TCS-TR-B-07-02

TCS Technical Report

Course Notes on Theory of Computation

by

Thomas Zeugmann

Division of Computer Science

Report Series B

August 15, 2007

Hokkaido University
Graduate School of
Information Science and Technology

Email: thomas@ist.hokudai.ac.jp

Phone: +81-011-706-7684
Fax: +81-011-706-7684

Contents

Abstract

The main purpose of this course is an introductory study of the formal relationships between machines, languages and grammars. The course covers regular languages, context-free languages and touches context-sensitive languages as well as recursive and recursively enumerable languages. Relations to and applications in UNIX are discussed, too.

Moreover, we provide a framework to study the most general models of computation. These models comprise Turing machines and partial recursive functions. This allows us to reason precisely about computation and to prove mathematical theorems about its capabilities and limitations. In particular, we present the universal Turing machine which enables us to think about the capabilities of computers in a technology-independent manner.

There will be a midterm problem set and a final report problem set each worth 100 points. So your grade will be based on these 200 points.

Note that the course is demanding. But this is just in line with William S. Clark's encouragement

`Boys, be ambitious !`

Of course, nowadays, we would reformulate this encouragement as

`Girls and Boys, be ambitious !`

Recommended Literature

The references given below are *mandatory*.

(1) **丸岡章**： 計算理論とオートマトン言語理論、 Information & Computing – 106, サイエンス社, 2005
ISBN 978-4-7819-1104-5

(2) **有川節夫、宮野悟**: オートマトンと計算可能性、培風館, 1986
ISBN 4-563-00789-7

(3) **J。ホップクロフト、J。ウルマン**： オートマトン言語理論　計算論 I, Information & Computing – 3, サイエンス社, 1984
ISBN 4-7819-0374-6

(4) **J。ホップクロフト、J。ウルマン**： オートマトン言語理論　計算論 II, Information & Computing – 3, サイエンス社, 1984
ISBN 4-7819-0432-7

There are some additional references to the literature given in some lectures. So please look there, too.

Part 1: Formal Languages

Lecture 1: Introducing Formal Languages

1.1. Introduction

This course is about the study of a fascinating and important subject: ***the theory of computation***. It comprises the fundamental mathematical properties of computer hardware, software, and certain applications thereof. We are going to determine what can and cannot be computed. If it can, we also seek to figure out on which type of computational model, how quickly, and with how much memory.

Theory of computation has many connections with engineering practice, and, as a true science, it also comprises philosophical aspects.

Since formal languages are of fundamental importance to computer science, we shall start our course by having a closer look at them.

First, we clarify the subject of *formal language theory*. Generally speaking, formal language theory concerns itself with sets of strings called ***languages*** and different mechanisms for generating and recognizing them. Mechanisms for generating sets of strings are usually referred to as ***grammars*** and mechanisms for recognizing sets of strings are called ***acceptors*** or ***automata***. If we compare formal languages with natural languages, then mechanisms for generating sets of strings are needed to model the speaker and mechanisms for recognizing or accepting strings model the listener. Clearly, the speaker is supposed to generate exclusively sentences belonging to the language on hand. On the other hand, the listener has first to check if the sentence she is listening to does really belong to the language on hand before she can start to further reflect about its semantics. Recent research in neuro-biology has shown that humans indeed first parse sentences they are listening to before they start thinking about them. As a matter of fact, though the parsing process in the brain is not yet understood, it could be shown that parsing is usually done quite fast.

The same objective is sought for formal languages. That is, we wish to develop a theory such that generators and acceptors do coincide with respect to the set of strings generated and accepted, respectively.

A mathematical theory for generating and accepting languages has been emerged in the later 1950's and has been extensively developed since then. Nowadays there are elaborated theories for both computer languages and natural languages. Clearly, a couple of lectures are just too few to cover even a bit of the most beautiful parts. We therefore have to restrict ourselves to the most fundamental parts of formal language theory, i.e., to the regular languages, the context-free languages, and the recursively enumerable languages. Nevertheless, this will suffice to obtain a basic understanding of what formal language theory is all about and what are the fundamental proof techniques. In order to have a common ground, we shortly recall the mathematical background needed.

1.2. Basic Definitions and Notation

By $\mathbb{N} = \{0, 1, 2, \ldots\}$ we denote the set of all natural numbers and we set $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. Let X, Y be any two sets; then we use $X \cup Y$, $X \cap Y$ and $X \setminus Y$ to denote the union, intersection and difference of X and Y, respectively. If we have countably many sets X_0, X_1, X_2, ..., then we write $\bigcup_{i \geqslant 0} X_i$ to denote the union of all sets X_i, $i \in \mathbb{N}$, i.e.,

$$\bigcup_{i \geqslant 0} X_i = X_0 \cup X_1 \cup \cdots \cup X_n \cup \cdots$$

Analogously, we use $\bigcap_{i \geqslant 0} X_i$ to denote the intersection of all X_i, $i \in \mathbb{N}$, i.e.,

$$\bigcap_{i \geqslant 0} X_i = X_0 \cap X_1 \cap \cdots \cap X_n \cap \cdots$$

Furthermore, we denote the empty set by $\emptyset$.

Next, we recall the definition of ***binary relation***. Let X, Y be any non-empty sets. We set $X \times Y = \{(x, y) \mid x \in X \text{ and } y \in Y\}$. Every subset $\rho \subseteq X \times Y$ is said to be a binary relation. Note that we sometimes use the notation $x \rho y$ instead of writing $(x, y) \in \rho$.

Definition 1. *Let $\rho \subseteq X \times Y$ and $\tau \subseteq Y \times Z$ be binary relations. The **composition** of ρ and τ is the binary relation $\zeta \subseteq X \times Z$ defined as follows:*

$$\zeta = \rho\tau = \{(x, z) \mid \text{ there exists a } y \in Y \text{ such that } (x, y) \in \rho \text{ and } (y, z) \in \tau\} .$$

Now, let X be any non-empty set; there is a special binary relation ρ^0 called ***equality***, and defined as $\rho^0 = \{(x, x) \mid x \in X\}$. Moreover, let $\rho \subseteq X \times X$ be any binary relation. Then we define for each $i \geqslant 0$ inductively $\rho^{i+1} = \rho^i \rho$.

Definition 2. *Let X be any non-empty set, and let ρ be any binary relation over X. The **reflexive–transitive closure** of ρ is the binary relation $\rho^* = \bigcup_{i \geqslant 0} \rho^i$.*

Let us illustrate the latter definition by using the following example. We define

$$\rho = \{(x, x + 1) \mid x \in \mathbb{N}\} .$$

Then, $\rho^0 = \{(x, x) \mid x \in \mathbb{N}\}$, and $\rho^1 = \rho$. Next we compute

$$\rho^2 = \rho\rho = \{(x, z) \mid x, z \in \mathbb{N} \text{ such that there is a } y \in \mathbb{N} \text{ with } (x, y) \in \rho \text{ and } (y, z) \in \rho\}$$

By the definition of ρ, $(x, y) \in \rho$ implies $y = x+1$, and $(x+1, z) \in \rho$ implies $z = x+2$. Hence,

$$\rho^2 = \{(x, x + 2) \mid x \in \mathbb{N}\} .$$

We proceed inductively. Taking into account that we have just proved the induction base, we can assume the following induction hypothesis

$$\rho^i = \{(x, x + i) \mid x \in \mathbb{N}\} .$$

Claim. $\rho^{i+1} = \{(x, x+i+1) \mid x \in \mathbb{N}\}$.

By definition, $\rho^{i+1} = \rho^i\rho$, and thus, by the definition of composition and the induction hypothesis we get:

$$\begin{aligned}\rho^i\rho &= \{(x,z) \mid x,\ z \in \mathbb{N} \text{ and there exists a } y \text{ such that } (x,y) \in \rho^i \text{ and } (y,z) \in \rho\} \\ &= \{(x, x+i+1) \mid x \in \mathbb{N}\}\ ,\end{aligned}$$

since $(x,y) \in \rho^i$ implies $y = x+i$. This proves the claim.

Finally, $\rho^* = \bigcup_{i \geqslant 0} \rho^i$, and therefore ρ^* is just the well known binary relation "$\leqslant$" over $\mathbb{N}$, i.e., $(x,y) \in \rho^*$ if and only if $x \leqslant y$.

Exercise 1. *Prove or disprove: For every binary relation ρ over a set X we have $\rho^* = (\rho^*)^*$, i.e., the reflexive–transitive closure of the reflexive–transitive closure is the reflexive–transitive closure itself.*

A formalism is required to deal with strings and sets of strings, and we therefore introduce it here. By Σ we denote a finite non-empty set called ***alphabet***. The elements of Σ are assumed to be *indivisible symbols* and referred to as ***letters*** or ***symbols***. For example, $\Sigma = \{0, 1\}$ is an alphabet containing the letters 0 and 1, and $\Sigma = \{a, b, c\}$ is an alphabet containing the letters a, b, and c. In certain applications, e.g., in compiling, we may also have alphabets containing for example **begin** and **end**. But the **begin** and **end** are also assumed to be indivisible.

Definition 3. *A **string** over an alphabet Σ is a finite length sequence of letters from Σ. A typical string is written as $s = a_1a_2\cdots a_k$, where $a_i \in \Sigma$ for $i = 1, \ldots, k$.*

Note that we also allow $k = 0$ resulting in the ***empty*** string which we denote by λ. We call k the ***length*** of s and denote it by $|s|$, so $|\lambda| = 0$. By Σ^* we denote the set of all strings over Σ, and we set $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. Now, let $s, w \in \Sigma^*$; we define a binary operation called ***concatenation*** (or word product). The concatenation of s and w is the string sw. For example, let $\Sigma = \{0, 1\}$, $s = 000111$ and $w = 0011$; then $sw = 0001110011$.

The following proposition summarizes the basic properties of concatenation.*

Proposition 1.1. *Let Σ be any alphabet.*

(1) *Concatenation is associative, i.e., for all $x, y, z \in \Sigma^*$, $x(yz) = (xy)z$*

(2) *The empty string λ is a two-sided identity for Σ^*, i.e., for all $x \in \Sigma^*$,*

$$x\lambda = \lambda x = x$$

(3) *Σ^* is free of nontrivial identities, i.e., for all $x, y, z \in \Sigma^*$,*

i) $zx = zy$ implies $x = y$ and,
ii) $xz = yz$ implies $x = y$.

(4) *For all $x, y \in \Sigma^*$, $|xy| = |x| + |y|$*

*Because of these properties, Σ^* is also referred to as *free monoid* in the literature.

Next, we extend our operations on strings to sets of strings. Let X, Y be sets of strings. Then the ***product*** of X and Y is defined as

$$XY = \{xy \mid x \in X \text{ and } y \in Y\}.$$

Let $X \subseteq \Sigma^*$; define $X^0 = \{\lambda\}$ and for all $i \geqslant 0$ set $X^{i+1} = X^i X$. The ***Kleene closure*** of X is defined as $X^* = \bigcup_{i \geqslant 0} X^i$, and the ***semigroup closure*** of X is $X^+ = \bigcup_{i \geqslant 1} X^i$.

Finally, we define the ***transpose*** of a string and of sets of strings.

Definition 4. *Let Σ be any alphabet. The* ***transpose*** *operator is defined on strings in Σ^* as follows:*

$$\lambda^T = \lambda, \text{ and}$$

$$(xa)^T = a(x^T) \text{ for all } x \in \Sigma^* \text{ and } a \in \Sigma\ .$$

We extend it to sets X of strings by setting $X^T = \{x^T \mid x \in X\}$. For example, let $s = aabbcc$, then $s^T = ccbbaa$. Furthermore, let $X = \{a^i b^j \mid i, j \geqslant 1\}$, then $X^T = \{b^j a^i \mid i, j \geqslant 1\}$. Here, a^i denotes the string $\underbrace{a \cdots a}_{i \text{ times}}$.

We continue by defining languages.

Definition 5. *Let Σ be any alphabet. Every subset $L \subseteq \Sigma^*$ is called* ***language****.*

Note that the empty set as well as $L = \{\lambda\}$ are also languages. Next, we ask how many languages there are. Let m be the cardinality of Σ. There is precisely one string of length 0, i.e., λ, there are m strings of length 1, i.e., a for all $a \in \Sigma$, there are m^2 many strings of length 2, and in general there are m^n many strings of length n. Thus, the cardinality of Σ^* is countably infinite. Therefore, by a famous theorem by G. Cantor we can conclude that there are *uncountably* many languages (as much as there are real numbers). Since the generation and recognition of languages should be done algorithmically, we immediately see that only countably many languages can be generated and recognized by an algorithm.

Finally, let us look at something interesting from natural languages and let us see how we can put this into the framework developed so far.

That is, we want to look at palindromes. A ***palindrome*** is a string that reads the same from left to right and from right to left. For having some examples from different languages, please look at the following strings.

トビコミコビト

にわとりとことりとわに

テングノハハノグンテ

AKASAKA, or removing space and punctuation symbols, the famous self-introduction of Adam to Eve: madamimadam (Madam, I'm Adam).

Now we ask how can we describe the language of all palindromes over the alphabet $\{a, b\}$ (just to keep it simple).

So far, you may have seen inductive definitions mainly in arithmetic, e.g., of the faculty function defined over $\mathbb{N}$ and denoted $n!$. It can be inductively defined as $0! = 1$ and $(n+1)! = (n+1)n!$.

One of the nice properties of free monoids is that we can adopt the concepts of "inductive definition" and "proof by induction." Please think about this. It may be helpful if you try even the more general problem in which mathematical structures inductive definitions and proof by induction is possible.

So let us try it. Of course λ, a, and b are palindromes. Since every palindrome must begin and end with the same letter, and if we remove the first and last letter of a palindrome, we still get a palindrome. This observation suggests the following basis and induction for defining L_{pal}.

Basis: λ, a, and b are palindromes.

Induction: If $w \in \{a, b\}^*$ is a palindrome, then awa and bwb are also palindromes. Furthermore, no string $w \in \{a, b\}^*$ is a palindrome, unless it follows from this basis and induction rule.

But stop, we could have also used the transpose operator T to define the language of all palindromes, i.e.,

$$\tilde{L}_{pal} = \{w \in \{a, b\}^* \mid w = w^T\} .$$

Note that we used a different notation in the latter definition, since we still do not know whether or not $L_{pal} = \tilde{L}_{pal}$. For establishing this equality, we need a proof.

Theorem 1.1. $L_{pal} = \tilde{L}_{pal}$

Proof. Equality of sets is usually proved by showing the two inclusions. So, let us first show that $L_{pal} \subseteq \tilde{L}_{pal}$.

We start with the strings defined by the basis, i.e., λ, a, and b. By the definition of the transpose operator, we have $\lambda^T = \lambda$. Thus, $\lambda \in \tilde{L}_{pal}$. Next, we deal with a. In order to apply the definition of the transpose operator, we use Property (2) of Proposition 1.1, i.e., $a = \lambda a$. Then, we have

$$a^T = (\lambda a)^T = a\lambda^T = a\lambda = a .$$

The proof for b is analogous and thus omitted.

Now, we have the induction hypothesis that for all strings w with $|w| \leqslant n$ we have $w \in L_{pal}$ implies $w \in \tilde{L}_{pal}$. In accordance with our definition of L_{pal}, the induction step is from n to $n+2$. So, let $w\{a, b\}^*$ be any string with $|w| = n+2$. Thus, $w = ava$ where $v \in \{a, b\}^*$ such that $|v| = n$. Then v is a palindrome in the sense of the definition of L_{pal}, and by the induction hypothesis, we know that $v = v^T$. Now, we have to establish the following claims providing a special property of the transpose operator.

Claim 1. *Let Σ be any alphabet, $n \in \mathbb{N}^+$, and $w = w_1 \dots w_n \in \Sigma^*$, where $w_i \in \Sigma$ for all $i \in \{1, \dots, n\}$. Then $w^T = w_n \dots w_1$.*

The proof is by induction. The induction basis is for $w_1 \in \Sigma$ and done as above. Now, we have the induction hypothesis that $(w_1 \dots w_n)^T = w_n \dots w_1$. The induction step is from n to $n+1$ and done as follows.

$$(w_1 \dots w_n w_{n+1})^T = w_{n+1}(w_1 \dots w_n)^T = w_{n+1} w_n \dots w_1 \ .$$

Note that the first equality above is by the definition of the transpose operator and the second one by the induction hypothesis. Thus, Claim 1 is proved.

Claim 2. *For all* $n \in \mathbb{N}$, *if* $p = p_1 x p_{n+2}$ *then* $p^T = p_{n+2} x^T p_1$ *for all* $p_1, p_{n+2} \in \{a, b\}$ *and* $x \in \{a, b\}^*$, *where* $|x| = n$.

Let $p = p_1 x p_{n+2}$ and $x = x_1 \dots x_n$, where $x_i \in \Sigma$. Then, $p = p_1 x_1 \dots x_n p_{n+2}$ and by Claim 1, we have $p^T = p_{n+2} x_n \dots x_1 p_1$ as well as $x^T = x_n \dots x_1$. Hence, $p_{n+2} x^T p_1 = p^T$ (see Proposition 1.1) and Claim 2 is shown.

Consequently, by using Claim 2 just established

$$w^T = (a v a)^T = a v^T a \underbrace{=}_{\text{by IH}} a v a = w \ .$$

Again, the case $w = bvb$ can be handled analogously and is thus omitted.

For completing the proof, we have to show $\tilde{L}_{pal} \subseteq L_{pal}$. For the induction basis, we know that $\lambda = \lambda^T$, i.e., $\lambda \in \tilde{L}_{pal}$ and by the "basis" part of the definition of L_{pal}, we also know that $\lambda \in L_{pal}$.

Thus, we have the induction hypothesis that for all strings w of length n: if $w = w^T$ then $w \in L_{pal}$.

The induction step is from n to $n+1$. That is, we have to show: if $|w| = n+1$ and $w = w^T$ then $w \in L_{pal}$.

Since the case $n = 1$ directly results in a and b and since $a, b \in L_{pal}$, we assume $n > 1$ in the following. So, let $w \in \{a, b\}$ be any string with $|w| = n+1$ and $w = w^T$, say $w = a_1 \dots a_{n+1}$, where $a_i \in \Sigma$. Thus, by assumption we have

$$a_1 \dots a_{n+1} = a_{n+1} \dots a_1 \ .$$

Now, applying Property (3) of Proposition 1.1 directly yields $a_1 = a_{n+1}$. We have to distinguish the cases $a_1 = a$ and $a_1 = b$. Since both cases can be handled analogously, we consider only the case $a_1 = a$ here. Thus, we can conclude $w = ava$, where $v \in \{a, b\}^*$ and $|v| = n - 1$. Next, applying the property of the transpose operator established above, we obtain $v = v^T$, i.e., $v \in L_{pal}$. Finally, the "induction" part of the definition of L_{pal} directly implies $w \in L_{pal}$. ∎

Since we shall see the language of palindromes throughout this course occasionally, please ensure that you have understood what we have done above.

Lecture 2: Introducing Formal Grammars

We start this lecture by formalizing what is meant by generating a language. If we look at natural languages, then we have the following situation. Σ consists of all words in the language. Although large, Σ is finite. What is usually done in speaking or writing natural languages is forming sentences. A typical sentence starts with a noun phrase followed by a verb phrase. Thus, we may describe this generation by

$$< \text{sentence} > \rightarrow < \text{noun phrase} >< \text{verb phrase} >$$

Clearly, more complicated sentences are generated by more complicated rules. If you look in a usual grammar book, e.g., for the German language, then you see that there are, however, only finitely many rules for generating sentences.

This suggest the following general definition of a grammar.

Definition 6. $\mathcal{G} = [T, N, \sigma, P]$ *is said to be a* ***grammar*** *if*

(1) T *and* N *are alphabets with* $T \cap N = \emptyset$.

(2) $\sigma \in N$

(3) $P \subseteq ((T \cup N)^+ \setminus T^*) \times (T \cup N)^*$ *is finite.*

We call T the ***terminal alphabet***, N the ***nonterminal alphabet***, σ the ***start symbol*** and P the set of ***productions*** (or ***rules***). Usually, productions are written in the form $\alpha \rightarrow \beta$, where $\alpha \in (T \cup N)^+ \setminus T^*$ and $\beta \in (T \cup N)^*$.

Next, we have to explain how to generate a language using a grammar. This is done by the following definition.

Definition 7. *Let* $\mathcal{G} = [T, N, \sigma, P]$ *be a grammar. Let* α', $\beta' \in (T \cup N)^*$. α' *is said to* ***directly generate*** β', *written* $\alpha' \Rightarrow \beta'$, *if there exist* α_1, α_2, α, $\beta \in (T \cup N)^*$ *such that* $\alpha' = \alpha_1\alpha\alpha_2$, $\beta' = \alpha_1\beta\alpha_2$ *and* $\alpha \rightarrow \beta$ *is in* P. *We write* $\overset{*}{\Rightarrow}$ *for the reflexive transitive closure of* $\Rightarrow$.

Finally, we can define the language generated by a grammar.

Definition 8. *Let* $\mathcal{G} = [T, N, \sigma, P]$ *be a grammar. The* ***language*** $L(\mathcal{G})$ *generated by* $\mathcal{G}$ *is defined as* $L(\mathcal{G}) = \{s \mid s \in T^* \text{ and } \sigma \overset{*}{\Rightarrow} s\}$.

Exercise 2. *Let* $T = \{a, b, c\}$ *and* $N = \{\sigma, h_1, h_2\}$ *and let* $\mathcal{G} = [T, N, \sigma, P]$, *where* P *is the set of the following productions:*

1. $\sigma \rightarrow abc$

2. $\sigma \rightarrow ah_1bc$

3. $h_1b \rightarrow bh_1$

4. $h_1c \rightarrow h_2bcc$

5. $bh_2 \to h_2b$
6. $ah_2 \to aah_1$
7. $ah_2 \to aa$

Determine $L(\mathcal{G})$.

Next, we are going to study special subclasses of grammars and the languages they generate. We start with the easiest subclass, the so-called regular languages.

2.1. Regular Languages

Definition 9. *A grammar* $\mathcal{G} = [T, N, \sigma, P]$ *is said to be* ***regular*** *provided for all* $\alpha \to \beta$ *in* P *we have* $\alpha \in N$ *and* $\beta \in T^* \cup T^*N$.

Definition 10. *A language* L *is said to be* ***regular*** *if there exists a regular grammar* $\mathcal{G}$ *such that* $L = L(\mathcal{G})$. *By* $\mathcal{REG}$ *we denote the set of all regular languages.*

Example 1. Let $\mathcal{G} = [\{a, b\}, \{\sigma\}, \sigma, P]$ with $P = \{\sigma \to ab,\ \sigma \to a\sigma\}$. $\mathcal{G}$ is regular and $L(\mathcal{G}) = \{a^n b \mid n \geqslant 1\}$.

Example 2. Let $\mathcal{G} = [\{a, b\}, \{\sigma\}, \sigma, P]$ with $P = \{\sigma \to \lambda,\ \sigma \to a\sigma,\ \sigma \to b\sigma\}$. Again, $\mathcal{G}$ is regular and $L(\mathcal{G}) = \Sigma^*$. Consequently, Σ^* is a regular language.

Example 3. Let Σ be any alphabet, and let $X \subseteq \Sigma^*$ be any finite set. Then, for $\mathcal{G} = [\Sigma, \{\sigma\}, \sigma, P]$ with $P = \{\sigma \to s \mid s \in X\}$, we have $L(\mathcal{G}) = X$. Consequently, every ***finite*** language is regular.

Now, we have already seen several examples for regular languages. As curious as we are, we are going to ask which languages are regular. For answering this question, we first deal with ***closure*** properties.

Theorem 2.1. *The regular languages are closed under union, product and Kleene closure.*

Proof. Let L_1 and L_2 be any regular languages. Since L_1 and L_2 are regular, there are regular grammars $\mathcal{G}_1 = [T_1, N_1, \sigma_1, P_1]$ and $\mathcal{G}_2 = [T_2, N_2, \sigma_2, P_2]$ such that $L_i = L(\mathcal{G}_i)$ for $i = 1, 2$. Without loss of generality, we may assume that $N_1 \cap N_2 = \emptyset$ for otherwise we simply rename the nonterminals appropriately.

We start with the union. We have to show that $L = L_1 \cup L_2$ is regular. Now, let

$$\mathcal{G} = [T_1 \cup T_2, N_1 \cup N_2 \cup \{\sigma\}, \sigma, P_1 \cup P_2 \cup \{\sigma \to \sigma_1,\ \sigma \to \sigma_2\}].$$

By construction, $\mathcal{G}$ is regular.

Claim 1. $L = L(\mathcal{G})$.

We have to start every generation of strings with σ. Thus, there are two possibilities, i.e., $\sigma \to \sigma_1$ and $\sigma \to \sigma_2$. In the first case, we can continue with all

generations that start with σ_1 yielding all strings in L_1. In the second case, we can continue with σ_2, thus getting all strings in L_2. Consequently, $L_1 \cup L_2 \subseteq L$.

On the other hand, $L \subseteq L_1 \cup L_2$ by construction. Hence, $L = L_1 \cup L_2$.

We continue with the closure under product. We have to show that L_1L_2 is regular. A first idea might be to use a construction analogous to the one above, i.e., to take as a new starting production $\sigma \rightarrow \sigma_1\sigma_2$. Unfortunately, this production is not regular. We have to be a bit more careful. But the underlying idea is fine, we just have to replace it by a sequential construction. The idea for doing that is easily described. Let $s_1 \in L_1$ and $s_2 \in L_2$. We want to generate s_1s_2. Then, starting with σ_1 there is a generation $\sigma_1 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow s_1$. But instead of finishing the generation at that point, we want to have the possibility to continue to generate s_2. Thus, all we need is a production having a right hand side resulting in $s_1\sigma_2$. This idea can be formalized as follows: Let

$$\mathcal{G} = [T_1 \cup T_2, N_1 \cup N_2, \sigma_1, P],$$

where

$$\begin{aligned} P &= P_1 \setminus \{h \rightarrow s \mid s \in T_1^* \text{ and } h \in N_1\} \\ &\cup \{h \rightarrow s\sigma_2 \mid h \rightarrow s \in P_1 \text{ and } s \in T_1^*\} \cup \{P_2\} . \end{aligned}$$

By construction, $\mathcal{G}$ is regular.

Claim 2. $L(\mathcal{G}) = L_1L_2$.

By construction we have $L_1L_2 \subseteq L(\mathcal{G})$. For showing $L(\mathcal{G}) \subseteq L_1L_2$, let $s \in L_1L_2$. Consequently, there are strings $s_1 \in L_1$ and $s_2 \in L_2$ such that $s = s_1s_2$. Since $s_1 \in L_1$, there is a generation $\sigma_1 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_n \Rightarrow s_1$ in $\mathcal{G}_1$. Note that w_n must contain precisely one nonterminal, say h, and it must be of the form $w_n = wh$ by Definition 9. Now, since $w_n \Rightarrow s_1$ and $s_1 \in T_1^*$, we must have applied a production $h \rightarrow s$ with $s \in T_1^*$ such that $wh \Rightarrow ws = s_1$. But in $\mathcal{G}$ all these productions have been replaced by $h \rightarrow s\sigma_2$. Therefore, the last generation $w_n \Rightarrow s_1$ is now replaced by $wh \Rightarrow ws\sigma_2$. All what is left, is now applying the productions from P_2 to generate s_2 which is possible, since $s_2 \in L_2$. This proves the claim.

Finally, we have to deal with the Kleene closure. Let L be a regular language, and let $\mathcal{G} = [T, N, \sigma, P]$ be a regular grammar such that $L = L(\mathcal{G})$. We have to show that L^* is regular. By definition $L^* = \bigcup_{i \geqslant 0} L^i$. Since $L^0 = \{\lambda\}$, we have to make sure that λ can be generated. This is obvious if $\lambda \in L$. Otherwise, we simply add the production $\sigma \rightarrow \lambda$. The rest is done analogously as in the product case, i.e., we set

$$\mathcal{G}^* = [T, N \cup \{\sigma^*\}, \sigma^*, P^*], \text{ where}$$

$$P^* = P \cup \{h \rightarrow s\sigma \mid h \rightarrow s \in P \text{ and } s \in T^*\} \cup \{\sigma^* \rightarrow \sigma,\ \sigma^* \rightarrow \lambda\} .$$

We leave it as an exercise to prove $L(\mathcal{G}^*) = L^*$. ∎

We finish this lecture by defining the equivalence of grammars.

Definition 11. *Let* $\mathcal{G}$ *and* $\hat{\mathcal{G}}$ *be any grammars.* $\mathcal{G}$ *and* $\hat{\mathcal{G}}$ *are said to be* ***equivalent*** *if* $\mathsf{L}(\mathcal{G}) = \mathsf{L}(\hat{\mathcal{G}})$. indexequivalence par In order to have an example for equivalent grammars, we consider

$\mathcal{G} = [\{\mathfrak{a}\}, \{\sigma\}, \sigma, \{\sigma \rightarrow \mathfrak{a}\sigma\mathfrak{a},\ \sigma \rightarrow \mathfrak{a}\mathfrak{a},\ \sigma \rightarrow \mathfrak{a}\}]$.

and the following grammar

$\hat{\mathcal{G}} = [\{\mathfrak{a}\}, \{\sigma\}, \sigma, \{\sigma \rightarrow \mathfrak{a},\ \sigma \rightarrow \mathfrak{a}\sigma\}]$.

Now, it is easy to see that $\mathsf{L}(\mathcal{G}) = \{\mathfrak{a}\}^+ = \mathsf{L}(\hat{\mathcal{G}})$, and hence $\mathcal{G}$ and $\hat{\mathcal{G}}$ are equivalent. Note however that $\hat{\mathcal{G}}$ is regular while $\mathcal{G}$ is not.

For further reading we recommend the following.

References

[1] M.A. HARRISON, *Introduction to Formal Language Theory*, Addison–Wesley Publishing Company, Reading Massachusetts, 1978.

[2] J.E. HOPCROFT AND J.D. ULLMAN, *Formal Languages and their Relation to Automata*, Addison–Wesley Publishing Company, Reading Massachusetts, 1969.

