quad \text{and} \quad \mathbf{Q} = (Q_1, \dots, Q_l),$$

and $\mathbf{P} = (P_{ij})$. Then

$$Z_l = R_l + Z_l P_{ll}$$

where

$$R_l = Q_l + Z_1 P_{1l} + \cdots + Z_{l-1}P_{l-1,l}.$$

Clearly R_l is an R-rational power series if $Z_1, \dots, Z_{l-1}$ are such. By the above

$$Z_l = \begin{cases} R_l \text{ if } P_{ll} = \mathbf{0} \\ R_l P_{ll}^* \text{ if } P_{ll} \neq \mathbf{0}. \end{cases}$$

(Note that P_{ll} is quasi-regular.) Substituting the Z_l thus obtained into the equation $\mathbf{Z} = \mathbf{Q} + \mathbf{Z}\mathbf{P}$ we get an equation of the lower dimension $m = l - 1$ whose matrix of coefficients is a quasi-regular formal power series in $R^{(l-1)\times(l-1)}\langle\langle\Sigma\rangle\rangle$. By the Induction Hypothesis, the elements of its solution $Z_1, \dots, Z_{l-1}$ are R-rational power series. Hence Z_l, too, is R-rational. □

We then get the following famous result, the equivalent of Kleene's Theorem for languages:

Schützenberger's Representation Theorem. *R-rational formal power series are exactly all R-recognizable formal power series.*

Proof. It remains to be proved that an R-recognizable formal power series over the alphabet $\Sigma = \{x_1, \dots, x_k\}$, with the $m \times m$-matrix representation

$$S = \sum_{w \in \Sigma^*} \mathbf{s}\boldsymbol{\mu}(w)\mathbf{a}^{\mathsf{T}} w,$$

is R-rational. The formal power series (polynomial)

$$\mathbf{P} = \boldsymbol{\mu}(x_1)x_1 + \cdots \boldsymbol{\mu}(x_k)x_k$$

of the semiring $R^{m \times m}\langle\!\langle \Sigma \rangle\!\rangle$ is quasi-regular and its elements are R-rational. Furthermore

$$\mathbf{P}^* = \sum_{w \in \Sigma^*} \boldsymbol{\mu}(w)w \quad \text{and} \quad S = \mathbf{s}\mathbf{P}^*\mathbf{a}^{\mathsf{T}}.$$

By the Lemma $\mathbf{Z} = \mathbf{s}\mathbf{P}^*$ is the only solution of the equation $\mathbf{Z} = \mathbf{s} + \mathbf{Z}\mathbf{P}$ and its elements are R-rational, hence $S = \mathbf{s}\mathbf{P}^*\mathbf{a}^{\mathsf{T}}$ is R-rational, too. □

9.5 Recognizability and Hadamard's Product

Hadamard's products of recognizable formal power series are always recognizable, too. This can be shown fairly easily using Kronecker's product of matrices.

Kronecker's product[3] of the matrices $\mathbf{A} = (a_{ij})$ (an $n_1 \times m_1$-matrix) and $\mathbf{B} = (b_{ij})$ (an $n_2 \times m_2$-matrix) is the $n_1 n_2 \times m_1 m_2$-matrix

$$\mathbf{A} \otimes \mathbf{B} = \left(\begin{array}{c|c|c|c} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1m_1}\mathbf{B} \\ \hline a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2m_1}\mathbf{B} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline a_{n_1 1}\mathbf{B} & a_{n_1 2}\mathbf{B} & \cdots & a_{n_1 m_2}\mathbf{B} \end{array}\right)$$

(block matrix). A special case is Kronecker's product of two vectors (take $n_1 = n_2 = 1$ or $m_1 = m_2 = 1$). The following basic properties of Kronecker's product are easily verified. It is assumed here that all appearing matrix products are well-defined.

1. Associativity:
$$(\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} = \mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C})$$
As a consequence chained Kronecker's products can be written without parentheses.