[2] H.R. LEWIS AND C.H. PAPADIMITRIOU, *Elements of the Theory of Computation (2nd Edition)*, Prentice Hall, Upper Saddle River, New Jersey, 1998.

Lecture 3 – Finite State Automata

In the previous lecture we learned how to formalize the generation of languages. This part looked at formal languages from the perspective of a speaker. Now, we turn our attention to accepting languages, i.e., we are going to formalize the perspective of a listener. In this lecture we deal with regular languages, and the machine model accepting them. The overall goal can be described as follows. Let Σ be again an alphabet, and let $L \subseteq \Sigma^*$ be any regular language. Now, for every string $s \in \Sigma^*$ we want to have a possibility to decide whether or not $s \in L$. Looking at the definition of a regular grammar, the following methods may be easily discovered. We start generating strings until one of the following two conditions happens. First, the string s is generated. Clearly, then $s \in L$. Second, the length of our string s is exceeded. Now, taking into account that all further generable strings must be longer, we may conclude that $s \notin L$. There is only one problem with this method, i.e., its efficiency. It may take time that is exponential in the length of the input string s to terminate. Besides that, this approach hardly reflects what humans are doing when accepting sentences of natural languages. We therefore favor a different approach which we define next[†].

Definition 12. *A 5-tuple $\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ is said to be a* ***nondeterministic finite automaton*** *if*

(1) Σ *is an alphabet (the so-called input alphabet),*

(2) Q *is a finite nonempty set (the set of* ***states****),*

(3) $\delta\colon Q \times \Sigma \mapsto \wp(Q)$*, the* ***transition relation****,*

(4) $q_0 \in Q$*, the* ***initial state****, and*

(5) $F \subseteq Q$*, the set of* ***final states****.*

In the definition above, $\wp(Q)$ denotes the power set of Q, i.e., the set of all subsets of Q. There is also a deterministic counterpart of a nondeterministic finite automaton which we define next.

Definition 13. *A 5-tuple $\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ is said to be a* ***deterministic finite automaton*** *if*

(1) Σ *is an alphabet (the so-called input alphabet),*

(2) Q *is a finite nonempty set (the set of* ***states****),*

(3) $\delta\colon Q \times \Sigma \mapsto Q$*, the* ***transition function****, which must be defined for every input.*

[†]Please note that M.O. Rabin and D.S. Scott received the Turing Award in 1976 for their paper *Finite Automata and Their Decision Problems*, IBM Journal of Research and Development **3**:114-125, (1959), which introduced the idea of nondeterministic machines – a concept which has proved to be enormously valuable.

(4) $q_0 \in Q$, *the* ***initial state****, and*

(5) $F \subseteq Q$, *the set of* ***final states****.*

When we do not want to specify whether an automaton is deterministic or nondeterministic, we simply refer to it as to a finite automaton.

So far, we have explained what a finite automaton is but not what it does. In order to explain how to compute with an automaton, we need some more definitions. For the deterministic case, we can easily define the language accepted by a finite automaton. First, we *extend* the definition of δ to strings. That is, formally we inductively define a function

$$\delta^*\colon Q \times \Sigma^* \mapsto Q \ ,$$

by setting

$$\begin{aligned} \delta^*(q,\lambda) &= q \quad \text{for all } q \in Q \ , \\ \delta^*(q,sa) &= \delta(\delta^*(q,s),a) \quad \text{for all strings } s \in \Sigma^*, \text{ all } a \in \Sigma, \text{ and all } q \in Q \ . \end{aligned}$$

The proof of the following lemma is left as an exercise.

Lemma 3.1. *Let $\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ be a deterministic finite automaton. Then for all strings $v, w \in \Sigma^*$ and all $q \in Q$ we have $\delta^*(q, vw) = \delta^*(\delta^*(q, v), w)$.*

Definition 14. *Let $\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ be a deterministic finite automaton. The language $L(\mathcal{A})$* ***accepted by*** *A is*

$$L(\mathcal{A}) = \{s \mid s \in \Sigma^*, \ \delta^*(q_0, s) \in F\} \ .$$

If we have $s \in L(\mathcal{A})$ for a string $s \in \Sigma^*$ then we say that there is an ***accepting computation*** for s. We adopt this notion also to nondeterministic automata. Note that $\lambda \in L(\mathcal{A})$ if $q_0 \in F$.

In order to keep notation simple, in the following we shall identify δ^* with δ. It should be clear from context what is meant.

Well, this seems very abstract, and so some explanation is in order. Conceptually, a finite automaton possesses an input tape that is divided into cells. Each cell can store a symbol from Σ or it may be empty. Additionally, a finite automaton has a ***head*** to read what is stored in the cells. Initially, a string $s = s_1 s_2 \cdots s_k$ is written on the tape and the head is positioned on the leftmost symbol of the input, i.e., on s_1 (cf. Figure 1).

Moreover, the automaton is put into its initial state q_0. Now, the automaton reads s_1. Then, it changes its state to one of the possible states in $\delta(q_0, s_1)$, say q, and the head moves right to the next cell. Note that in the deterministic case, the state $\delta(q_0, s_1)$ is uniquely defined. Next, s_2 is read, and the automaton changes its state to one of the possible states in $\delta(q, s_2)$. This process is iterated until the

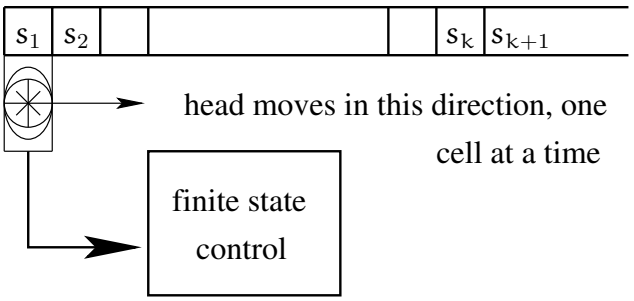

Figure 1. ***A finite automaton***

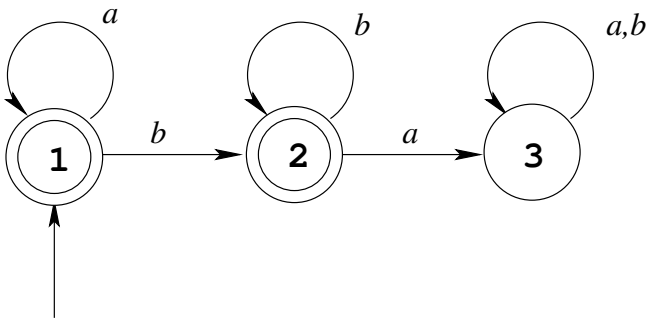

Figure 2. ***A finite automaton accepting*** $\mathsf{L} = \{\mathfrak{a}^{\mathfrak{i}}\mathfrak{b}^{\mathfrak{j}} \mid \mathfrak{i} \geqslant 0,\ \mathfrak{j} \geqslant 0\}$.

automaton reaches the first cell which is empty. Finally, after having read the whole string, the automaton is in some state, say r. If $\mathsf{r} \in \mathsf{F}$, then the computation has been an ***accepting*** one, otherwise, the string s is ***rejected***.

Now, we see what the problem is in defining the language accepted by a nondeterministic finite automaton. On input a string s, there are many possible computations. Some of these computations may finish in an accepting state and some may not. We therefore define the language accepted by a nondeterministic finite automaton as follows.

Definition 15. *Let* $\mathcal{A} = [\Sigma, \mathsf{Q}, \delta, \mathsf{q}_0, \mathsf{F}]$ *be a nondeterministic finite automaton. The language* $\mathsf{L}(\mathcal{A})$ ***accepted by*** $\mathcal{A}$ *is the set of all strings* $\mathsf{s} \in \Sigma^*$ *such that there exists an accepting computation for* s.

Finally, finite automata may be conveniently represented by their ***state diagram***. The state diagram is a directed graph whose nodes are labeled by the states of the automaton. The edges are labeled by symbols from Σ. Let p and q be nodes. Then, there is a directed edge from p to q if and only if there exists an $\mathfrak{a} \in \Sigma$ such that $\mathsf{q} = \delta(\mathsf{p}, \mathfrak{a})$ (deterministic case) or $\mathsf{q} \in \delta(\mathsf{p}, \mathfrak{a})$ (nondeterministic case). Figure 2 shows the state diagram of a finite automaton accepting the language $\mathsf{L} = \{\mathfrak{a}^{\mathfrak{i}}\mathfrak{b}^{\mathfrak{j}} \mid \mathfrak{i} \geqslant 0,\ \mathfrak{j} \geqslant 0\}$. Note that, by convention, $\mathfrak{a}^0\mathfrak{b}^0 = \lambda$.

The automaton displayed in Figure 2 has 3 states, i.e., $\mathsf{Q} = \{1, 2, 3\}$. The input alphabet is $\Sigma = \{\mathfrak{a}, \mathfrak{b}\}$, and the set F of final states is $\{1, 2\}$. As usual, we have marked

the final states by drawing an extra circle in then. The initial state is marked by an unlabeled arrow, that is, 1 is the initial state.

Now that we know what finite automata are, we can answer the question you probably have already in mind, i.e.,

What have finite automata to do with regular languages?

We answer this question by the following theorem.

Theorem 3.2. *Let* $L \subseteq \Sigma^*$ *be any language. Then, the following three assertions are equivalent.*

(1) *There exists a deterministic finite automaton* $\mathcal{A}$ *such that* $L = L(\mathcal{A})$.

(2) *There exists a nondeterministic finite automaton* $\mathcal{A}$ *such that* $L = L(\mathcal{A})$.

(3) L *is regular.*

Proof. We show the equivalence by proving (1) implies (2), (2) implies (3), and (3) implies (1).

Claim 1. (1) implies (2).

This is obvious by definition, since a deterministic finite automaton is a special case of a nondeterministic one.

Claim 2. (2) implies (3).

$\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ be a nondeterministic finite automaton such that $L = L(\mathcal{A})$. We have to construct a grammar $\mathcal{G}$ generating L. Let $\mathcal{G} = [\Sigma,\ Q \cup \{\sigma\},\ \sigma,\ P]$, where P is the following set of productions:

$$P = \{\sigma \rightarrow q_0\} \cup \{p \rightarrow aq \mid a \in \Sigma,\ p, q \in Q,\ q \in \delta(p, a)\} \cup \{p \rightarrow \lambda \mid p \in F\}.$$

Obviously, $\mathcal{G}$ is regular. We have to show $L(\mathcal{G}) = L(\mathcal{A})$. First we prove $L(\mathcal{A}) \subseteq L(\mathcal{G})$.

Let $s = a_1 \cdots a_k \in L$. Then, there exists an accepting computation of $\mathcal{A}$ for s. Let $q_0, p_1, \ldots, p_k$ be the sequence of states through which $\mathcal{A}$ goes while performing this accepting computation. Therefore, $p_1 \in \delta(q_0, a_1)$, $p_2 \in \delta(p_1, a_2)$, ..., $p_k \in \delta(p_{k-1}, a_k)$, and $p_k \in F$. Thus, $\sigma \Rightarrow q_0 \Rightarrow a_1 p_1 \Rightarrow \cdots \Rightarrow a_1 \cdots a_{k-1} p_{k-1} \Rightarrow a_1 \cdots a_k p_k \Rightarrow a_1 \cdots a_k$. Hence, $s \in L(\mathcal{G})$. The direction $L(\mathcal{G}) \subseteq L(\mathcal{A})$ can be proved analogously, and is therefore left as an exercise.

Claim 3. (3) implies (1).

In principle, we want to use an idea similar to the one used to prove Claim 2. But productions may have strings on their right hand side, while the transition function has to be defined over states and letters. Therefore, we first have to show a *normal form* lemma for regular grammars.

Lemma. *For every regular grammar* $\mathcal{G} = [T, N, \sigma, P]$ *there exists a grammar* $\mathcal{G}'$ *such that* $L(\mathcal{G}) = L(\mathcal{G}')$ *and all productions of* $\mathcal{G}'$ *have the form* $h \rightarrow ah'$ *or* $h \rightarrow \lambda$, *where* $h, h' \in N$ *and* $a \in T$.

First, each production of the form $h \to a_1 \cdots a_k h'$, $k > 0$, is equivalently replaced by the following productions:

$h \to a_1 h_{a_2 \cdots a_k h'}$, $h_{a_2 \cdots a_k h'} \to a_2 h_{a_3 \cdots a_k h'}$, $\ldots$, $h_{a_k h'} \to a_k h'$.

Next, each production of the form $h \to a_1 \cdots a_k$, $k > 0$, is equivalently replaced by the following productions:

$h \to a_1 h_{a_2 \cdots a_k}$, $h_{a_2 \cdots a_k} \to a_2 h_{a_3 \cdots a_k}$, $\ldots$, $h_{a_k} \to a_k h_\lambda$ and $h_\lambda \to \lambda$.

Finally, we have to deal with productions of the form $h \to h'$ where $h, h' \in N$. Let $\hat{G} = [T, \hat{N}, \sigma, \hat{P}]$ be the grammar constructed so far. Furthermore, let $U(h) =_{df} \{h' \mid h' \in \hat{N} \text{ and } h \stackrel{*}{\Rightarrow} h'\}$. Clearly, $U(h)$ is computable. Now, we delete all productions of the form $h \to h'$ and add the following productions to $\hat{P}$. If $h' \to \lambda \in \hat{P}$ and $h' \in U(h)$, then we add $h \to \lambda$. Moreover, we add all $h \to xh''$ for all productions in $\hat{P}$ such that there is a $h' \in U(h)$ with $h' \to xh'' \in \hat{P}$.

Let G' be the resulting grammar. Clearly, now all productions have the desired form. One easily verifies that $L(G) = L(G')$. This proves the lemma.

Now, assume that we are given a grammar $G = [T, N, \sigma, P]$ that is already in the normal form described in the lemma above. We define a deterministic finite automaton $\mathcal{A} = [T, Q, \delta, q_0, F]$ as follows. Let $Q = \wp(N)$ and $q_0 = \{\sigma\}$. The transition function δ is defined as

$$\delta(p, a) = \{h' \mid \exists h[h \in p \text{ and } h \to ah' \in P]\}.$$

Finally, we set

$$F = \{p \mid \exists h[h \in p \text{ and } h \to \lambda \in P]\}.$$

We leave it as an exercise to prove $L(\mathcal{A}) = L(G)$. ∎

Next, we illustrate the transformation of a grammar into a deterministic finite automaton by an example. Let $G = [\{a, b\}, \{\sigma, h\}, \sigma, P]$ be the grammar given, where $P = \{\sigma \to ab\sigma,\ \sigma \to h,\ h \to aah,\ h \to \lambda\}$. First, we have to transform G into an equivalent grammar in normal form. Thus, we obtain

$\hat{P} = \{\sigma \to ah_{b\sigma},\ h_{b\sigma} \to b\sigma,\ \sigma \to h,\ h \to ah_{ah},\ h_{ah} \to ah,\ h \to \lambda\}$.

Now, $U(\sigma) = \{h\}$, and $U(h) = \emptyset$. Thus, we delete the production $\sigma \to h$ and replace it by $\sigma \to ah_{ah}$ and $\sigma \to \lambda$. Summarizing this construction, we now have the following set P' of productions

$$\begin{aligned} P' \;=\; & \{\sigma \to ah_{b\sigma},\ h_{b\sigma} \to b\sigma,\ \sigma \to ah_{ah},\ \sigma \to \lambda,\ h \to ah_{ah}, \\ & h_{ah} \to ah,\ h \to \lambda\} \end{aligned}$$

as well as the following set N' of nonterminals

$$N' = \{\sigma,\ h_{ah},\ h_{b\sigma},\ h\}\ .$$

Thus, our automaton has 16 states, i.e.,

$\{\emptyset, \{\sigma\}, \{h_{ah}\}, \{h_{b\sigma}\}, \{h\}, \{\sigma, h_{ah}\}, \{\sigma, h_{b\sigma}\}, \{\sigma, h\}, \{h_{ah}, h_{b\sigma}\}, \{h_{ah}, h\}, \{h, h_{b\sigma}\}, \{\sigma, h_{ah}, h_{b\sigma}\}, \{\sigma, h_{ah}, h\}, \{\sigma, h_{b\sigma}, h\}, \{h_{ah}, h_{b\sigma}, h\}, \{\sigma, h_{ah}, h_{b\sigma}, h\}\}$

The set of final states is

$$\begin{aligned} F &= \{\{\sigma\}, \{h\}\} \\ &\cup \{\{\sigma, h_{ah}\}, \{\sigma, h_{b\sigma}\}, \{\sigma, h\}, \{h_{ah}, h\}, \{h, h_{b\sigma}\}, \{\sigma, h_{ah}, h_{b\sigma}\}, \{\sigma, h_{ah}, h\}, \\ &\quad \{\sigma, h_{b\sigma}, h\}, \{h_{ah}, h_{b\sigma}, h\}, \{\sigma, h_{ah}, h_{b\sigma}, h\}\} . \end{aligned}$$

Finally, we have to compute δ. We illustrate this part here only for two states, the rest is left as an exercise. For computing $\delta(\{\sigma\}, a)$, we have to consider the set of all productions having σ on the left hand side and a on the right hand side. There are two such productions, i.e., $\sigma \to a h_{b\sigma}$ and $\sigma \to a h_{ah}$; thus $\delta(\{\sigma\}, a) = \{h_{b\sigma}, h_{ah}\}$. Since there is no production having σ on the left hand side and b on the right hand side, we obtain $\delta(\{\sigma\}, b) = \emptyset$. Analogously, we get $\delta(\{h_{b\sigma}, h_{ah}\}, a) = \{h\}$, and $\delta(\{h_{b\sigma}, h_{ah}\}, b) = \{\sigma\}$, $\delta(\{h\}, a) = \{h_{ah}\}$, $\delta(\{h\}, b) = \emptyset$, $\delta(\{h_{ah}\}, a) = \{h\}$, $\delta(\{h_{ah}\}, b) = \emptyset$ as well as $\delta(\emptyset, a) = \delta(\emptyset, b) = \emptyset$.

Looking at the transitions computed so far, we see that none of the other states appeared, and thus they can be ignored.

Exercise 3. *Complete the calculation of the automaton above and draw its state diagram.*

Finally, we show how to use finite automata to prove that particular languages are ***not regular***. Consider the following grammar

$$G = [\{a, b\}, \{\sigma\}, \sigma, \{\sigma \to a\sigma b,\ \sigma \to \lambda\}]$$

Then, $L(G) = \{a^i b^i \mid i \geqslant 0\}$. Clearly, G is not regular, but this does not prove that there is no regular grammar at all that can generate this language.

Theorem 3.3. *The language* $L = \{a^i b^i \mid i \geqslant 0\}$ *is not regular.*

Proof. Suppose the converse. Then, by Theorem 3.2 there must be a deterministic finite automaton $\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ such that $L(\mathcal{A}) = L$. Clearly, $\{a, b\} \subseteq \Sigma$, and thus $\delta(q_0, a^i)$ must be defined for all i. However, there are only finitely many states, but infinitely many i, and hence there must exist i, j such that $i \neq j$ but $\delta(q_0, a^i) = \delta(q_0, a^j)$. Since the automaton is deterministic, $\delta(q_0, a^i) = \delta(q_0, a^j)$ implies $\delta(q_0, a^i b^i) = \delta(q_0, a^j b^i)$. Let $q = \delta(q_0, a^i b^i)$; then, if $q \in F$ we directly obtain $a^j b^i \in L$, a contradiction, since $i \neq j$. But if $q \notin F$, then $a^i b^i$ is rejected. This is again a contradiction, since $a^i b^i \in L$. Thus, L is not regular. ∎

There are numerous books on formal languages and automata theory. This lecturer would like to finish this lecture by recommending further reading in any of the books mentioned in the recommended literature.

Last but not least, we need some more advanced problems to see if we have understood so far everything correctly, and here they come.

Example 4. *Let* $\Sigma = \{\mathtt{a}, \mathtt{b}\}$*, and let the formal languages* L_1 *and* L_2 *be defined as* $\mathsf{L}_1 = \{w \in \Sigma^+ \mid |w| \text{ is divisible by } 4\}$ *and* $\mathsf{L}_2 = \{w \in \Sigma^* \mid w = w^{\mathsf{T}}\}$*. Prove or disprove* L_1 *and* L_2*, respectively, to be regular.*

Lecture 4: Characterizations of $\mathcal{REG}$

We start this lecture by proving an algebraic characterization for $\mathcal{REG}$. For this purpose, first we recall the definition of an equivalence relation.

Definition 16. *Let $M \neq \emptyset$ be any set. $\rho \subseteq M \times M$ is said to be an **equivalence relation** (over M) if*

(1) *ρ is reflexive, i.e., $(x, x) \in \rho$ for all $x \in M$.*

(2) *ρ is symmetric, i.e., if $(x, y) \in \rho$ then $(y, x) \in \rho$ for all $x, y \in M$.*

(3) *ρ is transitive, i.e., if $(x, y) \in \rho$ and $(y, z) \in \rho$ then $(x, z) \in \rho$ for all $x, y, z \in M$.*

Furthermore, for any $x \in M$, we write $[x]$ to denote the equivalence class generated by x, i.e.,

$$[x] = \{y \mid y \in M \text{ and } (x, y) \in \rho\} .$$

Recall that the set of all equivalence classes forms a partition of the underlying set M. We use $M/_\rho$ to denote the set of all equivalence classes of M with respect to ρ. Finally, as usual we also write sometimes $x \rho y$ instead of $(x, y) \in \rho$.

Additionally, we need the following definitions.

Definition 17. *Let $M \neq \emptyset$ be any set and let ρ be any equivalence relation over M. The relation ρ is said to be of **finite rank** if $M/_\rho$ is finite.*

Now, let Σ be any alphabet, let $L \subseteq \Sigma^*$ be any language and let $v, w \in \Sigma^*$. We define

$$v \sim_L w \text{ iff } \forall u \in \Sigma^*[vu \in L \Leftrightarrow wu \in L] .$$

Exercise 4. *Show that $\sim_L$ is an equivalence relation.*

Note that $\sim_L$ is also called ***Nerode relation***. Furthermore, let $\sim$ be any equivalence relation over Σ^*. We call $\sim$ ***right invariant*** if

$$\forall u, v, w \in \Sigma^*[u \sim v \text{ implies } uw \sim vw] .$$

Exercise 5. *Show $\sim_L$ to be right invariant.*

Now, we are ready to prove our first characterization.

Theorem 4.1 (Nerode Theorem) *Let Σ be any alphabet and let $L \subseteq \Sigma^*$ be any language. Then we have*

$L \in \mathcal{REG}$ if and only if $\sim_L$ is of finite rank.

Proof. Necessity: Let $L \in \mathcal{REG}$. Then there is a deterministic finite automaton $\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ such that $L(\mathcal{A}) = L$. We define the following relation $\sim$ over Σ^*: for all $v, w \in \Sigma^*$

$$v \sim w \text{ iff } \delta(q_0, v) = \delta(q_0, w) .$$

Claim 1. $\sim$ *is an equivalence relation.*

Since $=$ is an equivalence relation, we can directly conclude that $\sim$ is an equivalence relation.

Claim 2. $\sim$ *is of finite rank.*

This is also clear, since $\mathrm{card}(\Sigma^*/_\sim) \leqslant \mathrm{card}(Q)$ and $\mathrm{card}(Q)$ is finite by definition.

Claim 3. $v \sim w$ *implies* $v \sim_L w$ *for all* $v, w \in \Sigma^*$.

This can be seen as follows. Let $u, v, w \in \Sigma^*$ such that $v \sim w$. We have to show that $vu \in L \Leftrightarrow wu \in L$. Since $L = L(\mathcal{A})$, it suffices to prove that $vu \in L(\mathcal{A}) \Leftrightarrow wu \in L(\mathcal{A})$. By Definition 14 we know that $vu \in L(\mathcal{A})$ iff $\delta(q_0, vu) \in F$. Because of $v \sim w$ we also have $\delta(q_0, v) = \delta(q_0, w)$. Consequently, by Lemma 3.1 we have

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

Thus, $\delta(q_0, vu) \in F$ iff $\delta(q_0, wu) \in F$ and therefore $vu \in L \Leftrightarrow wu \in L$. This proves Claim 3.

Claim 1 through 3 directly imply that $\sim_L$ is of finite rank.

Sufficiency: Let $L \subseteq \Sigma^*$ be such that $\sim_L$ is of finite rank. We have to show that $L \in \mathcal{REG}$. By Theorem 3.2 it suffices to construct a deterministic finite automaton $\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ such that $L = L(\mathcal{A})$. This is done as follows. We set

- $Q = \Sigma^*/_{\sim_L}$,
- $q_0 = [\lambda]$,
- $\delta([w], x) = [wx]$ for all $[w] \in Q$ and all $x \in \Sigma$,
- $F = \{[w] \mid w \in L\}$.

Exercise 6. *Show that the definition of* δ *does not depend on the representative of the equivalence class* $[w]$.

Claim 4. $\delta([\lambda], w) = [w]$ *for all* $w \in \Sigma^*$.

The claim is proved inductively. For the induction base we have

$$\delta([\lambda], x) = [\lambda x] = [x] \ ,$$

since $\lambda x = x$ for all $x \in \Sigma$.

Now, suppose as induction hypothesis $\delta([\lambda], w) = [w]$ and let $x \in \Sigma$. We have to show $\delta([\lambda], wx) = [wx]$. So, we calculate

$$\delta([\lambda], wx) = \delta(\delta([\lambda], w), x) = \delta([w], x) = [wx] \ ,$$

where the first equality is by the extension of the definition of δ to strings, the second one by the induction hypothesis and the last one by the definition of δ. This proves Claim 4.

By construction and Claim 4, we directly obtain

$$\begin{aligned} w \in \mathsf{L}(\mathcal{A}) &\Leftrightarrow \delta([\lambda], w) \in \mathsf{F} \\ &\Leftrightarrow [w] \in \mathsf{F} \\ &\Leftrightarrow w \in \mathsf{L} \ . \end{aligned}$$

This shows the sufficiency and thus the theorem is shown. ∎

It should be helpful for you to prove the following exercise.

Exercise 7. *Let Σ be any alphabet and let $\mathsf{L} \subseteq \Sigma^*$ be any language. Then we have $\mathsf{L} \in \mathcal{RES}$ if and only if there is a right invariant equivalence relation $\approx$ such that $\approx$ is of finite rank and L is the union of some equivalence classes with respect to $\approx$.*

Next, we want to construct a language such that each string of it can be regarded as a generator of a regular language.

4.1. Regular Expressions

Let Σ be any fixed alphabet. We define

$$\underline{\Sigma} = \{\underline{x} |\ x \in \Sigma\} \cup \{\oslash, \Lambda, \vee, \cdot, \langle, \rangle, (,)\} \ ,$$

that is, for every $x \in \Sigma$ we introduce a *new* symbol called $\underline{x}$ and, additionally, we introduce the symbols $\oslash$, Λ, $\vee$, $\cdot$, $\langle$, $\rangle$, $($, $)$. Note that the comma is a *meta* symbol.

Next we set $\mathcal{G}_{reg} =_{\mathtt{df}} [\underline{\Sigma}, \{\sigma\}, \sigma, P]$, where P is defined as follows

$$P = \{\sigma \to \underline{x} |\ x \in \Sigma\} \cup \{\sigma \to \oslash,\ \sigma \to \Lambda,\ \sigma \to (\sigma \vee \sigma),\ \sigma \to (\sigma \cdot \sigma),\ \sigma \to \langle\sigma\rangle\} \ .$$

We call $\mathsf{L}(\mathcal{G}_{reg})$ the language of ***regular expressions*** over Σ. Thus, we have defined the syntax of regular expressions. Next, we define the interpretation of regular expressions. Let $T, T_1, T_2 \in \mathsf{L}(\mathcal{G}_{reg})$, we define inductively $\mathsf{L}(T)$ as follows.

IB: $\mathsf{L}(\underline{x}) = \{x\}$ for all $x \in \Sigma$, $\mathsf{L}(\oslash) = \emptyset$ and $\mathsf{L}(\Lambda) = \{\lambda\}$.

IS: $\mathsf{L}(T_1 \vee T_2) = \mathsf{L}(T_1) \cup \mathsf{L}(T_2)$

$\mathsf{L}(T_1 \cdot T_2) = \mathsf{L}(T_1)\mathsf{L}(T_2)$

$\mathsf{L}(\langle T \rangle) = \mathsf{L}(T)^*$.

Theorem 4.2. *Let Σ be any fixed alphabet and let $\mathcal{G}_{reg}$ be defined as above. Then we have: A language $\mathsf{L} \subseteq \Sigma^*$ is regular if and only if there exists a regular expression T such that $\mathsf{L} = \mathsf{L}(T)$.*

Proof. Sufficiency. Let $T \in \mathsf{L}(\mathcal{G}_{reg})$ be any regular expression. We have to show that $\mathsf{L}(T)$ is regular. This is done inductively over T.

For the induction base everything is clear, since all singleton languages, the empty language $\emptyset$ and the language $\{\lambda\}$ are obviously regular.

The induction step is also clear, since by Theorem 2.1 we already know that the regular languages are closed under union, product and Kleene closure. This proves the sufficiency.

Next, we define what is meant by prefix, suffix and substring, respectively. Let $w, y \in \Sigma^*$. We write $w \sqsubseteq y$ if there exists a string $v \in \Sigma^*$ such that $wv = y$. If $w \sqsubseteq y$, then we call w a ***prefix*** of y. Furthermore, if $v \neq \lambda$, then w is said to be a ***proper prefix*** of y. In this case we write $w \sqsubset y$. Analogously, we call w a ***suffix*** of y if there exists a string $v \in \Sigma^*$ such that $vw = y$ and if $v \neq \lambda$, then w is said to be a ***proper suffix*** of y. Finally, w is said to be a ***substring*** of y if there exist strings $u, v \in \Sigma^*$ such that $uwv = y$.

For showing the necessity, let $L \in \mathcal{REG}$ be arbitrarily fixed. We have to construct a regular expression T such that $L = L(T)$.

Since $L \in \mathcal{REG}$, there exists a deterministic finite automaton $\mathcal{A} = [\Sigma, Q, \delta, q_1, F]$ such that $L = L(\mathcal{A})$. Let $Q = \{q_1, \ldots, q_m\}$ and let $F \subseteq Q$. We distinguish the following cases.