2. Matrix multiplication of Kronecker's products (this follows more or less directly from multiplication of block matrices):
$$(\mathbf{A}_1 \otimes \mathbf{B}_1)(\mathbf{A}_2 \otimes \mathbf{B}_2) = (\mathbf{A}_1\mathbf{A}_2) \otimes (\mathbf{B}_1\mathbf{B}_2)$$

[3] Often called *tensor product.*

3. The Kronecker product of two identity matrices is again an identity matrix.
4. Inverse of a Kronecker product (follows from the above rule for matrix multiplication):
$$(\mathbf{A}\otimes\mathbf{B})^{-1}=\mathbf{A}^{-1}\otimes\mathbf{B}^{-1}$$
5. Transposition of a Kronecker product (follows directly from transposition of block matrices):
$$(\mathbf{A}\otimes\mathbf{B})^{\mathsf{T}}=\mathbf{A}^{\mathsf{T}}\otimes\mathbf{B}^{\mathsf{T}}$$
A similar formula holds for conjugate-transposition of complex matrices:
$$(\mathbf{A}\otimes\mathbf{B})^{\dagger}=\mathbf{A}^{\dagger}\otimes\mathbf{B}^{\dagger}$$
6. Kronecker's products of unitary matrices are unitary. (Follows from the above.)

Note especially that Kronecker's product of 1×1-matrices is simply a scalar multiplication. Thus, the matrix representations of the formal power series S_1 ja S_2

$$\kappa_1(w)=\mathbf{s}_1\boldsymbol{\mu}_1(w)\mathbf{a}_1^{\mathsf{T}}\quad\text{and}\quad\kappa_2(w)=\mathbf{s}_2\boldsymbol{\mu}_2(w)\mathbf{a}_2^{\mathsf{T}},$$

respectively, can now be multiplied using Kronecker's product:

$$\begin{aligned}\kappa_1(w)\kappa_2(w)=\kappa_1(w)\otimes\kappa_2(w)&=\big(\mathbf{s}_1\boldsymbol{\mu}_1(w)\mathbf{a}_1^{\mathsf{T}}\big)\otimes\big(\mathbf{s}_2\boldsymbol{\mu}_2(w)\mathbf{a}_2^{\mathsf{T}}\big)\\&=(\mathbf{s}_1\otimes\mathbf{s}_2)\big(\boldsymbol{\mu}_1(w)\otimes\boldsymbol{\mu}_2(w)\big)(\mathbf{a}_1^{\mathsf{T}}\otimes\mathbf{a}_2^{\mathsf{T}}).\end{aligned}$$

A matrix representation $\kappa(w)=\mathbf{s}\boldsymbol{\mu}(w)\mathbf{a}^{\mathsf{T}}$ for the Hadamard product $S_1\otimes S_2$ is thus obtained by taking
$$\mathbf{s}=\mathbf{s}_1\otimes\mathbf{s}_2\quad\text{and}\quad\mathbf{a}=\mathbf{a}_1\otimes\mathbf{a}_2$$
and
$$\boldsymbol{\mu}(x_l)=\boldsymbol{\mu}_1(x_l)\otimes\boldsymbol{\mu}_2(x_l)\quad(l=1,\dots,k).$$
Indeed, the matrix multiplication rule then gives us
$$\boldsymbol{\mu}(w)=\boldsymbol{\mu}_1(w)\otimes\boldsymbol{\mu}_2(w).$$

We thus get

Theorem 39. *Hadamard's products of R-recognizable formal power series are again R-recognizable.*

By Schützenberger's Representation Theorem we get further

Corollary. *Hadamard's products of R-rational formal power series are again R-rational.*

9.6 Examples of Formal Power Series

9.6.1 Multilanguages

Let us return to the representation of a nondeterministic finite automaton (without λ-transitions) $M=(Q,\Sigma,S,\delta,A)$ in Section 9.4. Recall that the stateset of the automaton was $\{q_1,\dots,q_m\}$. Here, however, the semiring is $N=(\mathbb{N},+,\cdot,0,1)$, and not B. The automaton is then considered as a *multiautomaton.* This means that

- the value $\delta(q_i, x_l, q_j)$ of the transition function—note the three arguments—tells in how many ways the automaton can move from the state q_i to the state q_j after reading x_l. This value may be $= 0$, meaning that there is no transition from q_i to q_j after reading x_l.

- the set S of initial states and the set A of terminal states are both multisets, i.e., a state can occur many times in them.

In the matrix representation the symbol $x_l \in \Sigma$ corresponds to an $m \times m$-matrix $\boldsymbol{\mu}(x_l) = \mathbf{D}_l = (d_{ij}^{(l)})$ with integral elements where $d_{ij}^{(l)} = \delta(q_i, x_l, q_j)$. The sets of initial and terminal states correspond to the row vectors $\mathbf{s} = (s_1, \dots, s_m)$ and $\mathbf{a} = (a_1, \dots, a_m)$ with integral elements where s_i tells how many times q_i is an initial state and a_i how many times q_i is a terminal state. If $s_i = 0$, then q_i is not an initial state, and similarly, if $a_i = 0$, then q_i is not a terminal state.

The number of ways of reaching each state from some initial state after reading x_l can then be seen in the vector $\mathbf{s}\mathbf{D}_l$. In general, the number of ways of reaching various states starting from initial states reading the input $w = x_{l_1} x_{l_2} \cdots x_{l_k}$ appear in the vector

$$\mathbf{s}\mathbf{D}_{l_1}\mathbf{D}_{l_2} \cdots \mathbf{D}_{l_k} = \mathbf{s}\boldsymbol{\mu}(w).$$

Hence the number of ways of accepting the word w—including the value 0 when the word is not accepted at all—is

$$\mathbf{s}\boldsymbol{\mu}(w)\mathbf{a}^{\mathsf{T}} = \kappa(w).$$

The formal power series

$$S(M) = \sum_{w \in \Sigma^*} \kappa(w) w,$$

the so-called *multilanguage* recognized by M, is then N-recognizable, and by Schützenberger's Representation Theorem also N-rational. $\kappa(w)$ is the so-called *multiplicity* of the word w.

Every N-recognizable, and thus also N-rational, multilanguage can thus be interpreted as the multilanguage recognized by a finite multiautomaton. In general formal power series in $N\langle\langle\Sigma\rangle\rangle$ may be thought of as multilanguages over the alphabet Σ, but the situation may not then have so straightforward an interpretation.

9.6.2 Stochastic Languages

A row vector $\mathbf{v}$ with real elements is called *stochastic* if the elements are ≥ 0 and sum to $= 1$. A real matrix is *stochastic* if its rows are stochastic vectors.

Let us now transform the finite automaton of Section 9.4 to a *stochastic finite automaton* by adding a probabilistic behaviour:

- The initial state vector is replaced by a stochastic vector $\mathbf{s} = (s_1, \dots, s_m)$ where s_i is the probability $\mathsf{P}(q_i$ is an initial state).

- The terminal state vector is replaced by the vector of terminal state probabilities $\mathbf{a} = (a_1, \dots, a_m)$ (not necessarily stochastic!) where a_i is the probability $\mathsf{P}(q_i$ is a terminal state).

- The state transition matrices are replaced by the stochastic matrices of state transition probabilities $\mathbf{D}_l = (d_{ij}^{(l)})$ where

$$d_{ij}^{(l)} = \mathsf{P}(\text{the next state is } q_j \text{ when } x_l \text{ is read} \mid \text{the present state is } q_i),$$

that is, the conditional probability that after reading x_l the next state is q_j when the present state is known to be q_i.

According to the rules of probability calculus

$$\mathsf{P}(\text{the next state is } q_j \text{ when } x_l \text{ is read}) = \sum_{i=1}^{m} d_{ij}^{(l)} \mathsf{P}(\text{the present state is } q_i).$$

Thus the vector $\mathbf{sD}_l$ gives the state probabilities when the first input symbol x_l is read. In general, the state probabilities after reading the input w appear in the vector

$$\mathbf{s}\boldsymbol{\mu}(w).$$

(In mathematical terms, the state transition chain is a so-called Markov process.) Rules of probability calculus give further

$$\begin{aligned}&\mathsf{P}(\text{the state after reading the input word } w \text{ is terminal})\\ &= \sum_{i=1}^{m} \mathsf{P}(\text{the state after reading the input } w \text{ is } q_i)\mathsf{P}(q_i \text{ is a terminal state})\\ &= \mathbf{s}\boldsymbol{\mu}(w)\mathbf{a}^{\mathsf{T}} = \kappa(w).\end{aligned}$$

Thus we get the formal power series

$$P = \sum_{w \in \Sigma^*} \kappa(w)w,$$

recognizable in the semiring $(\mathbb{R}_+, +, \cdot, 0, 1)$, where $\kappa(w)$ is the probability of the word w belonging in the language the language. This power series is a so-called *stochastic language.*

Obviously stochastic languages are not closed under sum, Cauchy's product or quasi-inversion since stochasticity requires that $\kappa(w) \leq 1$. We have, however,