Case 1. $F = \emptyset$.

In this case we can directly conclude $L = L(\mathcal{A}) = \emptyset$. Thus, clearly there exists $T \in L(\mathcal{G}_{reg})$ such that $L = L(T) = \emptyset$, i.e., $T = \oslash$ will do it.

Case 2. $F \neq \emptyset$.

By Definition 14 we can write

$$L(\mathcal{A}) = \bigcup_{q \in F} \{s |\ s \in \Sigma^* \text{ and } \delta(q_1, s) = q\} .$$

Now, we can decompose $\mathcal{A}$ into automata $\mathcal{A}_q = [\Sigma, Q, \delta, q_1, \{q\}]$, where $q \in F$ and F is the set of accepting states from automaton $\mathcal{A}$. Thus, we obtain

$$L(\mathcal{A}) = \bigcup_{q \in F} L(\mathcal{A}_q) .$$

Therefore, it suffices to construct for each automaton $\mathcal{A}_q$ a regular expression T_q such that $L(T_q) = L(\mathcal{A}_q)$, since then $L(\mathcal{A}) = L\left(\bigvee_{q \in F} T_q\right)$.

Let $i, j, k \leqslant m$ and define

$$\begin{aligned} L_{i,j}^k \;=\; & \{s |\ s \in \Sigma^* \text{ and } \delta(q_i, s) = q_j \text{ and} \\ & \forall t \forall r[\lambda \neq r \sqsubset s \text{ and } \delta(q_i, r) = q_t \text{ implies } t \leqslant k]\} \end{aligned}$$

We are now going to show that for each $L_{i,j}^k$ there exists a regular expression $T_{i,j}^k$ such that $L_{i,j}^k = L(T_{i,j}^k)$. This will complete the proof, since $L(\mathcal{A}_q) = L_{1,n}^m$, provided $q = q_n$, $n \leqslant m$.

The proof is done by induction on k. For the induction basis, let $k = 0$. Then

$$\begin{aligned} L^0_{i,j} &= \{s \mid s \in \Sigma^* \text{ and } \delta(q_i, s) = q_j \text{ and} \\ &\quad \forall t \forall r[\lambda \neq r \sqsubset s \text{ and } \delta(q_i, r) = q_t \text{ implies } t \leqslant 0\} \\ &= \{x \mid x \in \Sigma \text{ and } \delta(q_i, x) = q_j\} . \end{aligned}$$

That means, either we have $L^0_{i,j} = \emptyset$ or $L^0_{i,j}$ is a finite set of strings of length 1, i.e., $\mathrm{card}(L^0_{i,j}) \leqslant \mathrm{card}(\Sigma)$. If $L^0_{i,j} = \emptyset$ we set $T^0_{i,j} = \oslash$ and we are done. If $L^0_{i,j} \neq \emptyset$, say $L^0_{i,j} = \{x^{(1)}, \ldots, x^{(z)}\}$, where $z \leqslant \mathrm{card}(\Sigma)$, we set

$$T^0_{i,j} = \underline{x}^{(1)} \vee \cdots \vee \underline{x}^{(z)} ,$$

and, by the inductive definition of $L(T)$, we directly get $L^0_{i,j} = L(T^0_{i,j})$. This proves the induction basis.

Now, assume the induction hypothesis that $L^k_{i,j}$ is regular and there is a regular expression $T^k_{i,j}$ such that $L^k_{i,j} = L(T^k_{i,j})$ for all $i, j \leqslant m$ and $k < m$.

For the induction step, it suffices to construct a regular expression $T^{k+1}_{i,j}$ such that $L^{k+1}_{i,j} = L(T^{k+1}_{i,j})$.

This is done via the following lemma.

Lemma 4.3. $L^{k+1}_{i,j} = L^k_{i,j} \cup L^k_{i,k+1} \left(L^k_{k+1,k+1}\right)^* L^k_{k+1,j}$

The "$\supseteq$" direction is obvious. For showing

$$L^{k+1}_{i,j} \subseteq L^k_{i,j} \cup L^k_{i,k+1} \left(L^k_{k+1,k+1}\right)^* L^k_{k+1,j}$$

let $s \in L^{k+1}_{i,j}$, say $s = x_1 x_2 \cdots x_\ell$. We consider the sequence of states reached when successively processing s, i.e., $q_i, q^{(1)}, q^{(2)}, \ldots, q^{(\ell-1)}, q_j$. We distinguish the following cases.

Case α. $q_i, q^{(1)}, q^{(2)}, \ldots, q^{(\ell-1)}, q_j$ *does not contain* q_{k+1}.

Then, clearly $s \in L^k_{i,j}$ and we are done.

Case β. $q_i, q^{(1)}, q^{(2)}, \ldots, q^{(\ell-1)}, q_j$ *contains* q_{k+1}.

Now, we may depict the situation as follows.

$$q_i \xrightarrow{u} q_{k+1} \xrightarrow{s_1} q_{k+1} \xrightarrow{s_2} q_{k+1} \cdots q_{k+1} \xrightarrow{s_\mu} q_{k+1} \xrightarrow{v} q_j$$

More formally, let u be the shortest non-empty prefix of s such that $\delta(q_i, u) = q_{k+1}$ and let $s_1, \ldots, s_\mu$ and v be all strings such that

$$\begin{aligned} s &= u s_1 s_2 \cdots s_\mu v \quad \text{and} \\ \delta(q_i, u) &= \delta(q_i, u s_1) = \delta(q_i, u s_1 s_2) = \cdots = \delta(q_i, u s_1 s_2 \cdots s_\mu) = q_{k+1} \end{aligned}$$

Hence, $u \in L^k_{i,k+1}$ and $s_1, \ldots, s_\mu \in L^k_{k+1,k+1}$ as well as $v \in L^k_{k+1,j}$. Consequently, we arrive at

$$s \in L^k_{i,j} \cup L^k_{i,k+1} \left(L^k_{k+1,k+1}\right)^* L^k_{k+1,j} .$$

Therefore, we set

$$T^{k+1}_{i,j} = T^k_{i,j} \vee T^k_{i,k+1} \cdot \langle T^k_{k+1,k+1} \rangle \cdot T^k_{k+1,j} ,$$

and the induction step is shown. This completes the proof of Theorem 4.2. ∎

So, we succeeded to characterize the regular languages in purely algebraic terms. Besides their mathematical beauty, regular expressions are also of fundamental practical importance. We shall come back to this point in the next lecture.

For now, we shall continue with another important property of regular languages which is often stated as ***Pumping Lemma***.

Lemma 4.4. *For every infinite regular language* L *there is a number* $k \in \mathbb{N}$ *such that for all* $w \in L$ *with* $|w| \geqslant k$ *there are strings* s, r, u *such that* $w = sru$, $r \neq \lambda$ *and* $sr^iu \in L$ *for all* $i \in \mathbb{N}^+$.

Proof. Let L be any infinite regular language. Then there exists a deterministic finite automaton $\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ such that $L = L(\mathcal{A})$.

Let $n = \mathrm{card}(Q)$. We show that $k = n + 1$ satisfies the conditions of the lemma.

Let $w \in L$ such that $|w| \geqslant k$. Then there must be strings $s, r, u \in \Sigma^*$ such that $r \neq \lambda$, $w = sru$ and $q_* =_{df} \delta(q_0, s) = \delta(q_0, sr)$. Consequently, $q_* = \delta(q_0, sr^i)$ for all $i \in \mathbb{N}^+$. Because of $\delta(q_0, sru) \in F$ we conclude $\delta(q_0, sr^iu) \in F$ and thus the Pumping Lemma is shown. ∎

Exercise 8. *Prove the Pumping Lemma directly by using regular grammars instead of finite automata.*

As an example for the application of the pumping lemma, we consider again the language $L = \{a^ib^i \mid n \in \mathbb{N}^+\}$. We claim that $L \notin \mathcal{REG}$. Suppose the converse. Then, by the Pumping Lemma, there must be a number k such that for all $w \in L$ with $|w| \geqslant k$ there strings s, r, u such that $w = sru$, $r \neq \lambda$ and $sr^iu \in L$ for all $i \in \mathbb{N}^+$. So, let $w = a^kb^k = qru$. We distinguish the following cases.

Case 1. $r = a^ib^j$ for some $i, j \in \mathbb{N}^+$.

Then, we directly get $srru = a^{k-i}a^ib^ja^ib^jb^{k-j}$, i.e., $srru \notin L$, a contradiction.

Case 2. $r = a^i$ for some $i \in \mathbb{N}^+$.

Then, we directly get $srru = a^{k-i}a^ia^ib^k = a^{k+i}b^k$. Again, $srru \notin L$, a contradiction.

Case 3. $r = b^i$ for some $i \in \mathbb{N}^+$.

This case can be handled analogously to Case 2. Thus, we conclude $L \notin \mathcal{REG}$.

Now, you should try it yourself.

Exercise 9. *Prove or disprove: the language of regular expressions is not regular, i.e.,* $L(\mathcal{G}_{reg}) \notin \mathcal{REG}$.

Note that the pumping lemma provides a necessary condition for a language to be regular. The condition is not sufficient.

Exercise 10. *Show that there is a language* $L \notin \mathcal{REG}$ *such that* L *satisfies the conditions of the Pumping Lemma.*

Additionally, using the ideas developed so far, we can show another important property.

Theorem 4.5. *There is an algorithm which on input any regular grammar $\mathcal{G}$ decides whether or not $L(\mathcal{G})$ is infinite.*

Proof. Let $\mathcal{G}$ be a regular grammar. The algorithm first constructs a deterministic finite automaton $\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ such that $L(\mathcal{G}) = L(\mathcal{A})$. Let $\mathrm{card}(Q) = n$.

Then, the algorithm checks whether or not there is a string s such that $n + 1 \leqslant |s| \leqslant 2n + 2$ with $s \in L(\mathcal{A})$.

If not, then output "$L(\mathcal{G})$ is finite."

Otherwise output "$L(\mathcal{G})$ is infinite."

It remains to show that the algorithm always terminates and works correctly.

Since the construction of a deterministic finite automaton from a given grammar is constructive, this step will always terminate. Furthermore, the number of strings s satisfying $n + 1 \leqslant |s| \leqslant 2n + 2$ is finite. Thus, the test will terminate, too.

If there is such a string s with $n + 1 \leqslant |s| \leqslant 2n + 2$ and $s \in L(\mathcal{A})$, then by the proof of the Pumping Lemma we can directly conclude that $L(\mathcal{G})$ is infinite.

Finally, suppose there is no such string s but $L(\mathcal{G})$ is infinite. Then, there must be at least one string $w \in L(\mathcal{G})$ with $|w| > 2n + 2$. Since $\mathrm{card}(Q) = n$, it is obvious that $\mathcal{A}$, when processing w must reach some states more than once. Now, we can cut off sufficiently many substrings of w that transfer one of the states into itself. The resulting string w' must then have a length between $n+1$ and $2n+2$, a contradiction. So, we can conclude that $w \in L(\mathcal{G})$ implies $|w| \leqslant n$, and thus $L(\mathcal{G})$ is finite. ∎

Now, apply the knowledge gained so far to solve the following exercises.

Exercise 11. *Prove or disprove: there is algorithm which on input any regular grammar $\mathcal{G}$ decides whether or not $L(\mathcal{G})$ is finite.*

Exercise 12. *Prove or disprove: there is algorithm which on input any regular grammar $\mathcal{G}$ decides whether or not $L(\mathcal{G}) = \emptyset$.*

Exercise 13. *Prove or disprove: for all $L_1, L_2 \in \mathcal{REG}$ we have $L_1 \cap L_2 \in \mathcal{REG}$.*

Exercise 14. *Prove or disprove: there is algorithm which on input any regular grammars $\mathcal{G}_1$, $\mathcal{G}_2$ decides whether or not $L(\mathcal{G}_1) \cap L(\mathcal{G}_2) = \emptyset$.*

Exercise 15. *Prove or disprove: For all $L_1, L_2 \in \mathcal{REG}$ we have $L_1 \setminus L_2 \in \mathcal{REG}$.*

Exercise 16. *Prove or disprove: there is algorithm which on input any regular grammars $\mathcal{G}_1$, $\mathcal{G}_2$ decides whether or not $L(\mathcal{G}_1) \subseteq L(\mathcal{G}_2)$.*

Exercise 17. *Prove or disprove: there is algorithm which on input any regular grammars $\mathcal{G}_1$, $\mathcal{G}_2$ decides whether or not $L(\mathcal{G}_1) = L(\mathcal{G}_2)$.*

Lecture 5: Regular Expressions in UNIX

Within this lecture we want to deal with applications of the theory developed so far. In particular, we shall have a look at regular expressions as used in UNIX. Before we see the applications, we introduce the UNIX notation for extended regular expressions. Note that the full UNIX extensions allow to express certain non-regular languages. We shall not consider these extensions here. Also note that the "basic" UNIX syntax for regular expressions is now defined as obsolete by POSIX, but is still widely used for the purposes of backwards compatibility. Most UNIX utilities (`grep`, `sed` ...) use it by default. Note that `grep` stands for "**g**lobal search for a **r**egular **e**xpression and **p**rint out matched lines," and `sed` for "**s**tream **ed**itor."

Most real applications deal with the ASCII[‡] character set that contains 128 characters. Suppose we have the alphabet $\{\mathtt{a}, \mathtt{b}\}$ and want to express "any character." Then we could simply write $\underline{\mathtt{a}} \vee \underline{\mathtt{b}}$. However, if we have 128 characters, expressing "any character" in the same way would result in a very long expression, since we have to list all characters. Thus, UNIX regular expressions allow us to write ***character classes*** to represent large sets of characters succinctly. The rules for character classes are:

- The symbol . (dot) stands for any single character.
- The sequence $[\mathtt{a}_1\mathtt{a}_2\cdots\mathtt{a}_k]$ stands for the regular expression

  $$\mathtt{a}_1 \vee \mathtt{a}_2 \vee \cdots \vee \mathtt{a}_k$$

  This notation save half the characters we have to write, since we omit the $\vee$ sign.
- Between the square braces we can put a range of the form $\mathtt{a}-\mathtt{d}$ to mean all the characters from $\mathtt{a}$ to $\mathtt{d}$ in the ASCII sequence. Thus,

  $[\mathtt{a}-\mathtt{z}]$ matches any lowercase letter. So, if we want to express the set of all letters and digits, we can shortly write $[\mathtt{A}-\mathtt{Z}\mathtt{a}-\mathtt{z}0-9]$.
- Square braces or other characters that have a special meaning in UNIX regular expressions are represented by preceding them with a backslash \.
- [^] matches a single character that is not contained within the brackets. For example, $[\hat{}\,\mathtt{a}-\mathtt{z}]$ matches any single character that is not a lowercase letter.
- ^matches the start of the line and $ the end of the line, respectively.
- \(\) is used to treat the expression enclosed within the brackets as a single block.

[‡]ASCII stands for "American Standard Code for Information Interchange." It has been introduced in 1963, became a standard in 1967, and was last updated in 1986. It uses a 7-bit code.

- * matches the last block zero or more times, i.e., it stands for $\langle\ \rangle$ in our notation. For example, we can write \(abc \)* to match the empty string λ, or abc, abcabc, abcabcabc, and so on.
- \{x, y\} matches the last block at least x and at most y times. Consequently, a\{3, 5\}matches aaa, aaaa or aaaaa.
- There is no representation of the set union operator in this syntax.

The more modern UNIX regular expressions can often be used with modern UNIX utilities by including the command line flag "-E".

POSIX' extended regular expressions are similar in syntax to the traditional UNIX regular expressions, with some exceptions. The following meta-characters are added:

- | is used instead of the operator $\vee$ to denote union.
- The operator ? means "zero or one of" the last block, thus ba? matches b or ba.
- The operator + means "one or more of." Thus, R+ is a shorthand for R$\langle$R$\rangle$ in our notation.
- Interestingly, backslashes are removed in the more modern UNIX regular expressions, i.e., \(\) becomes ( ) and \{ \} becomes { }.
- Note that we can also omit the second argument in $\{x, y\}$ if $x = y$, thus a{5} stands for aaaaa.

Also, as you have hopefully recognized, one just uses the usual characters to write the regular expressions down and not the underlined symbols we had used, i.e., one simply writes a instead of $\underline{a}$. Furthermore, the $\cdot$ is also omitted.

Finally, it should be noted that Perl has a much richer syntax than even the extended POSIX regular expressions. This syntax has also been used in other utilities and applications. Although still named "regular expressions", the Perl extensions give an expressive power that far exceeds the regular languages.

Next, we look at some applications.

5.1. Lexical Analysis

One of the oldest applications of regular expressions was in specifying the component of a compiler called "lexical analyzer." This component scans the source program and recognizes *tokens*, i.e., those substrings of consecutive characters that belong together logically. Keywords and identifiers are common examples of tokens but there are many others.

The UNIX command `lex` and its GNU version `flex`, accept as input a list of regular expressions, in the UNIX style, each followed by a bracketed section of code indicating what the lexical analyzer should do when it finds an instance of that token. Such a facility is called *lexical-analyzer generator*, because it takes as input a high-level description of a lexical analyzer and produces from it a function that is working as lexical analyzer.

Commands like `lex` and its GNU version `flex` are very useful, since the regular expression notation is exactly as powerful as needed to describe tokens. These commands are able to use the regular-expression-to-DFA algorithm to generate an efficient function that breaks source programs into tokens. The main advantage is the code writing, since regular expressions are much easier to write than a deterministic finite automaton. Also, if we need to change something, then changing a regular expression is often quite simple, while changing the code implementing a deterministic finite automaton can be a nightmare.

Example. Consider the following regular expression:

$$(0|1)^*1(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)(0|1)$$

You may try to design a deterministic finite automaton that accepts the language described by this regular expression. But please plan to use the whole week-end for doing it, since it may have much more states than you may expect. For seeing this, you should start with much shorter versions of this regular expression, i.e., by looking at

$$(0|1)^*1(0|1)$$

$$(0|1)^*1(0|1)(0|1)$$

$$(0|1)^*1(0|1)(0|1)(0|1)$$

and so on. Now, try it yourself. Provide a regular expression such that it deterministic finite automaton has roughly 32,000,000 many states.

Now, let us come back to lexical analyzers. Figure 5.1 provides an example of partial input to the `lex` command.

```
else                                {return(ELSE);}
[A-Za-z][A-Za-z0-9]*                {code to enter the found identifier
                                    {in the symbol table;
                                    {return(ID);
                                    }
>=                                  {return(GE);}
=                                   {return(EQ);}
...
```

Figure 5.1: A sample of `lex` input

The first line handles the keyword `else` and the action is to return a symbolic constant, i.e., `ELSE` in this example.

The second line contains a regular expression describing identifiers: a letter followed by zero or more letters and/or digits. The action is to enter the found identifier to the symbol table if not already there and to return the symbolic constant `ID`, which has been chosen in this example to represent identifiers.

The third line is for the sign `>=`, a two character operator. The last line is for the sign `=`, a one character operator. In practice, there would appear expressions describing each of the keywords, each of the signs and punctuation symbols like commas and parentheses, and families of constants such as numbers and strings.

5.2. Finding Patterns in Text

You probably often use your text editor (or the UNIX program `grep`) to find some text in a file (e.g., the place where you defined your depth first search program). A closely related problem is to filter or to find suitable web-pages.

How do these tools work?

There are two commonly used algorithms: Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM). Both use similar ideas. Both take linear time: $O(m + n)$ where m is the length of the search string, and n is the length of the file. Both only test whether certain characters are equal or unequal, they do not do any complicated arithmetic on characters.

Boyer-Moore is a little faster in practice, but more complicated. Knuth-Morris-Pratt is simpler, so we shortly discuss it here.

Suppose we want to `grep` the string `saso`. The idea is to construct a deterministic finite automaton which stores in its current state the information we need about the string seen so far. Suppose the string seen so far is "nauvwxy", then we need to know two things.

1. Did we already match the string we are looking for (`saso`)?

2. If not, could we possibly be in the middle of a match?

If we are in the middle of a match, we also need to know how much of the string we are looking for we have already seen. So we want our states to be partial matches to the pattern. The possible partial matches to `saso` are λ, `s`, `sa`, `sas`, or the complete match `saso` itself. In other words, we have to take into account all prefixes including the empty one of our string. If the string has length m, then there are $m + 1$ such prefixes. Thus, we need $m + 1$ states for our automaton to memorize the possible partial matches. The start and the accept state are obvious, i.e., the empty match and the full match, respectively.

In general, the transition from state plus character to state is the longest string that is simultaneously a prefix of the original pattern and a suffix of the state plus character we have just seen. For example, if we have already seen `sas` but the next character is `a`, then we only have the partial match `sa`. Using the ideas outlined above, we are able to produce the deterministic finite automaton we are looking for. It is displayed in Figure 5.2.

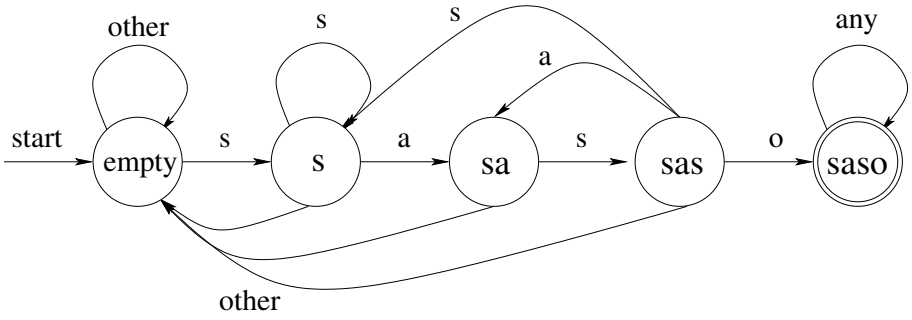

Figure 5.2: A finite automaton accepting all strings containing `saso`

The description given above is sufficient to get a string matching algorithm that takes time $O(m^3+n)$. Here we need time $O(m^3)$ to build the the state table described above, and $O(n)$ to simulate it on the input file. There are two tricky points to the KMP algorithm. First, it uses an alternate representation of the state table which takes only $O(m)$ space (the one above could take $O(m^2)$). And second, it uses a complicated loop to build the whole thing in $O(m)$ time. But these tricks you should have already seen in earlier courses. If not, please take a look into the reference [1] given below.

Reference

[1] M. CROCHEMORE AND W. RYTTER, *Jewels of Stringology, Text Algorithms*, World Scientific, New Jersey, London, Singapore Hong Kong, 2003.

Lecture 6: Context-Free Languages

Context-free languages were originally conceived by Noam Chomsky as a way to describe natural languages. That promise has not been fulfilled. However, context-free languages have found numerous applications in computer science, e.g., for designing programming languages, for constructing parsers for them, and for mark-up languages. The reason for the wide use of context-free grammars is that they represent the best compromise between power of expression and ease of implementation. Regular languages are too weak for many applications, since they cannot describe situations such as checking that the number of `begin` and `end` statements in a text are equal, since this would be a variant of the language $\{a^i b^i \mid i \in \mathbb{N}\}$ (cf. Theorem 3.3).

We therefore continue with a closer look at context-free languages. First, we need the following definitions.

Definition 18. *A grammar $G = [T, N, \sigma, P]$ is said to be **context-free** provided for all $\alpha \to \beta$ in P we have $\alpha \in N$ and $\beta \in (N \cup T)^*$.*

Definition 19. *A language L is said to be **context-free** if there exists a context-free grammar G such that $L = L(G)$. By $\mathcal{CF}$ we denote the set of all context-free languages.*

Our first theorem shows that the set of all context-free languages is richer than the set of regular languages.

Theorem 6.1. $\mathcal{RE}\mathcal{G} \subset \mathcal{CF}$.

Proof. Clearly, by definition we have that every regular grammar is also a context-free grammar. Thus, we can conclude $\mathcal{RE}\mathcal{G} \subseteq \mathcal{CF}$.

For seeing $\mathcal{CF} \setminus \mathcal{RE}\mathcal{G} \neq \emptyset$ it suffices to consider the language $L = \{a^i b^i \mid i \in \mathbb{N}\}$. By Theorem 3.3 we already know $L \notin \mathcal{RE}\mathcal{G}$. Thus, all we have to do is to provide a context-free grammar G for L. We define

$$G = [\{a, b\}, \{\sigma\}, \sigma, \{\sigma \to a\sigma b, \sigma \to \lambda\}] .$$

We leave it as an exercise to formally prove $L = L(G)$. ∎

Exercise 18. *Construct context-free grammars for the following languages:*

(1) $L = \{a^{3i} b^i \mid i \in \mathbb{N}^+\}$,

(2) $L = \{a^i b^j \mid i, j \in \mathbb{N}^+ \text{ and } i \geqslant j\}$,

(3) $L = \{s \mid s \in \{a, b\}^* \text{ and number of } a's \text{ in } s \text{ equals the number of } b's \text{ in } s\}$.

(4) *Prove or disprove the following language to be context-free*

$L = \{a^i c^k d^k b^i \mid i, k \in \mathbb{N}^+\}$.

Next, we study closure properties of context-free languages.

6.1. Closure Properties for Context-Free Languages

A first answer is given by showing a theorem analogous to Theorem 2.1.

Theorem 6.2. *The context-free languages are closed under union, product and Kleene closure.*

Proof. Let L_1 and L_2 be any context-free languages. Since L_1 and L_2 are context-free, there are context-free grammars $\mathcal{G}_1 = [T_1, N_1, \sigma_1, P_1]$ and $\mathcal{G}_2 = [T_2, N_2, \sigma_2, P_2]$ such that $L_i = L(\mathcal{G}_i)$ for $i = 1, 2$. Without loss of generality, we may assume that $N_1 \cap N_2 = \emptyset$ for otherwise we simply rename the nonterminals appropriately.

We start with the union. We have to show that $L = L_1 \cup L_2$ is context-free. This is done in the same way as in the proof of Theorem 2.1.

Next, we deal with the product. We define

$$\mathcal{G}_{prod} = [T_1 \cup T_2, N_1 \cup N_2 \cup \{\sigma\}, \sigma, P_1 \cup P_2 \cup \{\sigma \ \rightarrow \ \sigma_1\sigma_2\}] \ .$$

Note that the new production $\sigma \ \rightarrow \ \sigma_1\sigma_2$ is context-free. Using the same ideas *mutatis mutandis* as in the proof of Theorem 2.1, one easily verifies $L(\mathcal{G}_{prod}) = L_1L_2$.

Finally, we have to deal with Kleene closure. Let L be a context-free language, and let $\mathcal{G} = [T, N, \sigma, P]$ be a context-free grammar such that $L = L(\mathcal{G})$. We have to show that L^* is context-free. Recall that $L^* = \bigcup_{i \geqslant 0} L^i$. Since $L^0 = \{\lambda\}$, we have to make sure that λ can be generated. This is obvious if $\lambda \in L$. Otherwise, we simply add the production $\sigma \ \rightarrow \ \lambda$. Now, it suffices to define

$$\mathcal{G}^* = [T, N \cup \{\sigma^*\}, \sigma^*, P \cup \{\sigma^* \ \rightarrow \ \sigma\sigma^*, \ \sigma^* \ \rightarrow \ \lambda\}].$$

Again, we leave it as an exercise to show that $L(\mathcal{G}^*) = L^*$. ∎

Next, we show that $\mathcal{CF}$ is also closed under transposition. For doing it, we introduce the notation $h \overset{m}{\Rightarrow} w$ to denote a derivation of length m, i.e., we write $h \overset{m}{\Rightarrow} w$ if w can be derived from h within exactly m steps.

Additionally, we need the following lemma.

Lemma 6.3. *Let $\mathcal{G} = [T, N, \sigma, P]$ be a context-free grammar and let $\alpha, \beta \in (N \cup T)^*$. If $\alpha \overset{m}{\Rightarrow} \beta$ for some $m \geqslant 0$ and if $\alpha = \alpha_1 \cdots \alpha_n$ for some $n \geqslant 1$, where $\alpha_i \in (N \cup T)^*$ for $i = 1, \ldots, n$, then there exist $t_i \geqslant 0$, $\beta_i \in (N \cup T)^*$ for $i = 1, \ldots, n$ such that $\beta = \beta_1 \cdots \beta_n$ and $\alpha_i \overset{t_i}{\Rightarrow} \beta_i$ and $\sum_{i=1}^{n} t_i = m$.*

Proof. The proof is done by induction on m. For the induction basis we choose $m = 0$ and get $\alpha \overset{0}{\Rightarrow} \beta$ implies $\alpha = \beta$. Thus, we can choose $\alpha_i = \beta_i$ as well as $t_i = 0$ for $i = 1, \ldots, n$ and have $\alpha_i \overset{0}{\Rightarrow} \beta_i$ for $i = 1, \ldots, n$ and $\sum_{i=1}^{n} t_i = 0$. This proves the induction basis.

Our induction hypothesis is that if $\alpha = \alpha_1 \cdots \alpha_n \overset{m}{\Rightarrow} \beta$ then there exist $t_i \geqslant 0$, $\beta_i \in (N \cup T)^*$ for $i = 1, \ldots, n$ such that $\beta = \beta_1 \cdots \beta_n$ and $\alpha_i \overset{t_i}{\Rightarrow} \beta_i$ and $\sum_{i=1}^{n} t_i = m$.

For the induction step consider

$$\alpha = \alpha_1 \cdots \alpha_n \overset{m+1}{\Longrightarrow} \beta \ .$$

Then there exists $\gamma \in (N \cup T)^*$ such that

$$\alpha = \alpha_1 \cdots \alpha_n \ \Rightarrow \ \gamma \overset{m}{\Rightarrow} \beta \ .$$

Since $\mathcal{G}$ is context-free, the production used in $\alpha \ \Rightarrow \ \gamma$ must have been of the form $h \ \rightarrow \ \zeta$, where $h \in N$ and $\zeta \in (N \cup T)^*$. Let α_k, $1 \leqslant k \leqslant m$ contain the nonterminal h that was rewritten in using $h \ \rightarrow \ \zeta$ to obtain $\alpha \ \Rightarrow \ \gamma$. Then $\alpha_k = \alpha' h \beta'$ for some $\alpha', \beta' \in (N \cup T)^*$. Furthermore, let

$$\gamma_i = \begin{cases} \alpha_i \quad , & \text{if } i \neq k, \\ \alpha' \zeta \beta', & \text{if } i = k \ . \end{cases}$$

Then, for $i \neq k$, $\alpha_i \overset{0}{\Rightarrow} \gamma_i$ and $\alpha_k \ \Rightarrow \ \gamma_k$. Thus,

$$\alpha = \alpha_1 \cdots \alpha_n \ \Rightarrow \ \gamma_1 \cdots \gamma_n \overset{m}{\Rightarrow} \beta \ .$$

By the induction hypothesis there exist t_i, β_i such that $\beta = \beta_1 \cdots \beta_n$ and $\gamma_i \overset{t_i}{\Rightarrow} \beta_i$ and $\sum_{i=1}^{n} t_i = m$. Combining the derivations we find

$$\alpha_i \overset{0}{\Rightarrow} \gamma_i \overset{t_i}{\Rightarrow} \beta_i \text{ for } i \neq k \ ,$$

and

$$\alpha_k \ \Rightarrow \ \gamma_k \overset{t_k}{\Rightarrow} \beta_k \ .$$

Finally, we set

$$t'_i = \begin{cases} t_i \quad , & \text{if } i \neq k, \\ t_k + 1, & \text{if } i = k \ . \end{cases}$$

Thus, we have $t'_i \geqslant 0$ and $\beta_i \in (N \cup T)^*$ satisfy

$$\begin{aligned} \beta \ &= \ \beta_1 \cdots \beta_n \\ \alpha_i &\overset{t'_i}{\Rightarrow} \beta_i \ . \end{aligned}$$

Finally, we compute

$$\sum_{i=1}^{n} t'_i = 1 + \sum_{i=1}^{n} t_i = m + 1 \ ,$$

and the lemma follows. ∎

Now, we are ready to prove $\mathcal{CF}$ to be closed under transposition.

Theorem 6.4. *Let Σ be any alphabet, and let $L \subseteq \Sigma^*$. Then we have: If $L \in \mathcal{CF}$ then $L^T \in \mathcal{CF}$, too.*

Proof. Let L be any context-free language. Hence, there exists a context-free grammar $\mathcal{G} = [T, N, \sigma, P]$ such that $L = L(\mathcal{G})$. Now, we define a grammar $\mathcal{G}^T$ as follows. Let

$$\mathcal{G}^T = [T, N, \sigma, P^T], \text{ where } P^T = \{\alpha^T \to \beta^T \mid (\alpha \to \beta) \in P\} .$$

Taking into account that $(\alpha \to \beta) \in P$ implies $\alpha \in N$ and $\beta \in (N \cup T)^*$ (cf. Definition 18), and that $\alpha^T = \alpha$ as well as $\beta^T \in (N \cup T)^*$, we can directly conclude that $\mathcal{G}^T$ is context-free. Thus, it remains to show that $L(\mathcal{G}^T) = L^T$. This is done via the following claim.

Claim 1. *For each $h \in N$ we have,*

$$h \overset{m}{\Rightarrow}_{\mathcal{G}} w,\ w \in (N \cup T)^* \text{ if and only if } h \overset{m}{\Rightarrow}_{\mathcal{G}^T} w^T.$$

The claim is proved by induction on m. We start with the necessity.

For the induction basis, we choose $m = 0$, and clearly have $h \overset{0}{\Rightarrow}_{\mathcal{G}} h$ if and only if $h \overset{0}{\Rightarrow}_{\mathcal{G}^T} h$, since $h = h^T$.

Now, we have the following induction hypothesis.

For all $h \in N$, if $h \overset{t}{\Rightarrow}_{\mathcal{G}} w$, $w \in (N \cup T)^*$, and $t \leqslant m$, then $h \overset{t}{\Rightarrow}_{\mathcal{G}^T} w^T$.

For the induction step, we have to show that, if $h \overset{m+1}{\Longrightarrow}_{\mathcal{G}} s$, $s \in (N \cup T)^*$, then $h \overset{m+1}{\Longrightarrow}_{\mathcal{G}^T} s^T$.

Let $h \overset{m+1}{\Longrightarrow}_{\mathcal{G}} s$, then there exists an $\alpha \in (N \cup T)^*$ such that $h \Rightarrow_{\mathcal{G}} \alpha \overset{m}{\Rightarrow}_{\mathcal{G}} s$. Let

$$\alpha = u_1 h_1 u_2 h_2 \cdots u_n h_n u_{n+1} , \text{ where } u_i \in T^* \text{ and } h_i \in N .$$

Now, since $h \Rightarrow_{\mathcal{G}} \alpha$ implies $(h \to \alpha) \in P$, we directly get by construction that $(h \to \alpha^T) \in P^T$. Hence, we can conclude $h \Rightarrow_{\mathcal{G}^T} \alpha^T$, and obtain

$$h \Rightarrow_{\mathcal{G}^T} \alpha^T = u_{n+1}^T h_n u_n^T h_{n-1} \cdots u_2^T h_1 u_1^T .$$

Furthermore, we have

$$u_1 h_1 u_2 h_2 \cdots u_n h_n u_{n+1} \overset{m}{\Rightarrow}_{\mathcal{G}} s ,$$

and therefore, by Lemma 6.3, there exist $\gamma_i \in (N \cup T)^*$ and $t_i \geqslant 0$ such that $h_i \overset{t_i}{\Rightarrow}_{\mathcal{G}} \gamma_i$, as well as

$$s = u_1 \gamma_1 u_2 \gamma_2 \cdots \gamma_n u_{n+1} ,$$

and $\sum_{i=1}^{n} t_i = m$. Consequently, $t_i \leqslant m$ and by the induction hypothesis we obtain that

$$h_i \overset{t_i}{\Rightarrow}_{\mathcal{G}} \gamma_i \quad \text{implies} \quad h_i \overset{t_i}{\Rightarrow}_{\mathcal{G}^T} \gamma_i^T .$$

Therefore,

$$h \Rightarrow_{\mathcal{G}^T} u_{n+1}^T h_n u_n^T h_{n-1} \cdots u_2^T h_1 u_1^T \overset{m}{\Rightarrow}_{\mathcal{G}^T} u_{n+1}^T \gamma_n^T u_n^T \gamma_{n-1}^T \cdots u_2^T \gamma_1^T u_1^T = s^T .$$

This completes the induction step, and thus the necessity is shown.

The sufficiency can be shown analogously by replacing $\mathcal{G}$ by $\mathcal{G}^T$ and $\mathcal{G}^T$ by $\mathcal{G}$. Thus, we obtain $L(\mathcal{G}^T) = L^T$. ∎

Next, we are going to prove that $\mathcal{CF}$ is not closed under intersection.

Theorem 6.5. *There are context-free languages* L_1 *and* L_2 *such that* $L_1 \cap L_2 \notin \mathcal{CF}$.

Proof. Let

$$\begin{aligned} L_1 &= \{a^n b^n c^m \mid n, m \in \mathbb{N}\} \quad \text{and let} \\ L_2 &= \{a^m b^n c^n \mid n, m \in \mathbb{N}\} . \end{aligned}$$

Then we directly get

$$L = L_1 \cap L_2 = \{a^n b^n c^n \mid n \in \mathbb{N}\} ,$$

but $L \notin \mathcal{CF}$ as we shall show a bit later, when we have the pumping lemma for context-free languages. ∎

The latter theorem allows a nice corollary. For stating it, we introduce the following notation. Let $L \subseteq \Sigma^*$ be any language. Then we write $\overline{L}$ to denote the ***complement*** of L, i.e., $\overline{L} =_{df} \Sigma^* \setminus L$. Now, we are ready for our corollary.

Corollary 6.6. *$\mathcal{CF}$ is not closed under complement.*

Proof. The proof is done indirectly. Suppose the converse, i.e., for all context-free languages L_1 and L_2 we also have $\overline{L_1}$, $\overline{L_2} \in \mathcal{CF}$. By Theorem 6.2 we can conclude $\overline{L_1} \cup \overline{L_2} \in \mathcal{CF}$ (closure under union). But then, by our supposition, we must also have $\overline{\overline{L_1} \cup \overline{L_2}} \in \mathcal{CF}$. Since $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$ by de Morgan's laws for sets, this would imply the context-free languages to be closed under intersection, a contradiction to Theorem 6.5. This proves the corollary. ∎

Exercise 19. *Prove or disprove the following: For all* $L_1 \in \mathcal{CF}$ *and* $L_2 \in \mathcal{REG}$ *we have* $L_1 \cap L_2 \in \mathcal{CF}$.

We continue with further properties of context-free languages that are needed later but that are also of independent interest.

First, we start with the following observation. It can sometimes happen that grammars contain nonterminals that cannot generate any terminal string. Let us look at the following example.

Example 5. Let $\Sigma = \{a, +, *, (,), -\}$ be the terminal alphabet, and assume the set of productions given is:

$$\begin{aligned} E &\to E + E \\ E &\to E + T \\ E &\to E + F \\ F &\to F * E \\ F &\to F * (T) \\ F &\to a \\ T &\to E - T \end{aligned}$$

where F is the start symbol. Now, we see that the only rule containing T on the left-hand side is $\mathsf{T} \to \mathsf{E} - \mathsf{T}$. Thus it also contains T on the right-hand side. Consequently, from T *no* terminal string can be derived. Such a situation is highly undesirable, since it will only complicate many tasks such as analyzing derivations. Therefore, we introduce the notion of a ***reduced grammar***.

Definition 20 (Reduced Grammar). *A context-free grammar $\mathcal{G} = [\mathsf{T}, \mathsf{N}, \sigma, \mathsf{P}]$ is said to be **reduced** if*

(1) *for all $\mathfrak{h} \in \mathsf{N} \setminus \{\sigma\}$ there are strings $\mathfrak{p}, \mathfrak{q} \in (\mathsf{T} \cup \mathsf{N})^*$ such that $\sigma \overset{*}{\Rightarrow} \mathfrak{p}\mathfrak{h}\mathfrak{q}$, and*

(2) *there is a string $w \in \mathsf{T}^*$ such that $\mathfrak{h} \overset{*}{\Rightarrow} w$.*

The following theorem shows the usefulness of Definition 20.

Theorem 6.7. *For every context-free grammar $\mathcal{G} = [\mathsf{T}, \mathsf{N}, \sigma, \mathsf{P}]$ there is a reduced context-free grammar $\mathcal{G}' = [\mathsf{T}', \mathsf{N}', \sigma', \mathsf{P}']$ such that $\mathsf{L}(\mathcal{G}) = \mathsf{L}(\mathcal{G}')$.*

Proof. The proof proceeds in two steps. In the first step, we collect all nonterminal symbols from which a string in T^* can be derived. Let this set be $\tilde{\mathsf{H}}$. In the second step, we collect all those symbols from $\tilde{\mathsf{H}}$ that can be reached from the start symbol σ.

For the first step, we set $\mathsf{H}_0 = \mathsf{T}$ and proceed inductively as follows.

$$\begin{aligned} \mathsf{H}_1 &= \mathsf{H}_0 \cup \{\mathfrak{h} |\ \mathfrak{h} \in \mathsf{N},\ (\mathfrak{h} \to \mathfrak{p}) \in \mathsf{P} \text{ and } \mathfrak{p} \in \mathsf{H}_0^*\} \\ \mathsf{H}_{i+1} &= \mathsf{H}_i \cup \{\mathfrak{h} |\ \mathfrak{h} \in \mathsf{N},\ (\mathfrak{h} \to \mathfrak{p}) \in \mathsf{P} \text{ and } \mathfrak{p} \in \mathsf{H}_i^*\}\ . \end{aligned}$$

By construction we obviously have

$$\mathsf{H}_0 \subseteq \mathsf{H}_1 \subseteq \mathsf{H}_2 \subseteq \cdots \subseteq \mathsf{T} \cup \mathsf{N}\ .$$

Taking into account that $\mathsf{T} \cup \mathsf{N}$ is finite, there must be an index i_0 such that $\mathsf{H}_{i_0} = \mathsf{H}_{i_0+1}$.

Next, we show the following claim.

Claim 1. *If $\mathsf{H}_i = \mathsf{H}_{i+1}$ then $\mathsf{H}_i = \mathsf{H}_{i+m}$ for all $m \in \mathbb{N}$.*

The induction basis for $m = 0$ is obvious, since $\mathsf{H}_{i+0} = \mathsf{H}_i$ by construction.

Thus, we have the induction hypothesis $\mathsf{H}_i = \mathsf{H}_{i+m}$. For the induction step we have to show that $\mathsf{H}_i = \mathsf{H}_{i+m+1}$.

Let $\mathfrak{h} \in \mathsf{H}_{i+m+1}$, then we either have $\mathfrak{h} \in \mathsf{H}_{i+m}$ (and the induction hypothesis directly applies) or $(\mathfrak{h} \to \mathfrak{p}) \in \mathsf{P}$ and $\mathfrak{p} \in \mathsf{H}_{i+m}^*$. By the induction hypothesis we know that $\mathsf{H}_i = \mathsf{H}_{i+m}$. Hence, the condition $(\mathfrak{h} \to \mathfrak{p}) \in \mathsf{P}$ and $\mathfrak{p} \in \mathsf{H}_{i+m}^*$ can be rewritten as

$$(\mathfrak{h} \to \mathfrak{p}) \in \mathsf{P} \text{ and } \mathfrak{p} \in \mathsf{H}_i^*\ .$$

But the latter condition implies by construction that $\mathfrak{h} \in \mathsf{H}_{i+1}$. By assumption, $\mathsf{H}_i = \mathsf{H}_{i+1}$ and therefore we conclude $\mathfrak{h} \in \mathsf{H}_i$. This proves Claim 1.

Next, we set $\tilde{H} =_{df} H_{i_0} \setminus T$. Then, the following claim holds by construction.

Claim 2. $h \in \tilde{H}$ *if and only if there is a* $w \in T^*$ *such that* $h \stackrel{*}{\Rightarrow} w$.

Next, we perform the second step. For doing it, we define inductively $R_0 = \{\sigma\}$,

$$\begin{aligned} R_1 &= R_0 \cup \{h \mid h \in \tilde{H} \text{ and } \exists p, q[p, q \in (T \cup N)^* \text{ and } \sigma \stackrel{1}{\Rightarrow} phq]\} \text{ and for } i \geqslant 1 \\ R_{i+1} &= R_i \cup \{h \mid h \in \tilde{H} \text{ and } \exists r_i \exists p, q[r_i \in R_i,\ p, q \in (T \cup N)^* \text{ and } r_i \stackrel{1}{\Rightarrow} phq]\} \end{aligned}$$

In the same manner as above one can easily prove that there is an $i_0 \in \mathbb{N}$ such that $R_{i_0} = R_{i_0+m}$ for all $m \in \mathbb{N}$.

Finally, we define $T' = T$, $N' = R_{i_0} \cap \tilde{H}$, $\sigma' = \sigma$ and

$$\begin{aligned} P' &= \emptyset, \quad \text{if } \sigma \notin N' \\ P' &= P \setminus \{\alpha \to \beta \mid (\alpha \to \beta) \in P,\ \alpha \notin R_{i_0} \text{ or } \beta \notin (T \cup N')^*\}, \\ &\quad \text{otherwise .} \end{aligned}$$

i.e., we get $G' = [T, N', \sigma, P']$.

By construction $L(G') = L(G)$.

Note that Claim 2 implies Condition (2) of Definition 20 and Condition (1) of Definition 20 is fulfilled by the definition of the sets R_i. ∎

We illustrate the construction of a reduced grammar by looking at two examples.

Example 6. Let $G = [\{a, b, c, d\}, \{\sigma, \alpha, \beta, \gamma, \delta\}, \sigma, P]$, where P is defined as follows.

$$\begin{array}{llll} \sigma \to \alpha\beta & & \gamma \to c\beta \\ \sigma \to \gamma\alpha & & \gamma \to b \\ \alpha \to a & & \delta \to a\delta \\ \beta \to \gamma\beta & & \delta \to d \\ \beta \to \alpha\beta & & \end{array}$$

So, we obtain:

$$\begin{aligned} H_0 &= \{a, b, c, d\} \\ H_1 &= \{a, b, c, d, \alpha, \gamma, \delta\} \\ H_2 &= \{a, b, c, d, \alpha, \gamma, \delta, \sigma\} \\ H_3 &= \{a, b, c, d, \alpha, \gamma, \delta, \sigma\} = H_2 , \end{aligned}$$

and thus we terminate. Consequently, $\tilde{H} = \{\alpha, \gamma, \delta, \sigma\}$.

Next, we compute the sets R_i, i.e., $R_0 = \{\sigma\}$, $R_1 = \{\sigma, \alpha, \gamma\} = R_2$, and we terminate again. Thus, $N' = \{\sigma, \alpha, \gamma\}$.

Finally, we delete all productions having in their left hand side a symbol not in $\{\sigma, \alpha, \gamma\}$, i.e., $\beta \to \gamma\beta$, $\beta \to \alpha\beta$, $\delta \to a\delta$, and $\delta \to d$. Then we

delete from the remaining productions those containing a symbol in their right hand side not in N', i.e., $\sigma \to \alpha\beta$ and $\gamma \to c\beta$. Thus, we obtain the grammar $G' = [\{a, b, c, d\}, \{\sigma, \alpha, \gamma\}, \sigma, \{\sigma \to \gamma\alpha,\ \alpha \to a,\ \gamma \to b\}]$.

Example 7. Let $G = [\{a\}, \{\sigma, \beta, \gamma\}, \sigma, P]$, where

$$P = \{\sigma \to \beta\gamma,\ \beta \to a,\ \gamma \to \gamma\} .$$

Then we obtain, $H_0 = \{a\}$, $H_1 = \{a, \beta\} = H_2$, and we terminate. Thus $\tilde{H} = \{\beta\}$. Next, we get $R_0 = \{\sigma\}$ and $R_1 = \{\sigma, \beta\} = R_2$. Thus, we terminate again. This is the point where we need the intersection in the definition of N', since now $N' = \{\beta\}$. Consequently, in the definition of P' the first case applies and $P' = \emptyset$.

We continue with the following definition.

Definition 21. *A grammar $G = [T, N, \sigma, P]$ is said to be* ***λ-free*** *if P does not contain any production of the form $h \to \lambda$.*

Theorem 6.8. *For every context-free grammar $G = [T, N, \sigma, P]$ there exists a context-free grammar G' such that $L(G') = L(G) \setminus \{\lambda\}$ and G' is λ-free.*

Furthermore, if $\lambda \in L(G)$ then there exists an equivalent context-free grammar G'' such that $\sigma'' \to \lambda$ is the only production having λ on its right-hand side and σ'' does not occur at any right-hand side.

We leave it as an exercise to prove this theorem.

Lecture 7: Further Properties of Context-Free Languages

We shall start this lecture by defining the so-called BNF. Then we take a look at parse trees. Furthermore, within this lecture we shall define the first normal form for context-free grammars, i.e., the Chomsky Normal Form. Then we state the pumping lemma for context-free languages which is often referred to as the Lemma of Bar-Hillel. Finally, we discuss some consequences.

7.1. Backus-Naur Form

As mentioned in the last lecture, context-free grammars are of fundamental importance for programming languages. However, in the specification of programming languages usually a form different to the one provided in Definition 18 is used. This form is the so-called *Backus normal form* or *Backus-Naur Form*. It was created by John Backus to specify the grammar of ALGOL. Later it has been simplified by Peter Naur to reduce the character set used and Donald Knuth proposed to call the new form Backus-Naur Form. Fortunately, whether or not one is following Knuth's suggestion, the form is commonly abbreviated BNF.

The form uses four meta characters that are not allowed to appear in the working vocabulary, i.e., in $\mathsf{T} \cup \mathsf{N}$ in our definition. These meta characters are

$$< \quad > \quad ::= \quad |$$

and the idea to use them is as follows. Strings (*not* containing the meta characters) are enclosed by $<$ and $>$ denote nonterminals.The symbol $::=$ serves as a replacement operator (in the same way as $\rightarrow$) and | is read as "or."

The following example is from Harrison [1]. Consider an ordinary context-free grammar for unsigned digits in a programming language. Here, D stands for the class of digits and U for the class of unsigned integers.

$$\begin{array}{ll} \mathsf{D} \rightarrow 0 & \mathsf{D} \rightarrow 5 \\ \mathsf{D} \rightarrow 1 & \mathsf{D} \rightarrow 6 \\ \mathsf{D} \rightarrow 2 & \mathsf{D} \rightarrow 7 \\ \mathsf{D} \rightarrow 3 & \mathsf{D} \rightarrow 8 \\ \mathsf{D} \rightarrow 4 & \mathsf{D} \rightarrow 9 \\ \mathsf{U} \rightarrow \mathsf{D} & \mathsf{U} \rightarrow \mathsf{U}\mathsf{D} \ . \end{array}$$

Rewriting this example in BNF yields:

$$\begin{array}{rcl} < \texttt{digit} > & ::= & 0|1|2|3|4|5|6|7|8|9 \\ < \texttt{unsigned integer} > & ::= & < \texttt{digit} > | < \texttt{unsigned integer} >< \texttt{digit} > \end{array}$$

As this example clearly shows the BNF allows a very compact representation of the grammar. This is of particular importance when defining the syntax of a programming language, where the set of productions usually contains many elements.

Whenever appropriate, we shall adopt a blend of the notation used in BNFs, i.e., occasionally we shall use | as well as < and > but not ::=.

Context-free languages play an important role in many applications. As far as regular languages are concerned, we have seen that finite automata are very efficient recognizers. So, what about context-free languages? Again, for every context-free language a recognizer can be algorithmically constructed. Formally, these recognizers are ***pushdown automata***. There are many software systems around that perform the construction of the relevant pushdown automaton for a given language. These systems are important in that they allow the quick construction of the syntax analysis part of a compiler for a new language and are therefore highly valued. We shall come back to this topic later (see Lectures 9 and 10), since it is better to treat another important tool for syntax analysis first, i.e., ***parsers***.

One of the most widely used of these syntax analyzer generators is called `yacc` (yet another compiler-compiler). The generation of a parser, i.e., a function that creates parse trees from source programs has been institutionalized in the `YACC` command that appears in all UNIX systems. The input to `YACC` is a CFG, in a notation that differs only in details from the well-known BNF. Associated with each production is an *action*, which is a fragment of `C` code that is performed whenever a node of the parse tree that (combined with its children) corresponds to this production is created. So, let us continue with parse trees.

7.2. Parse Trees, Ambiguity

A nice feature of grammars is that they describe the hierarchical syntactic structure of the sentences of languages they define. These hierarchical structures are described by ***parse trees***.

Parse trees are a representation for derivations. When used in a compiler, it is the data structure of choice to represent the source program. In a compiler, the tree structure of the source program facilitates the translation of the source program into executable code by allowing natural, recursive functions to perform this translation process.

Parse trees are trees defined as follows. Let $\mathcal{G} = [\mathsf{T}, \mathsf{N}, \sigma, \mathsf{P}]$ be a grammar. A ***parse tree*** for $\mathcal{G}$ is a tree satisfying the following conditions.

(1) Each interior node and the root are labeled by a variable from N.

(2) Each leaf is either labeled by a non-terminal, a terminal or the empty string. If the leaf is labeled by the empty string, then it must be the only child of its parent.

(3) If an interior node is labeled A and its children are labeled by $\mathsf{X}_1, \ldots, \mathsf{X}_k$, respectively, from left to right, then $\mathsf{A} \to \mathsf{X}_1 \cdots \mathsf{X}_k$ is a production from P.

Thus, every subtree of a parse tree describes one instance of an abstraction in the statement. Next, we define the yield of a parse tree. If we look at the leaves of any parse tree and concatenate them from left to right, we get a string. This string is called the ***yield*** of the parse tree.

Exercise 20. *Prove that the yield is always a string that is derivable from the start symbol provided the root is labeled by* σ.

Clearly, of special importance is the case that the root is labeled by the start symbol and that the yield is a *terminal string*, i.e., all leaves are labeled with a symbol from T or the empty string. Thus, the language of a grammar can also be expressed as the set of yields of those parse trees having the start symbol at the root and a terminal string as yield.

We continue with an example. Let us assume that we have the following part of a grammar on hand that describes how assignment statements are generated.

Example 8.

$$
\begin{array}{rcl}
\texttt{<assign>} & \to & \texttt{<id>} := \texttt{<expr>} \\
\texttt{<id>} & \to & \texttt{A} \mid \texttt{B} \mid \texttt{C} \\
\texttt{<expr>} & \to & \texttt{<id>} + \texttt{<expr>} \\
& \mid & \texttt{<id>} * \texttt{<expr>} \\
& \mid & (\texttt{<expr>}) \\
& \mid & \texttt{<id>}
\end{array}
$$

Now, let us look at the assignment statement $\texttt{A} := \texttt{B} * (\texttt{A} + \texttt{C})$

which can be generated by the following derivation.

$$
\begin{array}{rcl}
\texttt{<assign>} & \Rightarrow & \texttt{<id>} := \texttt{<expr>} \\
& \Rightarrow & \texttt{A} := \texttt{<expr>} \\
& \Rightarrow & \texttt{A} := \texttt{<id>} * \texttt{<expr>} \\
& \Rightarrow & \texttt{A} := \texttt{B} * \texttt{<expr>} \\
& \Rightarrow & \texttt{A} := \texttt{B} * (\texttt{<expr>}) \\
& \Rightarrow & \texttt{A} := \texttt{B} * (\texttt{<id>} + \texttt{<expr>}) \\
& \Rightarrow & \texttt{A} := \texttt{B} * (\texttt{A} + \texttt{<expr>}) \\
& \Rightarrow & \texttt{A} := \texttt{B} * (\texttt{A} + \texttt{<id>}) \\
& \Rightarrow & \texttt{A} := \texttt{B} * (\texttt{A} + \texttt{C})
\end{array}
$$

The structure of the assignment statement that we have just derived is shown in the parse tree displayed in Figure 7.3.

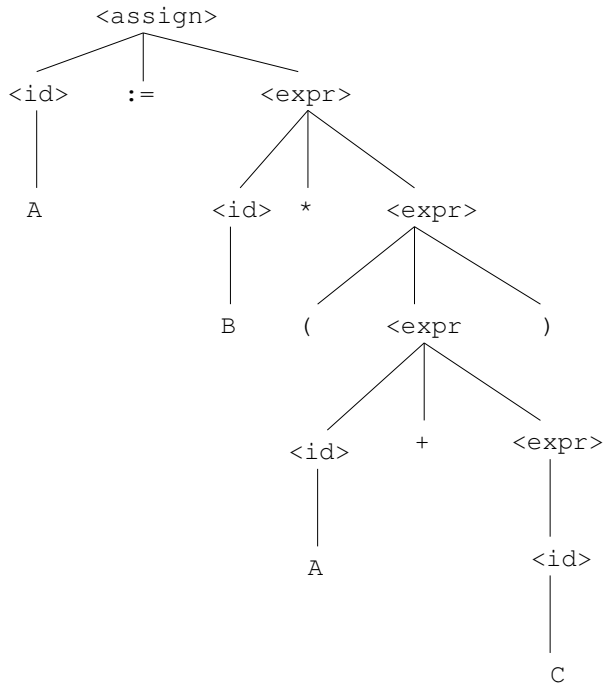

Figure 7.3: Parse tree for sentence $\mathtt{A} := \mathtt{B} * (\mathtt{A} + \mathtt{C})$

Note that syntax analyzers for programming languages, which are often called *parsers*, construct parse trees for given programs. Some systems construct parse trees only implicitly, but they also use the whole information provided by the parse tree during the parse. There are two major approaches of how to build these parse trees. One is top-down and the other one is bottom-up. In the top-down approach the parse tree is built from the root to the leaves while in the bottom-up approach the parse tree is built from the leaves upward to the root. A major problem one has to handle when constructing such parsers is *ambiguity*. We thus direct our attention to this problem.

7.2.1. Ambiguity

A grammar that generates a sentence for which there are two or more distinct parse trees is said to be ***ambiguous***. For having an example, let us look at the following part of a grammar given in Example 9. At first glance this grammar looks quite similar to the one considered above. The only difference is that the production for expressions has been altered by replacing `<id>` by `<expr>`. However, this small modification leads to serious problems, because now the grammar provides slightly less syntactic structure than the grammar considered in Example 8 does.

Example 9.

$$\begin{aligned} \texttt{<assign>} &\rightarrow \texttt{<id>} := \texttt{<expr>} \\ \texttt{<id>} &\rightarrow \mathtt{A} \mid \mathtt{B} \mid \mathtt{C} \end{aligned}$$

$$
\begin{array}{rcl}
\texttt{<expr>} & \rightarrow & \texttt{<expr>} + \texttt{<expr>} \\
& | & \texttt{<expr>} * \texttt{<expr>} \\
& | & (\texttt{<expr>}) \\
& | & \texttt{<id>}
\end{array}
$$

For seeing that this grammar is ambiguous, let us look at the following assignment statement:

$$\texttt{A} := \texttt{B} + \texttt{C} * \texttt{A} \ .$$

We skip the two formal derivations possible for this assignment and look directly at the two parse trees.

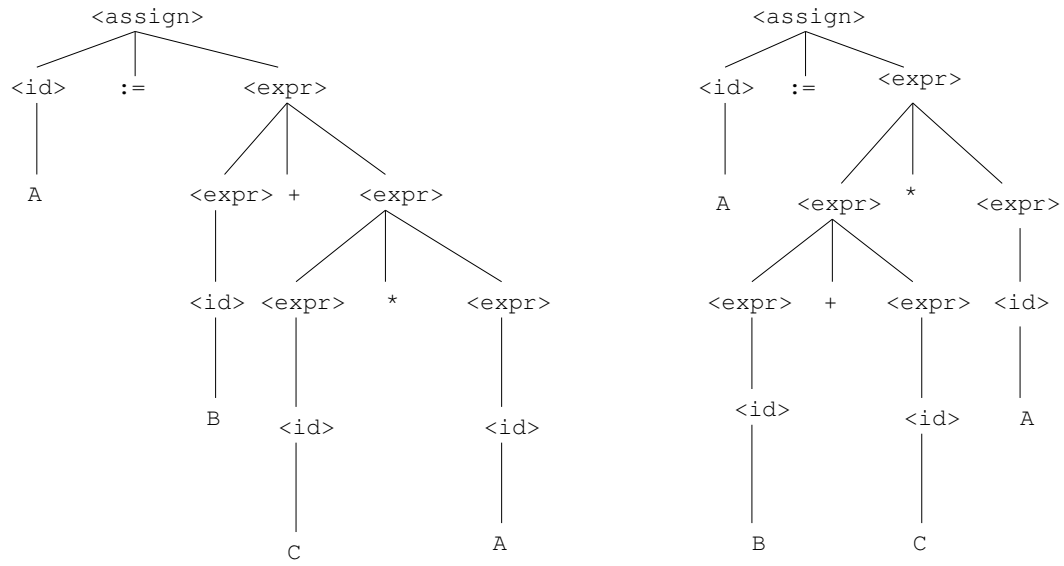

Figure 7.4: Two parse trees for the same sentence $\texttt{A} := \texttt{B} + \texttt{C} * \texttt{A}$

These two distinct parse trees cause problems because compilers base the semantics of the sentences on their syntactic structure. In particular, compilers decide what code to generate by examining the parse tree. So, in our example the *semantics* is not clear. Let us examine this problem in some more detail.