Theorem 40. *Stochastic languages are closed under Hadamard's product.*

Proof. Based on the construction in Section 9.5, it remains to just show that the Kronecker product of two stochastic vectors is stochastic, and that this is true for stochastic matrices as well. Nonnegativity of elements is clearly preserved by Kronecker's products. For the m-vector $\mathbf{v}$ and $m \times m$-matrix $\mathbf{M}$ the stochasticity conditions can be written as

$$\mathbf{v1}_m = 1 \quad \text{and} \quad \mathbf{M1}_m = \mathbf{1}_m$$

where $\mathbf{1}_m$ is the m-vector (column vector) all elements of which are $= 1$. Now, if $\mathbf{v}_1$ and $\mathbf{v}_2$ are stochastic vectors of dimensions m_1 and m_2, respectively, then

$$(\mathbf{v}_1 \otimes \mathbf{v}_2)\mathbf{1}_{m_1m_2} = (\mathbf{v}_1\mathbf{1}_{m_1}) \otimes (\mathbf{v}_2\mathbf{1}_{m_2}) = 1 \cdot 1 = 1$$

(note that $\mathbf{1}_{m_1} \otimes \mathbf{1}_{m_2} = \mathbf{1}_{m_1m_2}$). Thus the vector $\mathbf{v}_1 \otimes \mathbf{v}_2$ is stochastic. In an analogous way it is shown that the Kronecker product of two stochastic matrices is stochastic (try it!). □

Any stochastic language P gives rise to "usual languages" when thresholds for the probabilities are set, e.g.

$$\{w \mid \kappa(w) > 0.5\}$$

is such a language. These languages are called stochastic, too. For stochastic languages of this latter "threshold type" there is a famous characterization result. They can be written in the form

$$\{w \mid \kappa'(w) > 0\}$$

where

$$\sum_{w \in \Sigma^*} \kappa'(w)w$$

is an R-rational formal power series in the semiring $R = \{\mathbb{R}, +, \cdot, 0, 1\}$ of real numbers (with the usual arithmetic operations). Conversely, any language of this form is stochastic. This characterization result is known as *Turakainen's Theorem.*[4]

9.6.3 Length Functions

The *length function* of the language L is the mapping

$$\lambda_L(n) = \text{number of words of length } n \text{ in } L.$$

The corresponding formal power series over the unary alphabet $\{x\}$ in the semiring $N = (\mathbb{N}, +, \cdot, 0, 1)$ is

$$S_L = \sum_{n=0}^{\infty} \lambda_L(n) x^n.$$

Theorem 41. *For a regular language L the formal power series S_L is N-rational (and thus also N-recognizable).*

Proof. Consider a deterministic finite automaton $M = (Q, \Sigma, q_0, \delta, A)$ recognizing L. We transform M to a finite multiautomaton $M' = \big(Q, \{x\}, S, \delta', B\big)$ as follows. The state transition number

$$\delta'(q_i, x, q_j) = n$$

holds exactly in the case when there are n symbols x_l such that $\delta(q_i, x_l) = q_j$. In particular, $\delta'(q_i, x, q_j) = 0$ if and only if $\delta(q_i, x_l) \neq q_j$ for all symbols x_l. Further, in S there is only the initial state q_0 of M with multiplicity 1, and in B only the states of A appear each with multiplicity 1.

The multilanguage $S(M')$ apparently then is the power series S_L. □

9.6.4 Quantum Languages

A *quantum automaton* in the alphabet $\Sigma = \{x_1, \dots, x_l\}$ is obtained by taking a positive integer m, an m-vector (row vector) $\mathbf{s} = (s_1, \dots, s_m)$ with complex entries, a so-called *initial state* satisfying

$$\|\mathbf{s}\|^2 = \mathbf{s}\mathbf{s}^\dagger = \sum_{i=1}^{m} |s_i|^2 = 1,$$

[4] The original article reference is Turakainen, P.: Generalized Automata and Stochastic Languages. *Proceedings of the American Mathematical Society* **21** (1969), 303–309. Prof. Paavo Turakainen is a well-known Finnish mathematician.

and finally complex *state transition matrices*

$$\mathbf{U}_l = \boldsymbol{\mu}(x_l)$$

corresponding to the symbols in Σ. The state transition matrices should be unitary, i.e., always $\mathbf{U}_l^{-1} = \mathbf{U}_l^{\dagger}$, otherwise state transition is not a meaningful quantum-physical operation.