In the first parse tree (the left one) of Figure 7.4 the multiplication operator is generated lower in the tree which would indicate that it has precedence over the addition operator in the expression. The second parse tree in Figure 7.4, however, is just indicating the opposite. Clearly, in dependence on what decision the compiler makes, the result of an actual evaluation of the assignment given will be either the expected one (that is multiplication has precedence over addition) or an erroneous one.

Although the grammar in Example 8 is not ambiguous, the precedence order of its operators is not the usual one. Rather, in this grammar, a parse tree of a sentence with multiple operators has the rightmost operator at the lowest point, with the

other operators in the tree moving progressively higher as one moves to the left in the expression.

So, one has to think about a way to overcome this difficulty and to clearly define the usual operator precedence between multiplication and addition, or more generally with any desired operator precedence. As a matter of fact, this goal can be achieved for our example by using separate nonterminals for the operands of the operators that have different precedence. This not only requires additional nonterminals but also additional productions.

However, in general, the situation is much more subtle. First, there is *no* algorithm deciding whether or not any context-free grammar is ambiguous. Second, there are context-free languages that have nothing but ambiguous grammars. See Section 5.4.4 in [2] for a detailed treatment of this issue. But in practice the situation is not as grim as it may seem. Many techniques have been proposed to eliminate ambiguity in the sorts of constructs that typically appear in programming languages.

So, let us come back to our example and let us show how to eliminate the ambiguity we have detected. We continue by providing a grammar generating the same language as the grammars of Examples 8 and 9, but which clearly indicates the usual precedence order of multiplication and addition.

Example 10.

$$
\begin{array}{rcl}
\texttt{<assign>} & \rightarrow & \texttt{<id>} := \texttt{<expr>} \\
\texttt{<id>} & \rightarrow & \texttt{A} \mid \texttt{B} \mid \texttt{C} \\
\texttt{<expr>} & \rightarrow & \texttt{<expr>} + \texttt{<term>} \\
 & \mid & \texttt{<term>} \\
\texttt{<term>} & \rightarrow & \texttt{<term>} * \texttt{<factor>} \\
 & \mid & \texttt{<factor>} \\
\texttt{<factor>} & \rightarrow & (\texttt{<expr>}) \\
 & \mid & \texttt{<id>}
\end{array}
$$

Next, let us derive the same statement as above, i.e.,

$$\texttt{A} := \texttt{B} + \texttt{C} * \texttt{A}$$

The derivation is unambiguously obtained as follows:

$$
\begin{array}{rcl}
\texttt{<assign>} & \Rightarrow & \texttt{<id>} := \texttt{<expr>} \\
 & \Rightarrow & \texttt{A} := \texttt{<expr>} \\
 & \Rightarrow & \texttt{A} := \texttt{<expr>} + \texttt{<term>} \\
 & \Rightarrow & \texttt{A} := \texttt{<term>} + \texttt{<term>} \\
 & \Rightarrow & \texttt{A} := \texttt{<factor>} + \texttt{<term>} \\
 & \Rightarrow & \texttt{A} := \texttt{<id>} + \texttt{<term>}
\end{array}
$$

$$
\begin{aligned}
\Rightarrow \quad & \texttt{A} := \texttt{B} + \texttt{<term>} \\
\Rightarrow \quad & \texttt{A} := \texttt{B} + \texttt{<term>} * \texttt{<factor>} \\
\Rightarrow \quad & \texttt{A} := \texttt{B} + \texttt{<factor>} * \texttt{<factor>} \\
\Rightarrow \quad & \texttt{A} := \texttt{B} + \texttt{<id>} * \texttt{<factor>} \\
\Rightarrow \quad & \texttt{A} := \texttt{B} + \texttt{C} * \texttt{<factor>} \\
\Rightarrow \quad & \texttt{A} := \texttt{B} + \texttt{C} * \texttt{<id>} \\
\Rightarrow \quad & \texttt{A} := \texttt{B} + \texttt{C} * \texttt{A}
\end{aligned}
$$

Exercise 21. *Construct the parse tree for this sentence in accordance with the derivation given above.*

Furthermore, you should note that we have presented a ***leftmost derivation*** above, i.e., the leftmost nonterminal has always been handled first until it was replaced by a terminal. Thus, the sentence given above has more than one derivation. If we had always handled the rightmost nonterminal first, then we would have been arrived at a ***rightmost derivation***.

Exercise 22. *Provide a rightmost derivation of the sentence* $\texttt{A} := \texttt{B} + \texttt{C} * \texttt{A}$

Exercise 23. *Construct the parse tree corresponding to the rightmost derivation and compare it to the parse tree obtained in Exercise 21.*

Clearly, one could also choose arbitrarily the nonterminal which allows the application of a production. Try it out, and construct the resulting parse trees. Usually, one is implicitly assuming that derivations are leftmost. Thus, we can more precisely say that a context-free grammar is ambiguous if there is a sentence in the language it generates that possesses at least two different leftmost derivations. Otherwise it is called ***unambiguous***. A language is said to be unambiguous if there is an unambiguous grammar for it.

Exercise 24. *Prove or disprove that every regular language is unambiguous.*

Another important problem in describing programming languages is to express that operators are *associative*. As you have learned in mathematics, addition and multiplication are associative. Is this also true for computer arithmetic? As far as integer addition and multiplication are concerned, they are associative. But floating point computer arithmetic is not always associative. So, in general correct associativity is essential.

For example, look at the expression

$$\texttt{A} := \texttt{B} + \texttt{C} + \texttt{A}$$

Then it should not matter whether or not the expression is evaluated in left order (that is $(\texttt{B} + \texttt{C}) + \texttt{A}$) or in right order (i.e, $\texttt{B} + (\texttt{C} + \texttt{A})$).

Note that some programming languages define in what order expressions are evaluated. In particular, in most programming languages that provide it, the exponentiation operator is right associative. The formal tool to describe right (or left) associativity is right (or left) recursion in the corresponding production rules. If a BNF rule

has its left hand side also appearing at the beginning of its right hand side, then the rule is said to be ***left recursive***. Analogously, if a BNF rule has its left hand side also appearing at the right end of its right hand side, then it is ***right recursive***.

The following grammars exemplifies of how to use right recursiveness to express right associativity of the exponentiation operator.

```
<factor>  →  <exp> * *<factor>
          |  <exp>
  <exp>   →  <expr>
          |  <id>
```

Finally, we should have a look at the **if-then-else** statement that is present in many programming languages. Is there an unambiguous grammar for it? This is indeed the case, but we leave it as an exercise.

7.3. Chomsky Normal Form

In a context-free grammar, there is no *a priori* bound on the size of a right-hand side of a production. This may complicate many proofs. Fortunately, there is a normal form for context-free grammars bounding the right-hand side to be of length at most 2. Knowing and applying this considerably simplifies many proofs.

First, we define the notion of a separated grammar.

Definition 22 (Separated Grammar). *A grammar* $\mathcal{G} = [\mathsf{T}, \mathsf{N}, \sigma, \mathsf{P}]$ *is called* ***separated*** *if for all* $(\alpha \to \beta) \in \mathsf{P}$ *we either have* $\alpha, \beta \in \mathsf{N}^*$ *or* $\alpha \in \mathsf{N}$ *and* $\beta \in \mathsf{T}$.

Theorem 7.1. *For every context-free grammar* $\mathcal{G} = [\mathsf{T}, \mathsf{N}, \sigma, \mathsf{P}]$ *there exists an equivalent separated context-free grammar* $\mathcal{G}' = [\mathsf{T}, \mathsf{N}', \sigma, \mathsf{P}']$.

Proof. First, we introduce for every $\mathsf{t} \in \mathsf{T}$ a new nonterminal symbol h_t, where by new we mean that $\mathsf{h}_\mathsf{t} \notin \mathsf{N}$ for all $\mathsf{t} \in \mathsf{T}$. Furthermore, we set $\mathsf{N}' = \{\mathsf{h}_\mathsf{t} \mid \mathsf{t} \in \mathsf{T}\} \cup \mathsf{N}$.

Next, for $(\alpha \to \beta) \in \mathsf{P}$, we denote the production obtained by replacing every terminal symbol t in β by h_t by $(\alpha \to \beta)[\mathsf{t}/\!\!/\mathsf{h}_\mathsf{t}]$. The production set P' is the defined as follows.

$$\mathsf{P}' = \{(\alpha \to \beta)[\mathsf{t}/\!\!/\mathsf{h}_\mathsf{t}] \mid (\alpha \to \beta) \in \mathsf{P}\} \cup \{\mathsf{h}_\mathsf{t} \to \mathsf{t} \mid \mathsf{t} \in \mathsf{T}\} .$$

By construction, we directly see that $\mathcal{G}'$ is separated. Moreover, the construction ensures that $\mathcal{G}'$ is context-free, too. Thus, it remains to show that $\mathsf{L}(\mathcal{G}) = \mathsf{L}(\mathcal{G}')$.

Claim 1. $\mathsf{L}(\mathcal{G}) \subseteq \mathsf{L}(\mathcal{G}')$.

Let $\mathsf{s} \in \mathsf{L}(\mathcal{G})$. Then there exists a derivation

$$\sigma \overset{1}{\Rightarrow} w_1 \overset{1}{\Rightarrow} w_2 \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w_n \overset{1}{\Rightarrow} \mathsf{s} ,$$

where $w_1, \ldots, w_n \in (N \cup T)^+ \setminus T^*$, and $s \in T^*$. Let P_i be the production used to generate w_i, $i = 1, \ldots, n$.

Then we can generate s by using productions from P' as follows. Let $s = s_1 \cdots s_m$, where $s_j \in T$ for all $j = 1, \ldots, m$. Instead of applying P_i we use $P_i[t/\!\!/h_t]$ from P' and obtain

$$\sigma \overset{1}{\Rightarrow} w_1' \overset{1}{\Rightarrow} w_2' \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w_n' \overset{1}{\Rightarrow} h_{s_1} \cdots h_{s_m} ,$$

where now $w_i' \in (N')^+$ for all $i = 1, \ldots, n$.

Thus, in order to obtain s it now suffices to apply the productions $h_{s_j} \to s_j$ for $j = 1, \ldots m$. This proves Claim 1.

Claim 2. $L(G') \subseteq L(G)$.

This claim can be proved analogously by inverting the construction used in showing Claim 1. ∎

Now, we are ready to define the normal form for context-free grammars announced above.

Definition 23 (Chomsky Normal Form). *A grammar $G = [T, N, \sigma, P]$ is said to be in **Chomsky normal form** if all productions of P have the form $h \to h_1 h_2$, where $h, h_1, h_2 \in N$, or $h \to x$, where $h \in N$ and $x \in T$.*

The latter definition directly allows the following corollary.

Corollary 7.2. *Let $G = [T, N, \sigma, P]$ be a grammar in Chomsky normal form. Then we have*

(1) *G is context-free,*

(2) *G is λ-free,*

(3) *G is separated.*

Theorem 7.3. *For every context-free grammar $G = [T, N, \sigma, P]$ such that $\lambda \notin L(G)$ there exists an equivalent grammar G' that is in Chomsky normal form.*

Proof. Let $G = [T, N, \sigma, P]$ be given. Without loss of generality, we may assume that G is reduced. First, we eliminate all productions of the form $h \to h'$. This is done as follows.

We set

$$W_0(h) = \{h\} \text{ for every } h \in N$$

and for each $i \geqslant 0$ we define

$$W_{i+1}(h) = W_i \cup \{\tilde{h} \mid \tilde{h} \in N \text{ and } (\hat{h} \to \tilde{h}) \in P \text{ for some } \hat{h} \in W_i(h)\} .$$

Then, the following facts are obvious:

(1) $W_i(h) \subseteq W_{i+1}(h)$ for all $i \geqslant 0$,

(2) If $W_i(h) = W_{i+1}(h)$ then $W_i(h) = W_{i+m}(h)$ for all $m \in \mathbb{N}$,

(3) $W_n(h) = W_{n+1}(h)$ for $n = \mathrm{card}(N)$,

(4) $W_n(h) = \{B \mid B \in N \text{ and } h \stackrel{*}{\Rightarrow} B\}$.

Now, we define

$$P_1 = \{h \to \gamma \mid h \in N \text{ and } \gamma \notin N \text{ and } (B \to \gamma) \in P \text{ for some } B \in W_n(h)\} .$$

Let $G_1 = [T, N, \sigma, P_1]$, then by construction P_1 does not contain any production of the form $h \to h'$. These productions have been replaced by $h \to \gamma$. That is, if we had $h \stackrel{*}{\Rightarrow} B$ by using the productions from P and $B \to \gamma$, then we now have the production $h \to \gamma$ in P_1. Also note that P_1 contains all original productions $(h \to \gamma) \in P$, where $\gamma \notin N$ by the definition of W_0. We leave it as an exercise to formally verify that $L(G_1) = L(G)$.

Next, from G_1 we construct an equivalent separated grammar G_2 by using the algorithm given in the proof of Theorem 7.1. Now, the only productions in P_2 that still need modification are of the form

$$h \to h_1 h_2 \cdots h_n \ , \text{ where } n \geqslant 3 \ .$$

We replace any such production by the following productions

$$\begin{aligned} h &\to h_1 h_{h_2 \cdots h_n} \\ h_{h_2 \cdots h_n} &\to h_2 h_{h_3 \cdots h_n} \\ &\vdots \\ h_{h_{n-1} h_n} &\to h_{n-1} h_n \end{aligned}$$

Hence, the resulting grammar G' is in Chomsky normal form and by construction equivalent to G. We omit the details. ∎

Exercise 25. *Let $G = [\{a, b, c\}, \{\sigma, h\}, \sigma, P]$, where the set P of productions is $P = \{\sigma \to hbh,\ h \to hah,\ h \to ca\}$. Construct a grammar G' which is in Chomsky normal form and which is equivalent to grammar G.*

Exercise 26. *Extend the notion of Chomsky normal form to context-free languages containing λ.*

We finish this lecture by pointing to an important result which can be proved by using the Chomsky normal form. This result is usually referred to as Pumping Lemma for context-free languages or Lemma of Bar-Hillel or $qrsuv$-Theorem. You should compare this theorem to the Pumping Lemma for regular languages (see Lemma 4.4). Please note that Theorem 7.4 provides a necessary condition for a language to be

context-free. It is not sufficient. So its main importance lies in the fact that one can often use it to show that a language is *not* context-free.

Theorem 7.4. *For every context-free language* L *there are numbers* k, ℓ *such that for every* $w \in L$ *with* $|w| > k$ *there strings* q, r, s, u, v *such that*

(1) $w = qrsuv$,

(2) $|rsu| \leqslant \ell$,

(3) $ru \neq \lambda$, *and*

(4) $qr^i su^i v \in L$ *for all* $i \in \mathbb{N}^+$.

Due to lack of time we do not prove this theorem here but just provide a hint to understand why it holds. By Theorem 7.3 we can assume that the grammar G for L is in Chomsky normal form. Then all parse trees are binary trees. Therefore, if we consider any string $w \in L$ such that $|w| > 2^{|N|+2}$ then any of its parse trees must have depth at least $|N| + 2$. Consequently, there must exist a path from the root σ to a leaf containing some nonterminal h at least twice. Looking then at the corresponding subtrees it is not too difficult to show the Theorem.

Finally, you should try to solve the following exercises.

Exercise 27. *Prove that* $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ *is not context-free.*

Exercise 28. *Design an algorithm that, on input any context-free grammar* G*, decides whether or not* $L(G) = \emptyset$.

References

[1] M.A. Harrison, *Introduction to Formal Language Theory*, Addison–Wesley Publishing Company, Reading Massachusetts, 1978.

[2] J.E. Hopcroft, R. Motwani and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation – Second Edition*, Addison–Wesley Publishing Company, Reading Massachusetts, 2001.

Lecture 8: $\mathcal{CF}$ and Homomorphisms

We continue with further useful properties and characterizations of context-free languages. First, we look at substitutions.

8.1. Substitutions and Homomorphisms

Since we aim to prove some theorems, some formal definitions and notations are needed. Recall that we write $\wp(X)$ to denote the power set of X.

Definition 24. *Let Σ and Δ be any two finite alphabets. A mapping $\tau\colon \Sigma \longrightarrow \wp(\Delta^*)$ is said to be a **substitution**. We extend τ to be a mapping $\tau\colon \Sigma^* \longrightarrow \wp(\Delta^*)$ (i.e., to strings) by defining*

(1) $\tau(\lambda) = \lambda$,

(2) $\tau(wx) = \tau(w)\tau(x)$ *for all $w \in \Sigma^*$ and $x \in \Sigma$.*

The mapping τ is generalized to languages $L \subseteq \Sigma^*$ by setting

$$\tau(L) = \bigcup_{w \in L} \tau(w) \ .$$

So, a substitution maps every symbol of Σ to a language over Δ. The language a symbol is mapped to can be finite or infinite.

Example 11. Let $\Sigma = \{0, 1\}$ and let $\Delta = \{\mathfrak{a}, \mathfrak{b}\}$. Then, the mapping τ defined by $\tau(\lambda) = \lambda$, $\tau(0) = \{\mathfrak{a}\}$ and $\tau(1) = \{\mathfrak{b}\}^*$ is a substitution.

Let us calculate $\tau(010)$. By definition,

$$\tau(010) = \tau(01)\tau(0) = \tau(0)\tau(1)\tau(0) = \{\mathfrak{a}\}\{\mathfrak{b}\}^*\{\mathfrak{a}\} = \underline{\mathfrak{a}}\langle\underline{\mathfrak{b}}\rangle\underline{\mathfrak{a}} \ ,$$

where the latter equality is by the definition of regular expressions.

Next, we want define what is meant by closure of a language family $\mathcal{L}$ under substitution. Here special care is necessary. At first glance, we may be tempted to require that for every substitution τ the condition $\tau(L) \in \mathcal{L}$ has to be satisfied. But this is a *too strong demand*. For seeing this, consider $\Sigma = \{0, 1\}$, $\Delta = \{\mathfrak{a}, \mathfrak{b}\}$ and $\mathcal{L} = \mathcal{REG}$. Furthermore, suppose that $\tau(0) = L$, where L is any recursively enumerable but non-recursive language over Δ. Then we obviously have $\tau(\{0\}) = L$, too. Consequently, $\tau(\{0\}) \notin \mathcal{REG}$. On the other hand, $\{0\} \in \mathcal{REG}$, and thus we would conclude that $\mathcal{REG}$ is not closed under substitution. Also, the same argument would prove that $\mathcal{CF}$ is not closed under substitution.

The point to be made here is that we have to restrict the set of allowed substitution to those ones that map the elements of Σ to languages belonging to $\mathcal{L}$. Therefore, we arrive at the following definition.

Definition 25. *Let Σ be any alphabet, and let $\mathcal{L}$ be any language family over Σ. We say that $\mathcal{L}$ is **closed under substitutions** if for every substitution $\tau\colon \Sigma \longrightarrow \mathcal{L}$ and every $L \in \mathcal{L}$ we have $\tau(L) \in \mathcal{L}$.*

A very interesting special case of substitution is the homomorphism.

Definition 26. *Let Σ and Δ be any two finite alphabets. A mapping $\varphi\colon \Sigma^* \longrightarrow \Delta^*$ is said to be a **homomorphism** if*

$$\varphi(vw) = \varphi(v)\varphi(w) \text{ for all } v, w \in \Sigma^* \ .$$

*Furthermore, φ is said to be a **λ-free homomorphism**, if additionally*

$$h(w) = \lambda \text{ implies } w = \lambda \text{ for all } w \in \Sigma^* \ .$$

*Moreover, if $\varphi\colon \Sigma^* \longrightarrow \Delta^*$ is a homomorphism then we define the **inverse of the homomorphism** φ to be the mapping $\varphi^{-1}\colon \Delta^* \longrightarrow \wp(\Sigma^*)$ by setting for each $s \in \Delta^*$*

$$\varphi^{-1}(s) = \{w \mid w \in \Sigma^* \text{ and } \varphi(w) = s\} \ .$$

So, a homomorphism is a substitution that maps every symbols of Σ to a *singleton* set. Clearly, by the definition of homomorphism, it already suffices to declare the mapping φ for the symbols in Σ. Note that, when dealing with homomorphisms we usually identify the language containing exactly one string by the string itself, i.e., instead of $\{s\}$ we shortly write s.

Example 12. Let $\Sigma = \{0, 1\}$ and let $\Delta = \{a, b\}$. Then, the mapping $\varphi\colon \Sigma^* \longrightarrow \Delta^*$ defined by $\varphi(0) = ab$ and $\varphi(1) = \lambda$ is a homomorphisms but not a λ-free homomorphism. Applying φ to 1100 yields $\varphi(1100) = \varphi(1)\varphi(1)\varphi(0)\varphi(0) = \lambda\lambda ab ab = abab$ and to the language $\underline{1}\langle\underline{0}\rangle\underline{1}$ gives $\varphi(\underline{1}\langle\underline{0}\rangle\underline{1}) = \langle\underline{ab}\rangle$.

For seeing the importance of the notions just introduced consider the language $L = \{a^n b^n \mid n \in \mathbb{N}\}$. This language is context-free as we have shown (cf. Theorem 6.1). Thus, we intuitively know that $\{0^n 1^n \mid n \in \mathbb{N}\}$ is also context-free, because we could go through the grammar and replace all occurrences of a by 0 and all occurrences of b by 1. This observation would suggest that if we replace all occurrences of a and b by strings v and w, respectively, we also get a context-free language. However, it is much less intuitive that we also obtain a context-free language if all occurrences of a and b are replaced by context-free sets of strings V and W, respectively. Nevertheless, we just aim to prove this *closure* property. For the sake of representation, in the following we always assume two finite alphabets Σ and Δ as in Definition 24.

Theorem 8.1. *$\mathcal{CF}$ is closed under substitutions.*

Proof. Let $L \in \mathcal{CF}$ be arbitrarily fixed and let τ be a substitution such that $\tau(a)$ is a context-free language for all $a \in \Sigma$. We have to show that $\tau(L)$ is context-free. We shall do this by providing a context-free grammar $\overline{G} = [\overline{T}, \overline{N}, \overline{\sigma}, \overline{P}]$ such that $L(\overline{G}) = \tau(L)$.

Since $L \in \mathcal{CF}$, there exists a context-free grammar $\mathcal{G} = [\Sigma, N, \sigma, P]$ in Chomsky normal form such that $L = L(\mathcal{G})$. Next, let $\Sigma = \{a_1, \ldots, a_n\}$ and consider $\tau(a)$ for all $a \in \Sigma$. By assumption, $\tau(a) \in \mathcal{CF}$ for all $a \in \Sigma$. Thus, there are context-free grammars $\mathcal{G}_a = [T_a, N_a, \sigma_a, P_a]$ such that $\tau(a) = L(\mathcal{G}_a)$ for all $a \in \Sigma$. Without loss of generality, we can assume the sets $N_{a_1}, \ldots, N_{a_n}$ to be pairwise disjoint.

At this point we need an idea how to proceed. It is not too difficult to get this idea, if one looks at possible derivations in $\mathcal{G}$. Suppose we have a derivation

$$\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x_1 x_2 \cdots x_m \ ,$$

where all $x_i \in \Sigma$ for $i = 1, \ldots, m$. Then, since $\mathcal{G}$ is in Chomsky normal form, we can conclude that there must be productions $(h_{x_i} \ \rightarrow \ x_i) \in P$, $i = 1, \ldots, m$, and hence, we easily achieve the following.

$$\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} h_{x_1} h_{x_2} \cdots h_{x_m} \overset{m}{\underset{\mathcal{G}}{\Longrightarrow}} x_1 x_2 \cdots x_m \ , \tag{8.1}$$

where all $h_{x_i} \in N$. Taking into account that the image $\tau(x_1 \cdots x_m)$ is obtained by calculating

$$\tau(x_1)\tau(x_2) \cdots \tau(x_m) \ ,$$

we see that for every string $w_1 w_2 \cdots w_m$ in this image there must be a derivation

$$\sigma_{x_i} \overset{*}{\underset{\mathcal{G}_{x_i}}{\Longrightarrow}} w_i \quad i = 1, \ldots, m \ .$$

This directly yields the idea for constructing $\overline{\mathcal{G}}$. That is, we aim to cut the derivation in (8.1) when having obtained $h_{x_1} h_{x_2} \cdots h_{x_m}$. Then, instead of deriving $x_1 x_2 \cdots x_m$, all we need is to generate $\sigma_{x_1} \cdots \sigma_{x_m}$, and thus, we have to replace the productions $(h_{x_i} \ \rightarrow \ x_i) \in P$ by $(h_{x_i} \ \rightarrow \ \sigma_{x_i}) \in \overline{P}$, $i = 1, \ldots, m$.

Formalizing this idea yields the following definition.

$$\begin{aligned}
\overline{T} &= \bigcup_{a \in \Sigma} T_a \\
\overline{N} &= N \cup \left(\bigcup_{a \in \Sigma} N_a \right) \\
\overline{\sigma} &= \sigma \\
\overline{P} &= \left(\bigcup_{a \in \Sigma} P_a \right) \cup P[a /\!\!/ \sigma_a] \ .
\end{aligned}$$

We set $\overline{\mathcal{G}} = [\overline{T}, \overline{N}, \overline{\sigma}, \overline{P}]$. Note that $P[a /\!\!/ \sigma_a]$ is the set of all productions from P where those productions containing $a \in \Sigma$ on the right hand side, i.e., $h_a \ \rightarrow \ a$, are replaced by $h_a \ \rightarrow \ \sigma_a$.

It remains to show that $\tau(L) = L(\overline{\mathcal{G}})$.

Claim 1. $\tau(L) \subseteq L(\overline{\mathcal{G}})$.

This can be seen as follows. If $\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x_1 \cdots x_m$, where $x_i \in \Sigma$ for all $i = 1 \ldots, m$ and if $\sigma_{x_i} \overset{*}{\underset{\mathcal{G}_{x_i}}{\Longrightarrow}} w_i$, where $w_i \in T_{x_i}$, $i = 1 \ldots, m$, then we can also derive $x_1 \cdots x_m$ in the following way:

$$\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} h_{x_1} \cdots h_{x_m} \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x_1 \cdots x_m \ ,$$

where all $h_{x_i} \in N$. By construction, we can thus generate

$$\sigma \overset{*}{\underset{\overline{\mathcal{G}}}{\Longrightarrow}} h_{x_1} \cdots h_{x_m} \overset{*}{\underset{\overline{\mathcal{G}}}{\Longrightarrow}} \sigma_{x_1} \cdots \sigma_{x_m} \overset{*}{\underset{\overline{\mathcal{G}}}{\Longrightarrow}} w_1 \cdots w_m \ .$$

Hence, Claim 1 follows.

Claim 2. $L(\overline{\mathcal{G}}) \subseteq \tau(L)$.

Now, we start from

$$\sigma \overset{*}{\Rightarrow} w \ , \text{ where } w \in \overline{T}^* \ .$$

If $w = \lambda$, then also $\sigma \to \lambda$ in P, and we are done. Otherwise, the construction of $\overline{\mathcal{G}}$ ensures that the derivation of w must look as follows.

$$\sigma \overset{*}{\underset{\overline{\mathcal{G}}}{\Longrightarrow}} \sigma_{x_1} \cdots \sigma_{x_m} \overset{*}{\underset{\overline{\mathcal{G}}}{\Longrightarrow}} w \ .$$

Furthermore, by our construction we then know that $\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x_1 \cdots x_m$ as we have shown in (8.1).

Additionally, there are strings $w_1, \ldots, w_m \in \overline{T}^*$ such that

$$w = w_1 \cdots w_m$$

and $\sigma_{x_i} \overset{*}{\underset{\mathcal{G}_{x_i}}{\Longrightarrow}} w_i$ for all $i = 1, \ldots, m$. Consequently, $w_i \in \tau(x_i)$. Therefore, $w \in \tau(L)$ and we are done.

Putting it all together, we see that $\tau(L) = L(\overline{\mathcal{G}})$. ∎

Please note that we have skipped some formal parts within the above proof to make it easier to read and to understand. However, you should be aware of this omission. Since we shall also omit such tedious formal parts in future proofs, this is a good place

to tell you what you should do at home when reading these course notes until you have made your skills perfect.

The omitted part is formally verified as follows. First, we define a mapping

$$\zeta\colon (\Sigma \cup \mathsf{N}) \longrightarrow \mathsf{N} \cup \left(\bigcup_{a \in \Sigma} \{\sigma_a\} \right)$$

by setting

$$\zeta(a) = \begin{cases} a \ , & \text{if } a \in \mathsf{N}, \\ \sigma_a, & \text{if } a \in \Sigma \ . \end{cases}$$

We extend ζ to strings by defining

$$\begin{aligned} \zeta(\lambda) &= \lambda \ , \text{ and} \\ \zeta(x) &= \zeta(x_1)\zeta(x_2)\cdots\zeta(x_m) \text{ for all } x \in (\Sigma \cup \mathsf{N})^* \ , \ x = x_1 \cdots x_m \ . \end{aligned}$$

Furthermore, we define a grammar $\mathcal{G}' = [\mathsf{T}', \mathsf{N}', \sigma, \mathsf{P}']$, where

$$\begin{aligned} \mathsf{T}' &= \bigcup_{a \in \Sigma} \{\sigma_a\} \\ \mathsf{N}' &= \mathsf{N} \\ \mathsf{P}' &= \{A \ \to \ \zeta(\alpha) \mid \ (A \ \to \ \alpha) \in \mathsf{P}\} \ . \end{aligned}$$

Then we can prove the following.

Claim: *For all* $A \in \mathsf{N}$ *we have* $A \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} \alpha$ *iff* $A \overset{*}{\underset{\mathcal{G}'}{\Longrightarrow}} \zeta(\alpha)$.

Claim 1 is intuitively obvious, since ζ corresponds to a simple renaming of Σ. The formal proof is done by induction on the length of the derivation and left as an exercise.

In particular, we have

$$\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x_1 \cdots x_m \in \mathsf{L}(\mathcal{G}) \text{ if and only if } \sigma \overset{*}{\underset{\mathcal{G}'}{\Longrightarrow}} \sigma_{x_1} \cdots \sigma_{x_m} \ ,$$

where $x_i \in \Sigma$ for all $i = 1, \ldots, m$. This completes the formal verification.

Theorem 8.1 allows the following nice corollary.

Corollary 8.2. *$\mathcal{CF}$ is closed under homomorphisms.*

Proof. Since homomorphisms are a special type of substitution, it suffices to argue that every singleton subset is context-free. But this is obvious, because we have already shown that every finite language belongs to $\mathcal{REG}$ and that $\mathcal{REG} \subseteq \mathcal{CF}$. Thus, the corollary follows. ∎

Now, try it yourself.

Exercise 29. *Prove or disprove: $\mathcal{REG}$ is closed under substitutions.*

Exercise 30. *Prove or disprove: $\mathcal{REG}$ is closed under homomorphisms.*

Exercise 31. *Prove or disprove: $\mathcal{REG}$ is closed under inverse homomorphisms.*

The family of all languages generated by a grammar in the sense of Definition 6 is denoted by $\mathcal{L}_0$.

Exercise 32. *Figure out why the proof of Theorem 8.1 does not work for showing that $\mathcal{L}_0$ is closed under substitutions. Then provide the necessary modifications to show closure under substitution for the family $\mathcal{L}_0$.*

8.2. Homomorphic Characterization of $\mathcal{CF}$

When we started to study context-free languages, we emphasized that many programming languages use balanced brackets of different kinds. Therefore, we continue with a closer look at bracket languages. Such languages are called ***Dyck languages***[§].

In order to define Dyck languages, we need the following notations. Let $n \in \mathbb{N}^+$ and let

$$X_n = \{a_1, \overline{a}_1, a_2, \overline{a}_2, \ldots, a_n, \overline{a}_n\} .$$

We consider the set X_n as a set of different bracket symbols, where a_i is an opening bracket and $\overline{a}_i$ is the corresponding closing bracket. Thus, it is justified to speak of X_n as a set of n different bracket symbols.

Definition 27. *A language L is said to be a **Dyck language** with n bracket symbols if L is isomorphic to the language D_n generated by the following grammar $G_n = [X_n, \{\sigma\}, \sigma, P_n]$, where P_n is given by*

$$P_n = \{\sigma \to \lambda,\ \sigma \to \sigma\sigma,\ \sigma \to a_1\sigma\overline{a}_1,\ \ldots,\ \sigma \to a_n\sigma\overline{a}_n\} .$$

The importance of Dyck languages will become immediately transparent, since we are going to prove a beautiful characterization theorem for context-free languages by using them.

8.2.1. The Chomsky-Schützenberger Theorem

Theorem 8.3 (Chomsky-Schützenberger Theorem). *For every context-free language L there are $n \in \mathbb{N}^+$, a homomorphism h and a regular language R_L such that*

$$L = h(D_n \cap R_L) .$$

[§]Walter von Dyck (1856-1934) was a mathematician. He was a founder of combinatorial group theory.

Proof. Consider any arbitrarily fixed context-free language L. Without loss of generality we can assume that $\lambda \notin L$. Furthermore, let $\mathcal{G} = [T, N, \sigma, P]$ be a context-free grammar in Chomsky normal form such that $L = L(\mathcal{G})$. Let $T = \{x_1, \ldots, x_m\}$ and consider all productions in P. Since $\mathcal{G}$ is in Chomsky normal form, all productions have the form $h_i \to h_i' h_i''$ or $h_j \to x$. Let t be the number of all nonterminal productions, i.e., of all productions $h_i \to h_i' h_i''$. Note that for any two such productions it is well possible that some but not all nonterminal symbols coincide.

In all we have m terminal symbols and t nonterminal productions. Thus, we try the Dyck language D_{m+t} over

$$X_{m+t} = \{\overline{x}_1, \ldots, \overline{x}_m, \overline{x}_{m+1}, \ldots, \overline{x}_{m+t}, x_{m+1}, \ldots, x_{m+t}, x_1, \ldots, x_m\} .$$

Next, we consider the mapping $\chi_{m+t}: X_{m+t} \longrightarrow T^*$ defined as follows.

$$\chi_{m+t}(x_j) = \begin{cases} x_j, & \text{if } 1 \leqslant j \leqslant m, \\ \lambda , & \text{if } m+1 \leqslant j \leqslant m+t . \end{cases}$$

and $\chi_{m+t}(\overline{x}_j) = \lambda$ for all $j = 1, \ldots, m+t$. We leave it as an exercise to show that χ_{m+t} is a homomorphism.

Now we are ready to define the following grammar $\mathcal{G}_L = [X_{m+t}, N, \sigma, P_L]$, where

$$\begin{aligned} P_L &= \{h \to x_i \overline{x}_i \mid 1 \leqslant i \leqslant m \text{ and } (h \to x_i) \in P\} \\ &\cup \{h \to x_i \overline{x}_i \overline{x}_{m+j} h_j'' \mid 1 \leqslant i \leqslant m,\ (h \to x_i) \in P,\ 1 \leqslant j \leqslant t\} \\ &\cup \{h_j \to x_{m+j} h_j' \mid 1 \leqslant j \leqslant t\} \end{aligned}$$

Clearly, $\mathcal{G}_L$ is a regular grammar. We set $R_L = L(\mathcal{G}_L)$, and aim to prove that

$$L = \chi_{m+t}(D_{m+t} \cap R_L) .$$

This is done via the following claims and lemmata.

Claim 1. $L \subseteq \chi_{m+t}(D_{m+t} \cap R_L)$.

The proof of Claim 1 is mainly based on the following lemma.

Lemma 8.4. *Let $\mathcal{G}$ be the grammar for L fixed above, let $\mathcal{G}_L$ be the grammar for R_L and let $h \in N$. If*

$$h \underset{\mathcal{G}}{\overset{1}{\Longrightarrow}} w_1 \underset{\mathcal{G}}{\overset{1}{\Longrightarrow}} w_2 \underset{\mathcal{G}}{\overset{1}{\Longrightarrow}} \cdots \underset{\mathcal{G}}{\overset{1}{\Longrightarrow}} w_{n-1} \underset{\mathcal{G}}{\overset{1}{\Longrightarrow}} w_n \in T^*$$

then there exists a $q \in D_{m+t}$ such that $h \underset{\mathcal{G}_L}{\overset{*}{\Longrightarrow}} q$ and $\chi_{m+t}(q) = w_n$.

Proof. The lemma is shown by induction on the length n of the derivation. For the induction basis let $n = 1$. Thus, our assumption is that

$$h \overset{1}{\underset{\mathcal{G}}{\Longrightarrow}} w_1 \in T^* \ .$$

Since $\mathcal{G}$ is in Chomsky normal form, we can conclude that $(h \ \rightarrow \ w_1) \in P$. So, by the definition of Chomsky normal form, we must have $w_1 = x$ for some $x \in T$.

We have to show that there is a $q \in D_{m+t}$ such that $h \overset{*}{\underset{\mathcal{G}_L}{\Longrightarrow}} q$ and $\chi_{m+t}(q) = x$. By construction, the production $h \ \rightarrow \ x\overline{x}$ belongs to P_L (cf. the first set of the definition of P_L). Thus, we can simply set $q = x\overline{x}$. Now, the induction basis follows, since the definition of χ_{m+t} directly yields

$$\chi_{m+t}(q) = \chi_{m+t}(x\overline{x}) = \chi_{m+t}(x)\chi_{m+t}(\overline{x}) = x\lambda = x \ .$$

Assuming the induction hypothesis for $n \geqslant 1$, we are going to perform the induction step to $n + 1$. So, let

$$h \overset{1}{\underset{\mathcal{G}}{\Longrightarrow}} w_1 \overset{1}{\underset{\mathcal{G}}{\Longrightarrow}} \cdots \overset{1}{\underset{\mathcal{G}}{\Longrightarrow}} w_n \overset{1}{\underset{\mathcal{G}}{\Longrightarrow}} w_{n+1} \in T^*$$

be a derivation of length $n+1$. Because of $n \geqslant 1$, and since the derivation has length at least 2, we can conclude that the production used to derive w_1 must be of the form $h \ \rightarrow \ h'h''$, where $h, h', h'' \in N$. Therefore, there must be a j such that $1 \leqslant j \leqslant t$ and $h = h_j$ as well as $w_1 = h'_j h''_j$.

The latter observation implies that there must be v_1, v_2 such that $w_{n+1} = v_1 v_2$ and

$$h'_j \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} v_1 \quad \text{and} \quad h''_j \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} v_2 \ .$$

Since the length of the complete derivation is $n + 1$, both the generation of v_1 and of v_2 must have a length smaller than or equal to n.

Hence, we can apply the induction hypothesis. That is, there are strings q_1 and q_2 such that $q_1, \ q_2 \in D_{m+t}$ and $\chi_{m+t}(q_1) = v_1$ as well as $\chi_{m+t}(q_2) = v_2$. Furthermore, by the induction hypothesis we additionally know that

$$h'_j \overset{*}{\underset{\mathcal{G}_L}{\Longrightarrow}} q_1 \quad \text{and} \quad h''_j \overset{*}{\underset{\mathcal{G}_L}{\Longrightarrow}} q_2 \ .$$

Taking into account that $(h_j \ \rightarrow \ h'_j h''_j) \in P$ we know by construction that $h_j \ \rightarrow \ x_{m+j} h'_j$ is a production in P_L. Thus,

$$h = h_j \overset{1}{\underset{\mathcal{G}_L}{\Longrightarrow}} x_{m+j} h'_j \overset{*}{\underset{\mathcal{G}_L}{\Longrightarrow}} x_{m+j} q_1$$

is a regular derivation. Moreover, the last step of this derivation must look as follows:

$$x_{m+j} q_1' h_k \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} x_{m+j} q_1' x \overline{x} \ .$$

where $h_k \ \rightarrow \ x\overline{x}$ is the rule applied and where x is determined by the condition $q_1 = q_1' x \overline{x}$.

Now, we replace this step by using the production $h_k \ \rightarrow \ x\overline{x}\overline{x}_{m+j} h_j''$ which also belongs to P_L. Thus, we obtain

$$h = h_j \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} x_{m+j} h_j' \underset{\mathcal{G}_L}{\overset{*}{\Longrightarrow}} x_{m+j} q_1 \overline{x}_{m+j} h_j'' \underset{\mathcal{G}_L}{\overset{*}{\Longrightarrow}} x_{m+j} q_1 \overline{x}_{m+j} q_2 =: q \in D_{m+t} \ .$$

The containment in D_{m+t} is due to the correct usage of the brackets x_{m+j} and $\overline{x}_{m+j}$ around q_1 and the fact that $q_2 \in D_{m+t}$ as well as by the definition of the Dyck language. Finally, the definition of χ_{m+t} ensures that $\chi_{m+t}(x_{m+j} q_1 \overline{x}_{m+j} q_2) = \nu_1 \nu_2$. This proves Lemma 8.4.

Now, Claim 1 immediately follows for $h = \sigma$.

Claim 2. $L \supseteq \chi_{m+t}(D_{m+t} \cap R_L)$.

Again, the proof of the claim is mainly based on a lemma which we state next.

Lemma 8.5. *Let $\mathcal{G}$ be the grammar for L fixed above, let $\mathcal{G}_L$ be the grammar for R_L and let $h \in N$. If*

$$h \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} w_1 \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} \cdots \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} w_n \in D_{m+t}$$

then $h \underset{\mathcal{G}}{\overset{*}{\Longrightarrow}} \chi_{m+t}(w_n)$.

Proof. Again, the lemma is shown by induction on the length of the derivation. We perform the induction basis for $n = 1$. Consider

$$h \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} w_1 \in D_{m+t} \ .$$

Hence, we must conclude that $(h \ \rightarrow \ w_1) \in P_L$. So, there must exist $x_i \overline{x}_i$ such that $w_1 = x_i \overline{x}_i$, $1 \leqslant i \leqslant m$ and $(h \ \rightarrow \ x_i x_i) \in P_L$. By the definition of P_L we conclude that $(h \ \rightarrow \ x_i) \in P$. Hence

$$h \underset{\mathcal{G}}{\overset{1}{\Longrightarrow}} x_i = \chi_{m+t}(x_i \overline{x}_i) = \chi_{m+t}(w_1) \ .$$

This proves the induction basis.

Next, assume the induction hypothesis for all derivation of length less than or equal to n. We perform the induction step from n to $n+1$. Consider

$$h \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} w_1 \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} \cdots \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} w_n \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} w_{n+1} \in D_{m+t} \ .$$

The derivation $h \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} w_1$ must have been done by using a production $(h \to w_1) \in P_L$, where

$$w_1 \in \{x\overline{x}x_{m+j}h_j'',\ x_{m+j}h_j'\} \ .$$

But $w_1 \neq x\overline{x}x_{m+j}h_j''$, since $x\overline{x}x_{m+j}$ cannot be removed by any further derivation step. This would imply $w_{n+1} = x\overline{x}x_{m+j}r \notin D_{m+t}$. So, this case cannot happen.

Thus, the only remaining case is that $w_1 = x_{m+j}h_j'$. Therefore is a j with $1 \leqslant j \leqslant t$ such that $h = h_j$ and $w_1 = x_{m+j}h_j'$. This implies that there must be a derivation

$$h_j' \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} w_1' \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} w_2' \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} \cdots \underset{\mathcal{G}_L}{\overset{1}{\Longrightarrow}} w_{n+1}' \ ,$$

where $w_{n+1} = x_{m+j}w_{n+1}' \in D_{m+t}$. Therefore there are w' and w'' such that

$$w_{n+1} = x_{m+j}w'\overline{x}_{m+j}w'' \text{ and } w',\ w'' \in D_{m+t} \ .$$

Hence, there exists a k with $2 \leqslant k \leqslant m$ such that $w_k' = w'\overline{x}_{m+j}h_j''$. This means nothing else than having used in the kth derivation step a production of the form $h \to x\overline{x}\overline{x}_{m+j}h_j''$. Consequently, $(h \to x) \in P$ and $(h \to x\overline{x}) \in P_L$.

We thus replace the application of $h \to x\overline{x}\overline{x}_{m+j}h_j''$ by an application of $h \to x\overline{x}$ and obtain

$$h_j \underset{\mathcal{G}_L}{\overset{*}{\Longrightarrow}} w' \quad \text{and} \quad h_j' \underset{\mathcal{G}_L}{\overset{*}{\Longrightarrow}} w'' \ ,$$

where both derivations have a length less than or equal to n.

Applying the induction hypothesis yields

$$h_j' \underset{\mathcal{G}}{\overset{*}{\Longrightarrow}} \chi_{m+t}(w')$$

$$h_j'' \underset{\mathcal{G}}{\overset{*}{\Longrightarrow}} \chi_{m+t}(w'')$$

and thus

$$h \underset{\mathcal{G}}{\overset{1}{\Longrightarrow}} h_j'h_j'' \underset{\mathcal{G}}{\overset{*}{\Longrightarrow}} \chi_{m+t}(w'w'') = \chi_{m+t}(x_{m+j}w'\overline{x}_{m+j}w'')$$

$$= \chi_{m+t}(w_{n+1}) \ .$$

This proves Lemma 8.5. Finally, Claim 2 is a direct consequence of Lemma 8.5 for $h = \sigma$.

Claim 1 and Claim 2 together imply the theorem. ∎

Note that the Chomsky-Schützenberger Theorem has a nice counterpart which can be stated in terms of languages accepted by pushdown automata.

Lecture 9: Pushdown Automata

As already mentioned, the context-free languages also have a type of automaton that characterizes them. This automaton, called ***pushdown automaton***, is an extension of the nondeterministic finite automaton with λ-transitions. Here, by λ-transition we mean that the automaton is allowed to read the empty word λ on its input tape and to change its state accordingly. A pushdown automaton is essentially a nondeterministic finite automaton with λ-transitions with the addition of a stack. The stack can be *read*, *pushed*, and *popped* only at the top, just like the stack data structure you are already familiar with (cf. Figure 9.1).

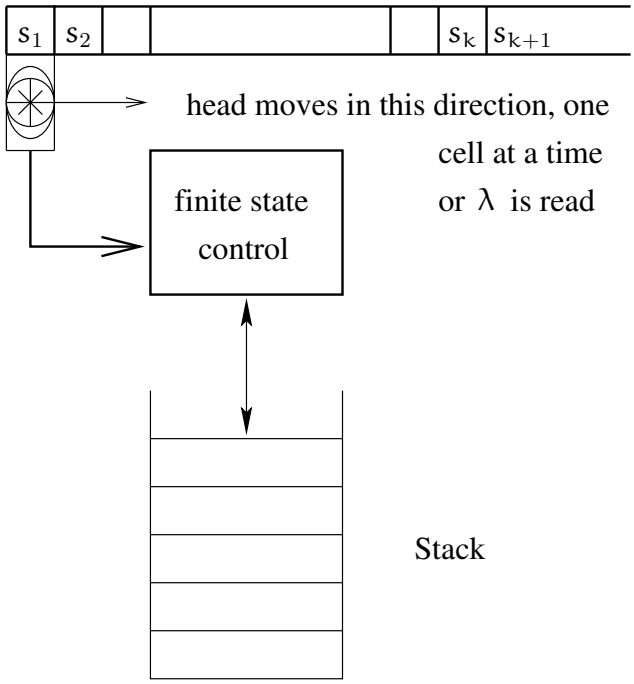

Figure 9.1: A pushdown automaton

Thus, informally the device shown in Figure 9.1 works as follows. A finite state control reads inputs, one symbol at a time or it reads λ instead of an input symbol. Additionally, it is allowed to observe the symbol at the top of the stack and to base its state transition on its current state, the input symbol (or λ), and the symbol at the top of its stack. If λ is read instead of the input symbol, then we also say that the pushdown automaton makes a *spontaneous* transition. In one transition, the pushdown automaton:

(1) Consumes from the input the symbol it reads. If λ is read then no input symbol is consumed.

(2) Goes to a new state which may or may not be the same state as its current state.

(3) Replaces the symbol at the top of the stack by any string. The string could be λ, which corresponds to a pop of the stack. It could be the same symbol that appeared at the top of the stack previously, i.e., no change is made to the stack. It could also replace the symbol on top of the stack by one other symbol. In this case, the pushdown automaton changes the top of the stack but does neither push or pop it.

Finally, the top stack symbol could be replaced by two or more symbols which has the effect of (possibly) changing the the top of stack symbol and then pushing one or more new symbols onto the stack.

Before presenting an example you should solve the following exercise.

Exercise 33. *Prove the language* $L = \{ww^T \mid w \in \{0,1\}^*\}$ *to be context-free.*

Example: We informally show $L = \{ww^T \mid w \in \{0,1\}^*\}$ to be acceptable by a pushdown automaton. Let $x \in \{0,1\}^*$ be given as input.

(1) Start in a state q_0 representing a "guess" that we have not yet seen the middle of x. While in state q_0, we read one symbol at a time and store the symbol read in the stack by pushing a copy of each input symbol onto the stack.

(2) At any time, we may guess that we have seen the middle (i.e., the end of w if $x = ww^T$ is an input string from L). At this time, w will be on the stack with the rightmost symbol of w at the top and the leftmost symbol of w at the bottom. We signify this choice by spontaneously changing the state to q_1 (i.e., we read λ instead of the next input symbol).

(3) Once in state q_1, we compare the input symbols with the symbols at the top of the stack. If the symbol read from input is equal to symbol at the top of the stack, we proceed in state q_1 and pop the stack. If they are different, we finish without accepting the input. That is, this branch of computation dies.

(4) If we reach the end of x and the stack is empty, then we accept x.

Clearly, if $x \in L$, then by guessing the middle of x rightly, we arrive at an accepting computation path. If $x \notin L$, then independently of what we are guessing, no computation path will lead to acceptance. Thus, the pushdown automaton described above is a nondeterministic acceptor for L.

Note that a pushdown automaton is allowed to change the stack as described above while performing a spontaneous transition. Before presenting the formal definition of pushdown automaton, this is a good place to think of ways to define the language accepted by a pushdown automaton. Looking at the example above, we see that the automaton has finished its computation with empty stack. Thus, it would be natural to define the language accepted by a pushdown automaton to be the set of all strings on which the pushdown automaton has a computation that ends with empty stack.

Second, we can adopt the method we have used for finite automata. That is, we choose a subset of the set of all states and declare each state in this subset to be an accepting state. If taking this approach, it would be natural to define the language accepted by a pushdown automaton to be the set of all strings for which there is a computation ending in an accepting state.

As we shall show below, both method are equivalent. Therefore, we have to define both modes of acceptance here.

We continue with a formal definition of a pushdown automaton.

Definition 28. $\mathcal{K} = [Q, \Sigma, \Gamma, \delta, q_0, k_0, F]$ *is said to be a* ***pushdown automaton*** *provided*

(1) Q *is a finite nonempty set* (*the set of* ***states***),

(2) Σ *is an alphabet* (*the so-called input alphabet*),

(3) Γ *is an alphabet* (*the so-called stack alphabet*),

(4) $\delta\colon Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \mapsto \wp_{\mathtt{fin}}(Q \times \Gamma^*)$, *the* ***transition relation***[¶],

(5) $q_0 \in Q$ *is the* ***initial state***,

(6) k_0 *is the so called* ***stack symbol***, *i.e.,* $k_0 \in \Gamma$ *and initially the stack contains exactly one* k_0 *and nothing else.*

(7) $F \subseteq Q$, *the set of* ***final states***.

In the following, unless otherwise stated, we use small letters from the beginning of the alphabet to denote input symbols, and small letters from the end of the alphabet to denote strings of input symbols. Furthermore, we use capital letters to denote stack symbols from Γ and small Greek letters to denote strings of stack symbols.

Next, consider

$$\delta(q, a, Z) = \{(q_1, \gamma_1), (q_2, \gamma_2), \ldots, (q_m, \gamma_m)\} ,$$

where $q, q_i \in Q$ for $i = 1, \ldots, m$, $a \in \Sigma$, $Z \in \Gamma$ and $\gamma_i \in \Gamma^*$ for $i = 1, \ldots, m$.

The interpretation is that the pushdown automaton $\mathcal{K}$ is in state q, reads a on its input tape and Z on the top of its stack. Then it can nondeterministically choose exactly one (q_i, γ_i), $i \in \{1, \ldots, m\}$ for the transition to be made. That is, it changes its internal state to q_i, moves the head on the input tape one position to the right provided $a \neq \lambda$ and replaces Z by γ_i. We make the convention that the rightmost symbol of γ_i is pushed first in the stack, then the second symbol (if any) from the right, and so on. Hence, the leftmost symbol of γ_i is the new symbol is then on the top of the stack. If $\gamma_i = \lambda$, then the interpretation is that Z has been removed from the stack.

[¶]Here we use $\wp_{\mathtt{fin}}(Q \times \Gamma^*)$ to denote the set of all finite subsets of $Q \times \Gamma^*$.

If $a = \lambda$, the interpretation is the same as above, except that the head on the input tape is *not* moved.

In order to formally deal with computations performed by a pushdown automaton we define *instantaneous descriptions*. An instantaneous description is a triple (q, w, γ), where $q \in Q$, $w \in \Sigma^*$ and $\gamma \in \Gamma^*$.

Let $\mathcal{K} = [Q, \Sigma, \Gamma, \delta, q_0, k_0, F]$ be a pushdown automaton. Then we write

$$(q, aw, Z\alpha) \xrightarrow[\mathcal{K}]{1} (p, w, \beta\alpha)$$

provided $(p, \beta) \in \delta(q, a, Z)$. Again note that a may be a symbol from Σ or $a = \lambda$.

By $\xrightarrow[\mathcal{K}]{*}$ we denote the reflexive transitive closure of $\xrightarrow[\mathcal{K}]{1}$.

Now we are ready to define the two modes of acceptance.

Definition 29. *Let $\mathcal{K} = [Q, \Sigma, \Gamma, \delta, q_0, k_0, F]$ be a pushdown automaton. We define the language* ***accepted by $\mathcal{K}$ via final state*** *to be the set*

$$L(\mathcal{K}) = \{w \mid (q_0, w, k_0) \xrightarrow[\mathcal{K}]{*} (p, \lambda, \gamma) \text{ for some } p \in F \text{ and a } \gamma \in \Gamma^*\} .$$

The language ***accepted by $\mathcal{K}$ via empty stack*** *is*

$$N(\mathcal{K}) = \{w \mid (q_0, w, k_0) \xrightarrow[\mathcal{K}]{*} (p, \lambda, \lambda) \text{ for some } p \in Q\} .$$

Since the sets of final states is irrelevant if acceptance via empty stack is considered, we always set $F = \emptyset$ in this case.

Exercise 34. *Provide a formal definition for a pushdown automaton that accepts the language $L = \{ww^T \mid w \in \{0, 1\}^*\}$.*

At this point it is only natural to ask what is the appropriate definition of determinism for pushdown automata. The answer is given by our next definition.

Definition 30. *A pushdown automaton $\mathcal{K} = [Q, \Sigma, \Gamma, \delta, q_0, k_0, F]$ is said to be* ***deterministic*** *if*

(1) *for every $q \in Q$ and $Z \in \Gamma$ we have $\delta(q, a, Z) = \emptyset$ for all $a \in \Sigma$ if $\delta(q, \lambda, Z) \neq \emptyset$, and*

(2) *for all $q \in Q$, $Z \in \Gamma$ and $a \in \Sigma \cup \{\lambda\}$ we have $\operatorname{card}(\delta(q, a, Z)) \leqslant 1$.*

In the latter definition, we had to include Condition (1) to avoid a choice between a normal transition and a spontaneous transition. On the other hand, Condition (2)

guarantees that there is no choice in any step. So, Condition (2) resembles the condition we had imposed when defining deterministic finite automata. The language accepted by a deterministic pushdown automaton is defined in the same way as for nondeterministic pushdown automata. That is, we again distinguish between acceptance via final state and empty stack, respectively.

As far as finite automata have been concerned, we could prove that the class of languages accepted by deterministic finite automata is the same as the class of languages accepted by nondeterministic finite automata. Note that an analogous result cannot be obtained for pushdown automata. For example, there is no deterministic pushdown automaton accepting the language $L = \{ww^T \mid w \in \{0,1\}^*\}$. We shall show this result later.

We continue by comparing the power of the two notions of acceptance. First, we show the following theorem.

Theorem 9.1. *Let* $L = L(\mathcal{K})$ *for a pushdown automaton* $\mathcal{K}$*. Then there exists a pushdown automaton* $\tilde{\mathcal{K}}$ *such that* $L = N(\tilde{\mathcal{K}})$*.*

Proof. Clearly, such a theorem is proved by providing a simulation, i.e., we want to modify $\mathcal{K}$ in a way such that the stack is emptied whenever $\mathcal{K}$ reaches a final state. In order to do so, we introduce a new state q_λ and a special stack symbol X_0 to avoid acceptance if $\mathcal{K}$ has emptied its stack without having reached a final state.

Formally, we proceed as follows. Let $\mathcal{K} = [Q, \Sigma, \Gamma, \delta, q_0, k_0, F]$ be any given pushdown automaton such that $L = L(\mathcal{K})$. We have to construct a pushdown automaton $\tilde{\mathcal{K}}$ such that $L = N(\tilde{\mathcal{K}})$. We set

$$\tilde{\mathcal{K}} = [Q \cup \{q_\lambda, \tilde{q}_0\}, \Sigma, \Gamma \cup \{X_0\}, \tilde{q}_0, X_0, \tilde{\delta}, \emptyset] \ ,$$

where $\tilde{\delta}$ is defined as follows.

$$\begin{aligned}
\tilde{\delta}(\tilde{q}_0, \lambda, X_0) &= \{(q_0, k_0 X_0)\} \\
\tilde{\delta}(q, a, Z) &= \delta(q, a, Z) \text{ for all } q \in Q \setminus F,\ a \in \Sigma \cup \{\lambda\}, \text{ and } Z \in \Gamma \\
\tilde{\delta}(q, a, Z) &= \delta(q, a, Z) \text{ for all } q \in F,\ a \in \Sigma \text{ and } Z \in \Gamma \\
\tilde{\delta}(q, \lambda, Z) &= \delta(q, \lambda, Z) \cup \{(q_\lambda, \lambda)\} \text{ for all } q \in F \text{ and } Z \in \Gamma \cup \{X_0\} \\
\tilde{\delta}(q_\lambda, \lambda, Z) &= \{(q_\lambda, \lambda)\} \text{ for all } Z \in \Gamma \cup \{X_0\} \ .
\end{aligned}$$

By construction, when starting $\tilde{\mathcal{K}}$, it is entering the initial instantaneous description of $\mathcal{K}$ but pushes additionally its own stack symbol X_0 into the stack. Then $\tilde{\mathcal{K}}$ simulates $\mathcal{K}$ until it reaches a final state. If $\mathcal{K}$ reaches a final state, then $\tilde{\mathcal{K}}$ can either continue to simulate $\mathcal{K}$ or it can change its state to q_λ. If $\tilde{\mathcal{K}}$ is in q_λ, it can empty the stack and thus accept the input.