The *quantum language*[5], corresponding to the quantum automaton above, is the formal power series

$$\mathbf{Q} = \sum_{w \in \Sigma^*} \boldsymbol{\kappa}(w) w$$

where

$$\boldsymbol{\kappa}(w) = \mathbf{s}\boldsymbol{\mu}(w)$$

is the *state* of the quantum automaton after reading the word w. Again here $\boldsymbol{\mu}$ is determined by the matrices $\boldsymbol{\mu}(x_l) = \mathbf{U}_l$ and the rules $\boldsymbol{\mu}(\Lambda) = \mathbf{I}_m$ and $\boldsymbol{\mu}(uv) = \boldsymbol{\mu}(u)\boldsymbol{\mu}(v)$. It should be noted that this $\mathbf{Q}$ really is not a formal power series since its coefficients are vectors. Any of its components on the other hand is a recognizable formal power series in the semiring $(\mathbb{C}, +, \cdot, 0, 1)$ (complex numbers and the usual arithmetic operations).

As was the case for stochastic languages, quantum languages have rather poor closure properties. Again as stochastic languages, quantum languages are closed under Hadamard's product because

$$(\mathbf{s}_1 \otimes \mathbf{s}_2)(\mathbf{s}_1 \otimes \mathbf{s}_2)^{\dagger} = (\mathbf{s}_1 \otimes \mathbf{s}_2)(\mathbf{s}_1^{\dagger} \otimes \mathbf{s}_2^{\dagger}) = (\mathbf{s}_1\mathbf{s}_1^{\dagger}) \otimes (\mathbf{s}_2\mathbf{s}_2^{\dagger}) = 1 \cdot 1 = 1$$

and Kronecker's product preserves unitarity, cf. the previous section. Indeed, Hadamard's product corresponds physically to the important operation of combining quantum registers to longer registers, cf. e.g. Nielsen & Chuang. Naturally here Hadamard's product is given via Kronecker's products of coefficient vectors.

The terminal states of the construction are in some sense obtained by including a final quantum physical *measuring* of the state. The state $\boldsymbol{\kappa}(w)$ is then multiplied by some projection matrix $\mathbf{P}$.[6] Multiplication by $\mathbf{P}$ projects the vector orthogonally to a certain linear subspace H of $\mathbb{C}^m$. According to the so-called *probability interpretation* of quantum physics then

$$\big\|\boldsymbol{\kappa}(w)\mathbf{P}\big\|^2$$

gives the probability that after the measuring $\boldsymbol{\kappa}(w)$ is found to be in H. Note that $\boldsymbol{\mu}(w)$ is a unitary matrix and that

$$\big\|\boldsymbol{\kappa}(w)\mathbf{P}\big\|^2 \leq \big\|\boldsymbol{\kappa}(w)\big\|^2 = \mathbf{s}\boldsymbol{\mu}(w)\boldsymbol{\mu}(w)^{\dagger}\mathbf{s}^{\dagger} = \mathbf{s}\mathbf{s}^{\dagger} = 1.$$

The purpose of the measuring is to find out whether or not the state is in the subspace H. A physical measuring operation always gives a yes/no answer. A positive answer might then be thought as accepting the input in some sense. It is known that the languages recognized in this way are in fact exactly all regular languages. The advantage of quantum

[5] The original article refence is Moore,C. & Crutchfield, J.P.: Quantum Automata and Quantum Grammars. *Theoretical Computer Science* **237** (2000), 257–306.

[6] A projection matrix $\mathbf{P}$ is always self-adjunct, i.e., $\mathbf{P}^{\dagger} = \mathbf{P}$, and idempotent, i.e., $\mathbf{P}^2 = \mathbf{P}$. Moreover, always $\|\mathbf{x}\mathbf{P}\| \leq \|\mathbf{x}\|$, since orthogonal projections cannot increase length.

automata over the usual "classical" deterministic finite automata is that they may have far fewer states.[7]

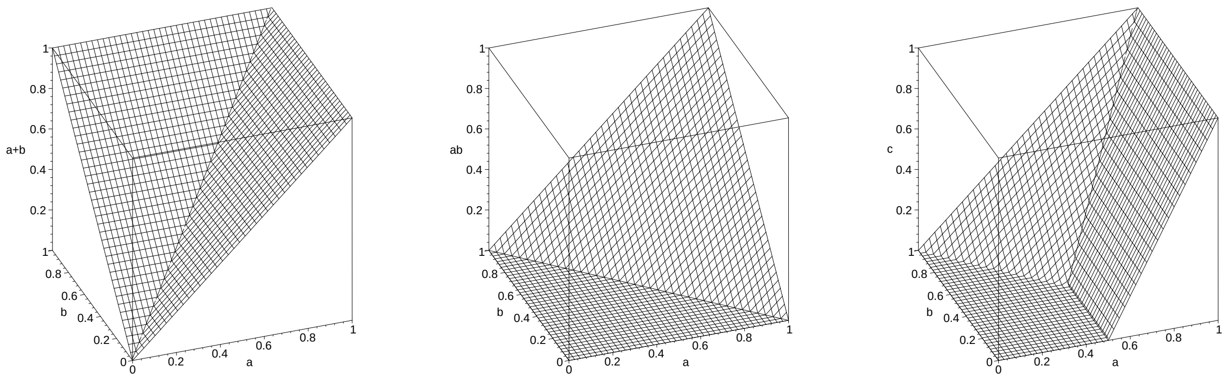
Considered as a kind of sequential machines and combined using Hadamard's product, quantum automata are considerably faster than classical computers, the latter, too, really being large (generalized) sequential machines. So much faster, in fact, that they could be used to break most generally used cryptosystems, cf. the course Mathematical Cryptology and Nielsen & Chuang. So far, however, only some fairly small quantum computers have been physically constructed.

9.6.5 Fuzzy Languages

Changing the "usual" two-valued logic to some many-valued logic, gives a variety of Boolean-like semirings.[8] As an example we take the simple semiring $S = \big\{[0,1], +, \cdot, 0, 1\big\}$ where $[0,1]$ is a real interval and the operations are defined by

$$a + b = \max(a, b) \quad \text{and} \quad a \cdot b = \max(0, a + b - 1).$$

The graphs (surfaces) of these operations are depicted below (by Maple), the third picture is the graph of the expression $c = a \cdot \big(a + b \cdot (1 + a)\big)$.



It is easily seen that S indeed is a semiring (try it!). Taking only the elements 0 and 1 returns us to the usual Boolean semiring B.

For each word w the number $\kappa(w)$ is some kind of "degree of membership" of the word in the language. The matrix representation of an S-recognizable formal power series in turn gives the corresponding *fuzzy automaton* whose working resembles that of a stochastic automaton.

[7] Se e.g. Ambainis, A, & Nahimovs, N.: Improved Constructions of Quantum Automata. *Theoretical Computer Science* **410** (2009), 1916–1922 and Freivalds, R. & Ozolsa, M. & Mančinskaa, L.: Improved Constructions of Mixed State Quantum Automata. *Theoretical Computer Science* **410** (2009), 1923–1931.

[8] In fact, any so-called MV-algebra determined by a many-valued logic gives several such semirings, see the course Applied Logics.

REFERENCES

1. Berstel, J. & Perrin, D. & Reutenauer, C.: *Codes and Automata.* Cambridge University Press (2008)

2. Berstel, J. & Reutenauer, C.: *Noncommutative Rational Series with Applications* (2008) (Available online)

3. Harrison, M.A.: *Introduction to Formal Language Theory.* Addison–Wesley (1978)

4. Hopcroft, J.E. & Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation.* Addison–Wesley (1979)

5. Hopcroft, J.E. & Motwani, R. & Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation.* Addison–Wesley (2006)

6. Kuich, W. & Salomaa, A.: *Semirings, Automata, Languages.* Springer–Verlag (1986)

7. Lewis, H.R. & Papadimitriou, C.H.: *Elements of the Theory of Computation.* Prentice–Hall (1998)

8. Martin, J.C.: *Introduction to Languages and the Theory of Computation.* McGraw–Hill (2002)

9. McEliece, R.J.: *The Theory of Information and Coding.* Cambridge University Press (2004)

10. Meduna, A.: *Automata and Languages. Theory and Applications.* Springer–Verlag (2000)

11. Nielsen, M.A. & Chuang, I.L.: *Quantum Computation and Quantum Information.* Cambridge University Press (2000)

12. Rozenberg, G. & Salomaa, A. (Eds.): *Handbook of Formal Languages. Vol. 1. Word, Language, Grammar.* Springer–Verlag (1997)

13. Salomaa, A.: *Formal Languages.* Academic Press (1973)

14. Salomaa, A.: *Jewels of Formal Language Theory.* Computer Science Press (1981)

15. Salomaa, A. & Soittola, M.: *Automata-Theoretic Aspects of Formal Power Series.* Springer–Verlag (1978)

16. Shallit, J.: *A Second Course in Formal Languages and Automata Theory.* Cambridge University Press (2008)

17. Simovici, D.A. & Tenney, R.L.: *Theory of Formal Languages with Applications.* World Scientific (1999)

18. Sipser, M.: *Introduction to the Theory of Computation.* Course Technology (2005)

Index