Thus, formally we can continue as follows. Let $x \in L(\mathcal{K})$. Then, there is a computation such that

$$(q_0, x, k_0) \xrightarrow[\mathcal{K}]{*} (q, \lambda, \gamma) \text{ for a } q \in F \ .$$

Now, we consider $\tilde{\mathcal{K}}$ on input x. By its definition, $\tilde{\mathcal{K}}$ starts in state $\tilde{q}_0$ and with stack symbol X_0. Thus, by using the first spontaneous transition, we get

$$(\tilde{q}_0, x, X_0) \xrightarrow[\tilde{\mathcal{K}}]{1} (q_0, x, k_0 X_0) \ .$$

Next, $\tilde{\mathcal{K}}$ can simulate every step of $\mathcal{K}$'s work; hence we also have

$$(\tilde{q}_0, x, X_0) \xrightarrow[\tilde{\mathcal{K}}]{1} (q_0, x, k_0 X_0) \xrightarrow[\tilde{\mathcal{K}}]{*} (q, \lambda, \gamma X_0)$$

Finally, using the last two transitions in the definition of $\tilde{\delta}$ we obtain

$$(q_0, x, k_0 X_0) \xrightarrow[\tilde{\mathcal{K}}]{*} (q_\lambda, \lambda, \lambda) \ .$$

Therefore we can conclude that $x \in N(\tilde{\mathcal{K}})$. ∎

So, we have shown that, if a language is accepted by a pushdown automaton via final state then it can also be accepted by a pushdown automaton via empty stack. It is only natural to ask whether or not the converse is also true. The affirmative answer is given by our next theorem.

Theorem 9.2. *Let $L = N(\mathcal{K})$ for a pushdown automaton $\mathcal{K}$. Then there exists a pushdown automaton $\tilde{\mathcal{K}}$ such that $L = L(\tilde{\mathcal{K}})$.*

Proof. Again the proof is done by simulation. The pushdown automaton $\tilde{\mathcal{K}}$ will simulate $\mathcal{K}$ until it detects that $\mathcal{K}$ has emptied its stack. If this happens then $\tilde{\mathcal{K}}$ will enter a final state and stop.

The formal construction is as follows. Let $\mathcal{K} = [Q, \Sigma, \Gamma, \delta, q_0, k_0, \emptyset]$ be any given pushdown automaton such that $L = N(\mathcal{K})$. We have to construct a pushdown automaton $\tilde{\mathcal{K}}$ such that $L = L(\tilde{\mathcal{K}})$. We set

$$\tilde{\mathcal{K}} = \Big[Q \cup \{\tilde{q}_0, q_f\}, \Sigma, \Gamma \cup \{X_0\}, \tilde{\delta}, \tilde{q}_0, X_0, \{q_f\} \Big] \ ,$$

where $\tilde{\delta}$ is defined as follows.

$$\begin{aligned} \tilde{\delta}(\tilde{q}_0, \lambda, X_0) &= \{(q_0, k_0 X_0)\} \\ \tilde{\delta}(q, a, Z) &= \delta(q, a, Z) \text{ for all } q \in Q,\ a \in \Sigma \cup \{\lambda\} \text{ and } Z \in \Gamma \\ \tilde{\delta}(q, \lambda, X_0) &= \{(q_f, \lambda)\} \text{ for all } q \in Q \ . \end{aligned}$$

The first line in the definition of $\tilde{\delta}$ ensures that $\tilde{\mathcal{K}}$ can start the simulation of $\mathcal{K}$. Note, however, that $\tilde{\mathcal{K}}$ is putting its own stack symbol *below* the stack symbol of $\mathcal{K}$. The second line in the definition of $\tilde{\delta}$ allows that $\tilde{\mathcal{K}}$ can simulate all steps of $\mathcal{K}$. If $\mathcal{K}$ empties its stack, then $\tilde{\mathcal{K}}$ also removes all symbols from its stack except its own stack symbol X_0 while performing the simulation. Finally, the last line in the definition of $\tilde{\delta}$ guarantees that $\tilde{\mathcal{K}}$ can perform a spontaneous transition into its final state q_f. Thus, $\tilde{\mathcal{K}}$ then also accepts the input string. The formal verification of $L(\tilde{\mathcal{K}}) = N(\mathcal{K})$ is left as exercise. ∎

Exercise 35. *Complete the proof of Theorem 9.2 by showing that* $L(\tilde{\mathcal{K}}) = N(\mathcal{K})$.

9.1. Pushdown Automata and Context-Free Languages

So far, we have only dealt with pushdown automata and their acceptance behavior. It remains to clarify what languages are accepted by pushdown automata. This is done by the following theorems. Recall that a derivation is said to be a leftmost derivation if at each step in derivation a production is applied to the leftmost nonterminal.

Theorem 9.3. *Let* $\mathcal{K} = [Q, \Sigma, \Gamma, \delta, q_0, k_0, \emptyset]$ *be any pushdown automaton and let* $L = N(\mathcal{K})$. *Then* L *is context-free.*

Proof. Let $\mathcal{K} = [Q, \Sigma, \Gamma, \delta, q_0, k_0, \emptyset]$ be any pushdown automaton. For proving that L defined as $L = N(\mathcal{K})$ is context-free, we have to construct a context-free grammar $\mathcal{G}$ such that $L = L(\mathcal{G})$. We set $\mathcal{G} = [\Sigma, N, \sigma, P]$, where N is defined as follows. The elements of N are denoted by $[q, A, p]$, where $p, q \in Q$ and $A \in \Gamma$. Additionally, N contains the symbol σ. Next, we have to define the set of productions. P contains the following rules.

(1) $\sigma \to [q_0, k_0, q]$ for every $q \in Q$,

(2) $[q, A, q_{m+1}] \to a[q_1, B_1, q_2][q_2, B_2, q_3] \cdots [q_m, B_m, q_{m+1}]$ for $q_1, \ldots, q_{m+1} \in Q$ and $A, B_1, \ldots, B_m \in \Gamma$ such that $(q_1, B_1 B_2 \cdots B_m) \in \delta(q, a, A)$ provided $m > 0$.

If $m = 0$ then the production is $[q, A, q_1] \to a$.

To understand the proof it helps to know that the nonterminals and productions of $\mathcal{G}$ have been defined in a way such that a leftmost derivation in $\mathcal{G}$ of a string x is a simulation of the pushdown automaton $\mathcal{K}$ when fed the input x. In particular, the nonterminals that appear in any step of a leftmost derivation in $\mathcal{G}$ correspond to the symbols on the stack of $\mathcal{K}$ at a time when $\mathcal{K}$ has seen as much of the input as the grammar has already generated. In other words, our intention is that $[q, A, p]$ derives x if and only if x causes $\mathcal{K}$ to erase an A from its stack by some sequence of moves beginning in state q and ending in state p.

For showing that $L(\mathcal{G}) = N(\mathcal{K})$ we prove inductively

$$[q, A, p] \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x \quad \text{if and only if} \quad (q, x, A) \overset{*}{\underset{\mathcal{K}}{\longrightarrow}} (p, \lambda, \lambda) \; . \tag{9.1}$$

First, we show by induction on i that if

$$(q, x, A) \overset{i}{\underset{\mathcal{K}}{\longrightarrow}} (p, \lambda, \lambda) \text{ then } [q, A, p] \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x \; .$$

For the induction basis let $i = 1$. In order to have $(q, x, A) \xrightarrow[\mathcal{K}]{1} (p, \lambda, \lambda)$ it must hold that $(p, \lambda) \in \delta(q, x, A)$. Consequently, either we have $x = \lambda$ or $x \in \Sigma$. In both cases, by construction of P we know that $([q, A, p] \rightarrow x) \in P$. Hence, $[q, A, p] \overset{1}{\underset{\mathcal{G}}{\Longrightarrow}} x$. This proves the induction basis.

Now suppose $i > 0$. Let $x = ay$ and

$$(q, ay, p) \xrightarrow[\mathcal{K}]{1} (q_1, y, B_1 B_2 \cdots B_n) \xrightarrow[\mathcal{K}]{i-1} (p, \lambda, \lambda) \ .$$

The string y can be written as $y = y_1 y_2 \cdots y_n$, where y_j has the effect of popping B_j from the stack, possibly after a long sequence of moves. That is, let y_1 be the prefix of y at the end of which the stack first becomes as short as $n - 1$ symbols. Let y_2 be the symbols of y following y_1 such that at the end of y_2 the stack first becomes as short as $n - 2$ symbols, and so on. This arrangement is displayed in Figure 9.2.

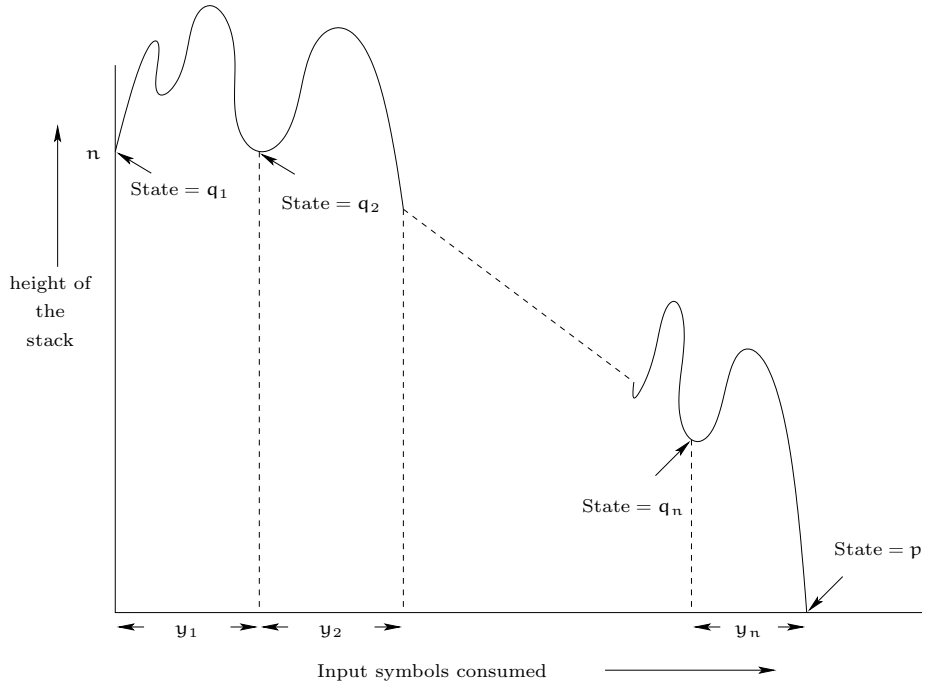

Figure 9.2: Hight of stack as a function of input symbols consumed

Note that B_1 does not need to be the nth stack symbol from the bottom during the entire time y_1 is being read by $\mathcal{K}$, since B_1 may be changed if it is at the top of the stack and is replaced by one or more symbols. However, none of $B_2 B_3 \cdots B_n$ are ever on top while y_1 is being read. Thus, none of $B_2 B_3 \cdots B_n$ can be changed or influence the computation while y_1 is processed. In general, B_j remains on the stack unchanged while $y_1 \cdots y_{j-1}$ is read.

There exists states $q_2, q_3, \ldots, q_{n+1}$, where $q_{n+1} = p$ such that

$$(q_j, y_j, B_j) \xrightarrow[\mathcal{K}]{*} (q_{j+1}, \lambda, \lambda)$$

by fewer than i moves. Note that q_j is the state entered when the stack first becomes as short as $n - j + 1$. Thus, we can apply the induction hypothesis and obtain

$$[q_j, B_j, q_{j+1}] \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} y_j \text{ for } 1 \leqslant j \leqslant n \ .$$

Recalling the original move

$$(q, ay, p) \xrightarrow[\mathcal{K}]{1} (q_1, y, B_1 B_2 \cdots B_n)$$

we know that

$$[q, A, p] \ \Rightarrow \ a[q_1, B_1, q_2][q_2, B_2, q_3] \cdots [q_n, B_n, q_{n+1}] \ ,$$

and thus $[q, A, p] \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} a y_1 y_2 \cdots y_n = x$. This proves the sufficiency of (9.1).

For showing the necessity of (9.1) suppose $[a, A, p] \overset{i}{\underset{\mathcal{G}}{\Longrightarrow}} x$. We prove by induction on i that $(q, x, A) \xrightarrow[\mathcal{K}]{*} (p, \lambda, \lambda)$. For the induction basis we again take $i = 1$. If $[a, A, p] \overset{1}{\underset{\mathcal{G}}{\Longrightarrow}} x$, then $([a, A, p] \ \to \ x) \in P$ and therefore $(p, \lambda) \in \delta(q, x, A)$.

Next, for the induction step suppose

$$[q, A, p] \ \Rightarrow \ a[q_1, B_1, q_2] \cdots [q_n, B_n, q_{n+1}] \overset{i-1}{\underset{\mathcal{G}}{\Longrightarrow}} x \ ,$$

where $q_{n+1} = p$. Then we may write x as $x = a x_1 \cdots x_n$, where $[q_j, B_j, q_{j+1}] \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x_j$ for $j = 1, \ldots, n$. Moreover, each derivation takes fewer than i steps. Thus, we can apply the induction hypothesis and obtain

$$(q_j, x_j, B_j) \xrightarrow[\mathcal{K}]{*} (q_{j+1}, \lambda, \lambda) \text{ for } j = 1, \ldots, n \ .$$

If we insert $B_{j+1} \cdots B_n$ at the bottom of each stack in the above sequence of instantaneous descriptions we see that

$$(q_j, x_j, B_j B_{j+1} \cdots B_n) \xrightarrow[\mathcal{K}]{*} ((q_{j+1}, \lambda, B_{j+1} \cdots B_n) \; . \tag{9.2}$$

Furthermore, from the first step in the derivation of x from $[q, A, p]$ we know that

$$(q, x, A) \xrightarrow[\mathcal{K}]{1} (q_1, x_1 x_2 \cdots x_n, B_1 B_2 \cdots B_n)$$

is a legal move of $\mathcal{K}$. Therefore, from this move and from (9.2) for $j = 1, 2, \ldots, n$ we directly obtain

$$(q, x, A) \xrightarrow[\mathcal{K}]{*} (p, \lambda, \lambda) \; .$$

This proves the necessity of (9.1).

The proof concludes with the observation that (9.1) with $q = q_0$ and $A = k_0$ says

$$[q_0, k_0, p] \underset{\mathcal{G}}{\overset{*}{\Longrightarrow}} \qquad \text{if and only if} \qquad (q_0, x, k_0) \xrightarrow[\mathcal{K}]{*} (p, \lambda, \lambda) \; .$$

This observation together with rule (1) of the construction of P says that $\sigma \underset{\mathcal{G}}{\overset{*}{\Longrightarrow}} x$ if and only if $(q_0, x, k_0) \xrightarrow[\mathcal{K}]{*} (p, \lambda, \lambda)$ for some state p. Therefore, we finally arrive at $x \in L(\mathcal{G})$ if and only if $x \in N(\mathcal{K})$. ∎

Lecture 10: $\mathcal{CF}$, PDAs and Beyond

Next, we want to show that all context-free languages are accepted by pushdown automata. For doing this, it is very convenient to use another normal form for context-free languages, i.e., the so-called Greibach normal form.

10.1. Greibach Normal Form

Definition 31. *A context-free grammar* $\mathcal{G} = [T, N, \sigma, P]$ *is said to be in* ***Greibach normal form*** *if every production of* $\mathcal{G}$ *has the form* $h \to a\alpha$*, where* $a \in T$ *and* $\alpha \in (N \setminus \{\sigma\})^*$.

Clearly, we aim to show that every context-free language does possess a grammar in Greibach normal form. This is, however, not as easy as it might seem. First, we need the following lemmata. For providing them, we need the following notion. For a context-free grammar $\mathcal{G}$ we define an **h-*production*** to be a production with nonterminal h on the left.

Lemma 10.1. *Let* $\mathcal{G} = [T, N, \sigma, P]$ *be any context-free grammar. Furthermore, let* $h \to \alpha_1 h' \alpha_2$ *be any production from* P*, where* $h, h' \in N$ *and let* $h' \to \beta_1$*,* $h' \to \beta_2$*, ...,* $h' \to \beta_r$ *be all* h'*-productions. Let* $\mathcal{G}_1 = [T, N, \sigma, P_1]$ *be the grammar obtained from* $\mathcal{G}$ *by deleting the production* $h \to \alpha_1 h' \alpha_2$ *and by adding the productions* $h \to \alpha_1\beta_1\alpha_2$*,* $h \to \alpha_1\beta_2\alpha_2$*,...,* $h \to a_1\beta_r\alpha_2$*. Then* $L(\mathcal{G}) = L(\mathcal{G}_1)$.

Proof. The inclusion $L(\mathcal{G}_1) \subseteq L(\mathcal{G})$ is obvious, since if $h \to \alpha_1\beta_i\alpha_2$ is used in a derivation of $\mathcal{G}_1$, then

$$h \underset{\mathcal{G}}{\overset{1}{\Longrightarrow}} \alpha_1 h' \alpha_2 \underset{\mathcal{G}}{\overset{1}{\Longrightarrow}} \alpha_1\beta_i\alpha_2$$

can be used in $\mathcal{G}$.

For the opposite direction $L(\mathcal{G}) \subseteq L(\mathcal{G}_1)$ one simply notes that $h \to \alpha_1 h' \alpha_2$ is the only production which is in $\mathcal{G}$ but not in $\mathcal{G}_1$. Whenever this production is used in a derivation by $\mathcal{G}$, the nonterminal h' must be rewritten at some late step by using a production of the form $h' \to \beta_i$ for some $i \in \{1, \ldots, r\}$. These two steps can be replaced by the single step

$$h' \underset{\mathcal{G}_1}{\overset{1}{\Longrightarrow}} \alpha_1\beta_i\alpha_2 \ ,$$

and thus we are done. ∎

Lemma 10.2. $\mathcal{G} = [T, N, \sigma, P]$ *be any context-free grammar. Let* $h \to h\alpha_1$*, ...,* $h \to h\alpha_r$ *be all* h*-productions for which* h *is the leftmost symbol of the right-hand side. Furthermore, let* $h \to \beta_1$*, ...,* $h \to \beta_s$ *be the remaining* h*-productions. Let* $\mathcal{G}_1 = [T, N \cup \{B\}, \sigma, P_1]$ *be the context-free grammar formed by adding the nonterminal* B *to* N *and by replacing all the* h*-productions by* $h \to \beta_i$ *and* $h \to \beta_i B$ *for*

$i = 1, \ldots, s$ *and all the remaining* h*-productions by* $B \to \alpha_j$ *and* $B \to \alpha_j B$, $j = 1, \ldots, r$. *Then* $L(\mathcal{G}) = L(\mathcal{G}_1)$.

We leave it as an exercise to show Lemma 10.1.

We continue by providing the following fundamental theorem.

Theorem 10.3. *For every language* $L \in \mathcal{CF}$ *with* $\lambda \notin L$ *there exists a grammar* $\tilde{\mathcal{G}}$ *such that* $L = L(\tilde{\mathcal{G}})$ *and* $\tilde{\mathcal{G}}$ *is in Greibach normal form.*

Proof. Let L be any context-free language with $\lambda \notin L$. Then there exists a grammar $\mathcal{G} = [T, N, \sigma, P]$ in Chomsky normal such that $L = L(\mathcal{G})$. Let $N = \{h_1, h_2, \ldots, h_m\}$. The first step in the construction of the Greibach normal form is to modify the productions of $\mathcal{G}$ in a way such that if $h_i \to h_j\gamma$ is a production, then $j > i$. Starting with h_1 and proceeding to h_m this is done as follows.

We assume that the productions have been modified so that for $1 \leqslant i < k$, $h_i \to h_j\gamma$ is a production only if $j > i$. Now, we modify the h_k-productions.

```
begin
(1)  for k := 1 to m do
       begin
(2)      for j := 1 to k − 1 do
(3)          for each production of the form h_k → h_j α do
                 begin
(4)                  for all productions h_j → β do
(5)                  add production h_k → βα;
(6)                  remove production h_k → h_j α
                 end;
(7)      for each production of the form h_k → h_k α do
             begin
(8)              add productions B_k → α and B_k → αB_k;
(9)              remove production h_k → h_k α
             end;
(10)     for each production h_k → β, where β does not
             begin with h_k do
(11)             add production h_k → βB_k
       end
   end
```

Figure 10.1: Step 1 in the Greibach normal form algorithm

If $h_k \to h_j\gamma$ is a production with $j < k$, then we generate a new set of productions by substituting for h_j the right-hand side of each h_j-production according to Lemma 10.1. By repeating the process at most $k-1$ times, we obtain productions of the form $h_k \to h_\ell\gamma$, where $\ell \geqslant k$. The productions with $\ell = k$ are then replaced by applying Lemma 10.2. That means we have to introduce a new nonterminal B_k. The complete algorithm is provided in Figure 10.1.

By repeating the above process for each original variable, we have only productions of the forms:

1) $H_i \to H_j\gamma, \quad j > i,$
2) $H_j a\gamma, \qquad a \in T,$
3) $B_i \to \gamma, \qquad \gamma \in (N \cup \{B_1, B_2, \ldots, B_{j-1}\})^*.$

Note that the leftmost symbol on the right-hand side of any production for H_m must be a terminal, since H_m is the highest-numbered nonterminal. The leftmost symbol on the right-hand side of any production for H_{m-1} must be either H_m or a terminal symbol. When it is H_m, we can generate new productions by replacing H_m by the right-hand side of the productions for H_m according to Lemma 10.1. These productions must have right-hand sides that start with a terminal symbol. We then proceed to the productions for $H_{m-2}, \ldots, H_2, H_1$ until the right hand-side of each production for an H_i starts with a terminal symbol.

As the last step we examine the productions of the new nonterminals $B_1, \ldots, B_m$. Since we started with a grammar in Chomsky normal form, it is easy to prove by induction on the number of applications of Lemmata 10.1 and 10.2 that the right-hand side of every H_i-production, $1 \leqslant i \leqslant n$, begins with a terminal or H_jH_k for some j and k. Thus α in Instruction (7) of Figure 10.1 can never be empty or begin with some B_j. So no B_i production can start with another B_j. Therefore, all B_i productions have right hand sides beginning with terminals or H_i's, and one more application of Lemma 10.1 for each B_i-production completes the construction. ∎

Since the construction outlined above is rather complicated, we exemplify it by the following example.

Example 13. *Let* $G = [\{a, b\}, \{H_1, H_2, H_3\}, H_1, P]$*, where*

$$P = \{H_1 \to H_2H_3,\ H_2 \to H_3H_1,\ H_2 \to b,\ H_3 \to H_1H_2,\ H_3 \to a\}\ .$$

We want to convert G into Greibach normal form.

Step 1. Since the right-hand side of the productions for H_1 and H_2 start with terminals or higher numbered nonterminals, we begin with the production $H_3 \to H_1H_2$ and substitute the string H_2H_3 for H_1. Note that $H_1 \to H_2H_3$ is the only production with H_1 on the left.

The resulting set of productions is:

$$\begin{aligned} H_1 &\to H_2H_3 \\ H_2 &\to H_3H_1 \\ H_2 &\to b \\ H_3 &\to H_2H_3H_2 \\ H_3 &\to a \end{aligned}$$

Since the right-hand side of the production $H_3 \to H_2H_3H_2$ begins with a lower numbered nonterminal, we substitute for the first occurrence of H_2 both H_3H_1 and b.

Thus, $\mathtt{h}_3 \rightarrow \mathtt{h}_2\mathtt{h}_3\mathtt{h}_2$ is replaced by $\mathtt{h}_3 \rightarrow \mathtt{h}_3\mathtt{h}_1\mathtt{h}_3\mathtt{h}_2$ and $\mathtt{h}_3 \rightarrow \mathtt{b}\mathtt{h}_3\mathtt{h}_2$. The new set is

$$\begin{array}{lcl}
\mathtt{h}_1 & \rightarrow & \mathtt{h}_2\mathtt{h}_3 \\
\mathtt{h}_2 & \rightarrow & \mathtt{h}_3\mathtt{h}_1 \\
\mathtt{h}_2 & \rightarrow & \mathtt{b} \\
\mathtt{h}_3 & \rightarrow & \mathtt{h}_3\mathtt{h}_1\mathtt{h}_3\mathtt{h}_2 \\
\mathtt{h}_3 & \rightarrow & \mathtt{b}\mathtt{h}_3\mathtt{h}_2 \\
\mathtt{h}_3 & \rightarrow & \mathtt{a}
\end{array}$$

Now, we apply Lemma 10.2 to the productions $\mathtt{h}_3 \rightarrow \mathtt{h}_3\mathtt{h}_1\mathtt{h}_3\mathtt{h}_2$, $\mathtt{h}_3 \rightarrow \mathtt{b}\mathtt{h}_3\mathtt{h}_2$ and $\mathtt{h}_3 \rightarrow \mathtt{a}$.

Symbol $\mathtt{B}_3$ is introduced, and the production $\mathtt{h}_3 \rightarrow \mathtt{h}_3\mathtt{h}_1\mathtt{h}_3\mathtt{h}_2$ is replaced by $\mathtt{h}_3 \rightarrow \mathtt{b}\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3$, $\mathtt{h}_3 \rightarrow \mathtt{a}\mathtt{B}_3$, $\mathtt{B}_3 \rightarrow \mathtt{h}_1\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3$, and $\mathtt{B}_3 \rightarrow \mathtt{h}_1\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3$. For simplifying notation, we adopt the Backus-Naur notation. The resulting set is

$$\begin{array}{lcl}
\mathtt{h}_1 & \rightarrow & \mathtt{h}_2\mathtt{h}_3 \\
\mathtt{h}_2 & \rightarrow & \mathtt{h}_3\mathtt{h}_1 \mid \mathtt{b} \\
\mathtt{h}_3 & \rightarrow & \mathtt{b}\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3 \mid \mathtt{a}\mathtt{B}_3 \mid \mathtt{b}\mathtt{h}_3\mathtt{h}_2 \mid \mathtt{a} \\
\mathtt{B}_3 & \rightarrow & \mathtt{h}_1\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3 \mid \mathtt{h}_1\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3
\end{array}$$

Step 2. Now all the productions with $\mathtt{h}_3$ on the left have right-hand sides that start with terminals. These are used to replace $\mathtt{h}_3$ in the production $\mathtt{h}_2 \rightarrow \mathtt{h}_3\mathtt{h}_1$ and then the productions with $\mathtt{h}_2$ on the left are used to replace $\mathtt{h}_2$ in the production $\mathtt{h}_1 \rightarrow \mathtt{h}_2\mathtt{h}_3$. The result is the following.

$$\begin{array}{lcllcl}
\mathtt{h}_3 & \rightarrow & \mathtt{b}\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3 & \mathtt{h}_3 & \rightarrow & \mathtt{b}\mathtt{h}_3\mathtt{h}_2 \\
\mathtt{h}_3 & \rightarrow & \mathtt{a}\mathtt{B}_3 & \mathtt{h}_3 & \rightarrow & \mathtt{a} \\
\mathtt{h}_2 & \rightarrow & \mathtt{b}\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3\mathtt{h}_1 & \mathtt{h}_2 & \rightarrow & \mathtt{b}\mathtt{h}_3\mathtt{h}_2\mathtt{h}_1 \\
\mathtt{h}_2 & \rightarrow & \mathtt{a}\mathtt{B}_3\mathtt{h}_1 & \mathtt{h}_2 & \rightarrow & \mathtt{a}\mathtt{h}_1 \\
\mathtt{h}_2 & \rightarrow & \mathtt{b} & & & \\
\mathtt{h}_1 & \rightarrow & \mathtt{b}\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3\mathtt{h}_1\mathtt{h}_3 & \mathtt{h}_1 & \rightarrow & \mathtt{b}\mathtt{h}_3\mathtt{h}_2\mathtt{h}_1\mathtt{h}_3 \\
\mathtt{h}_1 & \rightarrow & \mathtt{a}\mathtt{B}_3\mathtt{h}_1\mathtt{h}_3 & \mathtt{h}_1 & \rightarrow & \mathtt{a}\mathtt{h}_1\mathtt{h}_3 \\
\mathtt{h}_1 & \rightarrow & \mathtt{b}\mathtt{h}_3 & & & \\
\mathtt{B}_3 & \rightarrow & \mathtt{h}_1\mathtt{h}_3\mathtt{h}_2 & \mathtt{B}_3 & \rightarrow & \mathtt{h}_1\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3
\end{array}$$

Step 3. The two $\mathtt{B}_3$-productions are converted to proper form, resulting in 10 more productions. That is, the productions $\mathtt{B}_3 \rightarrow \mathtt{h}_1\mathtt{h}_3\mathtt{h}_2$ and $\mathtt{B}_3 \rightarrow \mathtt{h}_1\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3$ are altered by substituting the right side of each of the five productions with $\mathtt{h}_1$ on the left for the first occurrences of $\mathtt{h}_1$. Thus, $\mathtt{B}_3 \rightarrow \mathtt{h}_1\mathtt{h}_3\mathtt{h}_2$ becomes

$$\begin{array}{lcllcl}
\mathtt{B}_3 & \rightarrow & \mathtt{b}\mathtt{h}_3\mathtt{h}_2\mathtt{B}_3\mathtt{h}_1\mathtt{h}_3\mathtt{h}_3\mathtt{h}_2 & \mathtt{B}_3 & \rightarrow & \mathtt{a}\mathtt{B}_3\mathtt{h}_1\mathtt{h}_3\mathtt{h}_3\mathtt{h}_2 \\
\mathtt{B}_3 & \rightarrow & \mathtt{b}\mathtt{h}_3\mathtt{h}_3\mathtt{h}_2 & \mathtt{B}_3 & \rightarrow & \mathtt{b}\mathtt{h}_3\mathtt{h}_2\mathtt{h}_1\mathtt{h}_3\mathtt{h}_3\mathtt{h}_2 \\
\mathtt{B}_3 & \rightarrow & \mathtt{a}\mathtt{h}_1\mathtt{h}_3\mathtt{h}_3\mathtt{h}_2 & & &
\end{array}$$

The other production for B_3 is replaced similarly. The final set of productions is thus

$$
\begin{array}{ll}
h_3 \to bh_3h_2B_3 & h_3 \to bh_3h_2 \\
h_3 \to aB_3 & h_3 \to a \\
h_2 \to bh_3h_2B_3h_1 & h_2 \to bh_3h_2h_1 \\
h_2 \to aB_3h_1 & h_2 \to ah_1 \\
h_2 \to b & \\
h_1 \to bh_3h_2B_3h_1h_3 & h_1 \to bh_3h_2h_1h_3 \\
h_1 \to aB_3h_1h_3 & h_1 \to ah_1h_3 \\
h_1 \to bh_3 & \\
B_3 \to bh_3h_2B_3h_1h_3h_3h_2B_3 & B_3 \to bh_3h_2B_3h_1h_3h_3h_2 \\
B_3 \to aB_3h_1h_3h_3h_2B_3 & B_3 \to aB_3h_1h_3h_3h_2 \\
B_3 \to bh_3h_3h_2B_3 & B_3 \to bh_3h_3h_2 \\
B_3 \to bh_3h_2h_1h_3h_3h_2B_3 & B_3 \to bh_3h_2h_1h_3h_3h_2 \\
B_3 \to ah_1h_3h_3h_2B_3 & B_3 \to ah_1h_3h_3h_2
\end{array}
$$

end (Example)

10.2. Main Theorem

Now, we are ready to show the remaining fundamental theorem concerning the power of pushdown automata.

Theorem 10.4. *For every language* $L \in \mathcal{CF}$ *there exists a pushdown automaton* $\mathcal{K}$ *such that* $L = N(\mathcal{K})$.

Proof. We assume that $\lambda \notin L$. It is left as an exercise to modify the construction for the case that $\lambda \in L$. Let $\mathcal{G} = [T, N, \sigma, P]$ be a context-free grammar in Greibach normal form such that $L = L(\mathcal{G})$. Furthermore, let

$$\mathcal{K} = [\{q\}, T, N, \delta, q, \sigma, \emptyset] \ ,$$

where $(q, \gamma) \in \delta(q, a, A)$ whenever $(A \to a\gamma) \in P$.

The pushdown automaton $\mathcal{K}$ simulates leftmost derivations of $\mathcal{G}$. Since $\mathcal{G}$ is in Greibach normal form, each sentential form[‖] in a leftmost derivation consists of a string x of terminals followed by a string of nonterminals α. $\mathcal{K}$ stores the suffix α of the left sentential form on its stack after processing the prefix x. Formally, we show the following claim.

Claim 1. $\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x\alpha$ *by a leftmost derivation if and only if* $(q, x, \sigma) \overset{*}{\underset{\mathcal{K}}{\longrightarrow}} (q, \lambda, \alpha)$.

[‖] A string α of terminals and nonterminals is called a *sentential form* if $\sigma \overset{*}{\Longrightarrow} \alpha$.

We start with the sufficiency. The prove is done by induction. That is, we assume $(q, x, \sigma) \xrightarrow[\mathcal{K}]{i} (q, \lambda, \alpha)$ and show $\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x\alpha$.

For the induction basis, let $i = 0$. That is, we assume $(q, x, \sigma) \xrightarrow[\mathcal{K}]{0} (q, \lambda, \alpha)$. By the definition of the reflexive transitive closure $\xrightarrow[\mathcal{K}]{*}$, this means nothing else than $(q, x, \sigma) = (q, \lambda, \alpha)$. Consequently, $x = \lambda$ and $\alpha = \sigma$. Obviously, by the definition of the reflexive transitive closure $\overset{*}{\underset{\mathcal{G}}{\Longrightarrow}}$ we can conclude $\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} \sigma$, again in zero steps. This proves the induction basis.

For the induction step, assume $i \geqslant 1$ and let $x = ya$, where $y \in T^*$. Now, we consider the next-to-last-step, i.e.,

$$(q, ya, \sigma) \xrightarrow[\mathcal{K}]{i-1} (q, a, \beta) \xrightarrow[\mathcal{K}]{1} (q, \lambda, \alpha) \; . \tag{10.1}$$

If we remove a from the end of the input string in the first i instantaneous descriptions of the sequence (10.1), we discover that $(q, y, \sigma) \xrightarrow[\mathcal{K}]{i-1} (q, \lambda, \beta)$, since a cannot influence $\mathcal{K}$'s behavior until it is actually consumed from the input. Thus, we can apply the induction hypothesis and obtain

$$\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} y\beta \; .$$

Taking into account that the pushdown automaton $\mathcal{K}$, while consuming a, is making the move $(q, a, \beta) \xrightarrow[\mathcal{K}]{1} (q, \lambda, \alpha)$, we directly get by construction that $\beta = A\gamma$ for some $A \in N$, $(A \rightarrow a\eta) \in P$ and $\alpha = \eta\gamma$. Hence, we arrive at

$$\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} y\beta \overset{1}{\underset{\mathcal{G}}{\Longrightarrow}} ya\eta\gamma = x\alpha \; .$$

This completes the sufficiency proof.

For showing the necessity suppose that $\sigma \overset{i}{\underset{\mathcal{G}}{\Longrightarrow}} x\alpha$ by a leftmost derivation. We prove by induction on i that $(q, x, \sigma) \xrightarrow[\mathcal{K}]{*} (q, \lambda, \alpha)$. The induction basis is again done for $i = 0$, and can be shown by using similar arguments as above.

For the induction step let $i \geqslant 1$ and suppose

$$\sigma \overset{i-1}{\underset{\mathcal{G}}{\Longrightarrow}} yA\gamma \overset{1}{\underset{\mathcal{G}}{\Longrightarrow}} y\alpha\eta\gamma \ ,$$

where $x = y\alpha$ and $\alpha = \eta\gamma$. By the induction hypothesis we directly get

$$(q, y, \sigma) \xrightarrow[\mathcal{K}]{*} (q, \lambda, A\gamma)$$

and thus $(q, y\alpha, \sigma) \xrightarrow[\mathcal{K}]{*} (q, \alpha, A\gamma)$. Since $(A \ \rightarrow \ \alpha\eta) \in P$, we can conclude that $(q, \eta) \in \delta(q, \alpha, A)$. Thus,

$$(q, x, \sigma) \xrightarrow[\mathcal{K}]{*} (q, \alpha, A\gamma) \xrightarrow[\mathcal{K}]{1} (q, \lambda, \alpha) \ ,$$

and the necessity follows. This proves Claim 1.

To conclude the proof of the theorem, we have only to note Claim 1 with $\alpha = \lambda$ says

$$\sigma \overset{*}{\underset{\mathcal{G}}{\Longrightarrow}} x \text{ if and only if } (q, x, \sigma) \xrightarrow[\mathcal{K}]{*} (q, \lambda, \lambda) \ .$$

That is, $x \in L(\mathcal{G})$ if and only if $x \in N(\mathcal{K})$. ∎

Theorems 9.3 and 10.4 as well as Theorems 9.1 and 9.2 together directly allow the following main theorem.

Theorem 10.5. *Let L be any language. Then the following three assertions are equivalent:*

(1) $L \in \mathcal{CF}$

(2) *There exists a pushdown automaton $\mathcal{K}_1$ such that $L = L(\mathcal{K}_1)$.*

(3) *There exists a pushdown automaton $\mathcal{K}_2$ such that $L = N(\mathcal{K}_2)$.*

After so much progress we may want to ask questions like whether or not $L(\mathcal{G}_1) \cap L(\mathcal{G}_2) = \emptyset$ for any given context-free grammars $\mathcal{G}_1$, $\mathcal{G}_2$. Remembering Exercise 28 you may be tempted to think this is not a too difficult task. But as a matter of fact, nobody succeeded to design an algorithm solving this problem for all context-free grammars. Maybe, there is a deeper reason behind this situation. Before we can explore such problems, we have to deal with computability.

On the other hand, so far we have studied regular and context-free languages. But we have already seen a language which is not context-free, i.e., $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$.

Thus, it is only natural to ask what other language families are around. Due to the lack of time, we can only sketch these parts of formal language theory.

10.3. Context-Sensitive Languages

First, we provide a formal definition.

Definition 32. *A Grammar* $\mathcal{G} = [T, N, \sigma, P]$ *is said to be* ***context-sensitive*** *if all its productions satisfy the following conditions.*

(1) $(\alpha \rightarrow \beta) \in P$ *iff there are* s_1, s_2, r, *and* h *such that* $h \in N$, $s_1, s_2 \in (T \cup N)^*$ *and* $r \in (T \cup N)^+$ *and* $\alpha = s_1 h s_2$ and $\beta = s_1 r s_2$, or

(2) $\alpha = h$ and $\beta = \lambda$ and h *does not occur at any right-hand side of a production from* P.

Definition 33. *A language is said to be* ***context-sensitive*** *if there exists a context-sensitive grammar* $\mathcal{G}$ *such that* $L = L(\mathcal{G})$.

By $\mathcal{CS}$ we denote the family of all context-sensitive languages. The name context-sensitive is quite intuitive, since the replacement or rewriting of a nonterminal is only possible in a certain context expressed by a prefix s_1 and suffix s_2. The definition provided above directly allows the observation that $\mathcal{CF} \subseteq \mathcal{CS}$.

Exercise 36. *Prove that* $\mathcal{CF} \subset \mathcal{CS}$.

Furthermore, using the same ideas *mutatis mutandis* as in the proof of Theorem 6.2 one can easily show the following.

Theorem 10.6. *The context-sensitive languages are closed under union, product and Kleene closure.*

The proof is left as an exercise.

Also, in the same way as Theorem 6.4 has been shown, one can prove the context-sensitive languages to be closed under transposition. That is, we directly get the next theorem.

Theorem 10.7. *Let* Σ *be any alphabet, and let* $L \subseteq \Sigma^*$. *Then we have: If* $L \in \mathcal{CS}$ *then* $L^T \in \mathcal{CS}$, *too.*

Moreover, in contrast to Theorem 6.5 we have:

Theorem 10.8. *The context-sensitive languages are closed under intersection.*

For establishing further properties of context-sensitive languages, we need the following definition.

Definition 34. *A Grammar* $\mathcal{G} = [T, N, \sigma, P]$ *is said to be* ***length-increasing*** *if for each production* $(\alpha \rightarrow \beta) \in P$ *the condition* $|\alpha| \leqslant |\beta|$ *is satisfied. In addition,* P *may contain the production* $\sigma \rightarrow \lambda$ *and in this case* σ *does not occur on the right-hand side of any production from* P.

Looking at Definition 32 directly allows the following corollary.

Corollary 10.9. *Every context-sensitive grammar is length-increasing.*

Nevertheless, one can show the following.

Theorem 10.10. *For every length-increasing grammar there exists an equivalent context-sensitive grammar.*

The proof of the latter theorem is also left as an exercise. Putting Corollary 10.9 and Theorem 10.10 together directly yields the following equivalence.

Theorem 10.11. *Let* L *be any language. Then the following statements are equivalent:*

(1) *there exists a context-sensitive grammar* $\mathcal{G}$ *such that* $\mathsf{L} = \mathsf{L}(\mathcal{G})$,

(2) *there exists a length-increasing grammar* $\tilde{\mathcal{G}}$ *such that* $\mathsf{L} = \mathsf{L}(\tilde{\mathcal{G}})$.

Example 14. Let T be any alphabet. We define a grammar $\mathcal{G} = [\mathsf{T}, \mathsf{N}, \sigma, \mathsf{P}]$ as follows. Let

$$\mathsf{N} = \{\sigma\} \cup \{\mathsf{X}_\mathfrak{i} \mid \mathfrak{i} \in \mathsf{T}\} \cup \{\mathsf{A}_\mathfrak{i} \mid \mathfrak{i} \in \mathsf{T}\} \cup \{\mathsf{B}_\mathfrak{i} \mid \mathfrak{i} \in \mathsf{T}\} .$$

and the following set of productions

1. $\sigma \rightarrow \mathfrak{i}\sigma\mathsf{X}_\mathfrak{i}$
2. $\sigma \rightarrow \mathsf{A}_\mathfrak{i}\mathsf{B}_\mathfrak{i}$
3. $\mathsf{B}_\mathfrak{i}\mathsf{X}_\mathfrak{j} \rightarrow \mathsf{X}_\mathfrak{j}\mathsf{B}_\mathfrak{i}$
4. $\mathsf{B}_\mathfrak{i} \rightarrow \mathfrak{i}$
5. $\mathsf{A}_\mathfrak{i}\mathsf{X}_\mathfrak{j} \rightarrow \mathsf{A}_\mathfrak{i}\mathsf{B}_\mathfrak{j}$
6. $\mathsf{A}_\mathfrak{i} \rightarrow \mathfrak{i}$

where $\mathfrak{i}, \mathfrak{j} \in \mathsf{T}$.

Inspecting the productions we see that $\mathcal{G}$ is a length-increasing grammar which is not context-sensitive, since the context in Production 3 is destroyed. Nevertheless, by Theorem 10.10 we know that the language $\mathsf{L}(\mathcal{G})$ is context-sensitive. But what language is generated by $\mathcal{G}$? The answer is provided by our next exercise.

Exercise 37. *Prove that the grammar* $\mathcal{G}$ *given in Example 14 generates*

$$\mathsf{L} = \{ww \mid w \in \mathsf{T}^+\} .$$

Exercise 38. *Provide a context-sensitive grammar for the language*

$$\mathsf{L} = \{ww \mid w \in \mathsf{T}^+\} .$$

The notion of length-increasing grammar has another nice implication which we state next.

Theorem 10.12. *There is an algorithm that on input any context-sensitive grammar* $\mathcal{G} = [\mathsf{T}, \mathsf{N}, \sigma, \mathsf{P}]$ *and any string* $\mathsf{s} \in \mathsf{T}^*$ *decides whether or not* $\mathsf{s} \in \mathsf{L}(\mathcal{G})$.

Proof. (Sketch). Since every context-sensitive grammar is also a length-increasing grammar, it suffices to examine all finite sequences $w_0, w_1, \ldots, w_n$ with $|w_i| < |w_{i+1}|$, $i = 0, \ldots, n-1$ and $\sigma = w_0$ as well as $w_n = \mathsf{s}$, where $w_i \in (\mathsf{T} \cup \mathsf{N})^+$. The number of all those sequences is finite. Let $\mathcal{S}$ be the set of all such sequences.

Now, the only thing one has to check is whether or not

$$w_i \overset{1}{\underset{\mathcal{G}}{\Longrightarrow}} w_{i+1} \text{ for all } i = 0, \ldots, n-1 \; . \tag{10.2}$$

Thus, one either finds sequence in $\mathcal{S}$ fulfilling (10.2). Then one can directly conclude that $\mathsf{s} \in \mathsf{L}(\mathcal{G})$. If all sequences in $\mathcal{S}$ fail to satisfy (10.2) then $\mathsf{s} \notin \mathsf{L}(\mathcal{G})$. ∎

Recall that we have proved the regular languages to be closed under complement and the context-free languages to be *not* closed under complement. As curious as we are, we clearly like to know whether or not the context-sensitive languages are closed under complement. To answer this question is by no means easy. As a matter of fact, it took more than 20 years to resolve this question. So you have to be a bit patient. We shall provide an answer in the course on "Complexity and Cryptography."

Finally, we mention what happens if we pose no restrictions whatsoever on the set of productions. We shall call the resulting family of languages $\mathcal{L}_0$, where 0 should remind you to *zero* restrictions. Then, it is quite obvious that all languages L in $\mathcal{L}_0$ share the property that we can algorithmically *enumerate all and only* the elements contained in L provided we are given are grammar $\mathcal{G}$ for L. However, as we shall see later, Theorem 10.12 *cannot* be generalized to $\mathcal{L}_0$. So, we can directly conclude that $\mathcal{CS} \subset \mathcal{L}_0$. Putting it all together gives us the famous Chomsky Hierarchy, i.e.,

$$\mathcal{REG} \subset \mathcal{CF} \subset \mathcal{CS} \subset \mathcal{L}_0 \; .$$

Part 2: Computability

Lecture 11: Models of Computation

The history of algorithms goes back, approximately, to the origins of mathematics at all. For thousands of years, in most cases, the solution of a mathematical problem had been equivalent to the construction of an algorithm that *solved* it. The ancient development of algorithms culminated in Euclid's famous *Elements*. For example, in Book VII of the elements we find the Euclidean algorithm for computing the greatest common divisor of two integers.

The Elements have been studied for 20 centuries in many languages starting, of course, in the original Greek, then in Arabic, Latin, and many modern languages.

A larger part of Euclid's *Elements* deals with the problem to construct geometrical figures by using only ruler and compass. Over the centuries, often quite different constructions have been proposed for certain problems. Moreover, in classical geometry there have been also a couple of construction problems that nobody could solve by using only ruler and compass. Perhaps the most famous of these problems are the trisection of an angle, squaring the circle and duplicating the cube. Another important example is the question which regular $\mathfrak{n}$-gons are constructible by using only ruler and compass. The latter problem was only resolved by Gauss in 1798.

However, even after Lindemann's proof in 1882 that it is impossible to square the circle, it took roughly another 50 years before modern computability theory started. The main step to be still undertaken was to formalize the notion of algorithm. The famous impossibility results obtained for the classical geometrical problems "only" proved that there is no particular type of algorithm solving, e.g. the problem to square the circle. Here the elementary operations are the application of ruler and compass.

So what else can be said concerning the notion of algorithm? The term *algorithm* is derived from the name of Al-Hwarizmi (approx.: 780 - 850) who worked in the *house of wisdom* in Bagdad. He combined the scientific strength of Greek mathematics with the versatility of Indian mathematics to perform calculations.

Another influential source for the development of our thinking was Raimundus Lullus (1232 - 1316) who published more than 280 papers. His *Ars magna* (Engl.: the great art) developed the idea to logically combine termini by using a machine.

Inspired by Lullus' ideas Leibniz (1646 - 1716) split the *Ars magna* into an *Art indicanti* (decision procedures) and an *Art inveniendi* (generation or enumeration procedures). Here by *decision procedure* a method is meant which, for every object, can *find out* within a *finite* amount of time whether or not it possesses the property asked for.

An *enumeration procedure* outputs all and only the objects having the property asked for. Therefore, in a finite amount of time we can only detect that an object has the property asked for, if it has this property. For objects not possessing the property

asked for, in general, within a finite amount of time we cannot detect the absence of the property.

Also, Leibniz pointed out that both decision procedures and enumeration procedures must be realizable by a machine. He also designed a machine for the four basic arithmetic operations and presented it in 1673 at the Royal Society in London. Note that also Schickardt (1624) and Pascal (1641) have built such machines. As a matter of fact, Leibniz was convinced that one can find for any problem an algorithm solving it.

But there have been problems around that could not be solved despite enormous efforts of numerous mathematicians. For example, the design of an algorithm deciding whether a given Diophantine equation has an integral solution (Hilbert's 10th problem) remained unsolved until 1967 when it was shown by Matijasevic that there is no such algorithm.

So, modern computation theory starts with the question:

Which problems can be solved algorithmically ?

In order to answer it, first of all, the *intuitive notion of an algorithm has to be formalized mathematically.*

Hopefully, you also have an intuitive understanding of what an algorithm is. But what is meant by "there is no algorithm solving problem Π," where Π is e.g. Hilbert's 10th problem or the problem we stated at the end of Lecture 9? Having a particular algorithm on hand, we may be able to check if it is solving problem Π. However, what can we say about all the algorithms still to be discovered? How can we know that none of them will ever solve problem Π?

For a better understanding of the problem we may think of algorithms as of computer programs. There are many computer programs around and many projects under development. So, while you are sleeping, a new computer program may be written. In general, we have no idea who is writing what computer program. There are very talented programmers around. So, how can we be sure that by tomorrow or during the next year, or even during the next decade there will be no program solving our problem Π?

This is the point where the beauty and strength of mathematics comes into play. Let us see how we can get started. Looking at all the algorithms we know, we can say that an algorithm is a computation method having the following properties.

(1) The instruction is a finite text.

(2) The computation is done step by step, where each step performs an elementary operation.

(3) In each step of the execution of the computation it is uniquely determined which elementary operation we have to perform.

(4) The next computation step depends only on the input and the intermediate results computed so far.

Now, we can also assume that there is a finite alphabet Σ such that every algorithm can be represented as a string from Σ^*. Since the number of all strings from Σ^* is *countably infinite* there are at most countably infinite many algorithms. Recalling Cantor's famous result that

$$\{f \mid f\colon \mathbb{N} \mapsto \{0,1\}\}$$

is *uncountably infinite*, we directly arrive at the following theorem.

Theorem 11.1. *There exists a noncomputable function* $f\colon \mathbb{N} \mapsto \{0,1\}$.

While this result is of fundamental epistemological importance, it is telling nothing about any particular function. For achieving results in this regard, we have to do much more. First, we have formalize the notion of algorithm in a mathematically precise way. Note that our description given above is not a formal one. We start with Gödel's [1] approach.

11.1. Partial Recursive Functions

For all $n \in \mathbb{N}^+$ we write $\mathcal{P}^n$ to denote the set of all partial recursive functions from $\mathbb{N}^n$ into $\mathbb{N}$. Here we define $\mathbb{N}^1 = \mathbb{N}$ and $\mathbb{N}^{n+1} = \mathbb{N}^n \times \mathbb{N}$, i.e., $\mathbb{N}^n$ is the set of all ordered n-tuples of natural numbers. Gödel's [1] idea to define the set $\mathcal{P}$ of all partial recursive functions is as follows.

Step (1): Define some basic functions which are intuitively computable.

Step (2): Define some rules that can be used to construct new computable functions from functions that are already known to be computable.

In order to complete Step (1), we define the following functions $Z,\ S,\ V\colon \mathbb{N} \mapsto \mathbb{N}$ by setting

$$\begin{aligned} Z(n) &= 0 \text{ for all } n \in \mathbb{N} \\ S(n) &= n+1 \text{ for all } n \in \mathbb{N} \\ V(n) &= \begin{cases} 0\,, & \text{if } n = 0 \\ n-1\,, & \text{for all } n \geqslant 1 \end{cases} \end{aligned}$$

That is, Z is the constant 0 function, S is the successor function and V is the predecessor function. Clearly, these functions are intuitively computable. Therefore, by definition we have $Z,\ S,\ V \in \mathcal{P}^1$. This completes Step (1) of the outline given above. Next, we define the rules (cf. Step (2)).

(2.1) (Introduction of fictitious variables)

Let $n \in \mathbb{N}^+$; then we have: if $\tau \in \mathcal{P}^n$ and $\psi(x_1, \ldots, x_n, x_{n+1}) =_{df} \tau(x_1, \ldots, x_n)$, then $\psi \in \mathcal{P}^{n+1}$.

(2.2) (Identifying variables)

Let $n \in \mathbb{N}^+$; then we have: if $\tau \in \mathcal{P}^{n+1}$ and $\psi(x_1, \ldots, x_n) =_{df} \tau(x_1, \ldots, x_n, x_n)$, then $\psi \in \mathcal{P}^n$.

(2.3) (Permuting variables)

Let $n \in \mathbb{N}^+$, $n \geqslant 2$ and let $i \in \{1, \ldots, n\}$; then we have: if $\tau \in \mathcal{P}^n$ and $\psi(x_1, \ldots, x_i, x_{i+1}, \ldots, x_n) =_{df} \tau(x_1, \ldots, x_{i+1}, x_i, \ldots, x_n)$, then $\psi \in \mathcal{P}^n$.

(2.4) (Composition)

Let $n \in \mathbb{N}$ and $m \in \mathbb{N}^+$. Furthermore, let $\tau \in \mathcal{P}^{n+1}$, let $\psi \in \mathcal{P}^m$ and define $\phi(x_1, \ldots, x_n, y_1, \ldots, y_m) =_{df} \tau(x_1, \ldots, x_n, \psi(y_1, \ldots, y_m))$. Then $\phi \in \mathcal{P}^{n+m}$.

(2.5) (Primitive recursion)

Let $n \in \mathbb{N}$, let $\tau \in \mathcal{P}^n$ and let $\psi \in \mathcal{P}^{n+2}$. Then we have: if

$$\begin{aligned} \phi(x_1, \ldots, x_n, 0) &=_{df} \tau(x_1, \ldots, x_n) \\ \phi(x_1, \ldots, x_n, y+1) &=_{df} \psi(x_1, \ldots, x_n, y, \phi(x_1, \ldots, x_n, y)) \ , \end{aligned}$$

then $\phi \in \mathcal{P}^{n+1}$.

(2.6) (μ-recursion)

Let $n \in \mathbb{N}^+$; then we have:

if $\tau \in \mathcal{P}^{n+1}$ and $\psi(x_1, \ldots, x_n) =_{df} \mu y[\tau(x_1, \ldots, x_n, y) = 1]$

$$=_{df} \begin{cases} \text{the smallest } y \text{ such that} & \\ (1)\ \tau(x_1, \ldots, x_n, v) \text{ is defined for all } v \leqslant y & \\ (2)\ \tau(x_1, \ldots, x_n, v) \neq 1 \text{ for all } v \leqslant y \text{ and} & \\ (3)\ \tau(x_1, \ldots, x_n, y) = 1 \ , & \text{if such a } y \text{ exists} \\ \text{not defined} \ , & \text{otherwise} \ . \end{cases}$$

then $\psi \in \mathcal{P}^n$.

Note that all operations given above except Operation (2.5) are explicit. Operation (2.5) itself constitutes an *implicit* definition, since ϕ appears on both the left and right hand side. Thus, before we can continue, we need to verify whether or not Operation (2.5) does *always* defines a function. This is by no means obvious. Recall that every implicit definition needs a justification. Therefore, we have to show the following theorem.

Theorem 11.2 (**Dedekind's Justification Theorem**.) *If τ and ψ are functions, then there is precisely one function ϕ satisfying the scheme given in Operation (2.5).*

Proof. We have to show uniqueness and existence of ϕ.

Claim 1. *There is at most one function ϕ satisfying the scheme given in Operation (2.5).*

Suppose there are functions ϕ_1 and ϕ_2 satisfying the scheme given in Operation (2.5). We show by induction over y that

$$\phi_1(x_1, \ldots, x_n, y) = \phi_2(x_1, \ldots, x_n, y) \text{ for all } x_1, \ldots, x_n, y \in \mathbb{N} .$$

For the induction basis, let $y = 0$. Then we directly get for all $x_1, \ldots, x_n \in \mathbb{N}$

$$\begin{aligned} \phi_1(x_1, \ldots, x_n, 0) &= \tau(x_1, \ldots, x_n) \\ &= \phi_2(x_1, \ldots, x_n, 0) . \end{aligned}$$

Now, we assume as induction hypothesis (abbr. IH) that for all $x_1, \ldots, x_n \in \mathbb{N}$ and some $y \in \mathbb{N}$

$$\phi_1(x_1, \ldots, x_n, y) = \phi_2(x_1, \ldots, x_n, y) .$$

The induction step is done from y to $y + 1$. Using the scheme provided in Operation (2.5) we obtain

$$\begin{aligned} \phi_1(x_1, \ldots, x_n, y + 1) &= \psi(x_1, \ldots, x_n, y, \phi_1(x_1, \ldots, x_n, y)) \text{ by definition} \\ &= \psi(x_1, \ldots, x_n, y, \phi_2(x_1, \ldots, x_n, y)) \text{ by the IH} \\ &= \phi_2(x_1, \ldots, x_n, y + 1) \text{ by definition} . \end{aligned}$$

Consequently $\phi_1 = \phi_2$, and Claim 1 is proved.

Claim 2. *There is a function* ϕ *satisfying the scheme given in Operation (2.5).*

For showing the existence of ϕ we replace the inductive and implicit definition of ϕ by an infinite sequence of explicit definitions, i.e., let

$$\phi_0(x_1, \ldots, x_n, y) = \begin{cases} \tau(x_1, \ldots, x_n), & \text{if } y = 0, \\ \text{not defined}, & \text{otherwise} . \end{cases}$$

$$\phi_1(x_1, \ldots, x_n, y) = \begin{cases} \phi_0(x_1, \ldots, x_n, y), & \text{if } y < 1, \\ \psi(x_1, \ldots, x_n, 0, \phi_0(x_1, \ldots, x_n, 0)), & \text{if } y = 1, \\ \text{not defined}, & \text{otherwise} . \end{cases}$$

$$\vdots$$

$$\phi_{i+1}(x_1, \ldots, x_n, y) = \begin{cases} \phi_i(x_1, \ldots, x_n, y), & \text{if } y < i + 1, \\ \psi(x_1, \ldots, x_n, i, \phi_i(x_1, \ldots, x_n, i)), & \text{if } y = i + 1, \\ \text{not defined}, & \text{otherwise} . \end{cases}$$

$$\vdots$$

All definitions of the functions ϕ_i are *explicit*, and thus the functions ϕ_i exist by the set forming axiom. Consequently, for $y \in \mathbb{N}$ and every $x_1, \ldots, x_n \in \mathbb{N}$ the function ϕ defined by

$$\phi(x_1, \ldots, x_n, y) =_{df} \phi_y(x_1, \ldots, x_n, y)$$

does exist. Furthermore, by construction we directly get

$$\begin{aligned}\phi(x_1,\ldots,x_n,0) &= \phi_0(x_1,\ldots,x_n,0)\\ &= \tau(x_1,\ldots,x_n) \text{ and}\\ \phi(x_1,\ldots,x_n,y+1) &= \phi_{y+1}(x_1,\ldots,x_n,y+1)\\ &= \psi(x_1,\ldots,x_n,y,\phi_y(x_1,\ldots,x_n,y))\\ &= \psi(x_1,\ldots,x_n,y,\phi(x_1,\ldots,x_n,y)) \ ,\end{aligned}$$

and thus, ϕ is satisfying the scheme given in Operation (2.5). ∎

Now, we are ready to define the class of all partial recursive functions.

Definition 35. *We define the class $\mathcal{P}$ of all **partial recursive functions** to be the smallest function class containing the functions Z, S and V and all functions that can be obtained from Z, S and V by finitely many applications of the Operations* (2.1) *through* (2.6).

That is $\mathcal{P} = \bigcup_{n\in\mathbb{N}^+} \mathcal{P}^n$.

Furthermore, we define the important subclass of primitive recursive functions as follows.

Definition 36. *We define the class Prim of all **primitive recursive functions** to be the smallest function class containing the functions Z, S and V and all functions that can be obtained from Z, S and V by finitely many applications of the Operations* (2.1) *through* (2.5).

Note that, by definition, we have $Prim \subseteq \mathcal{P}$. We continue with some examples.

Example 15. *The identity function $I\colon \mathbb{N} \mapsto \mathbb{N}$ defined by $I(x) = x$ for all $x \in \mathbb{N}$ is primitive recursive.*

Proof. We want to apply Operation (2.4). Let $n = 0$ and $m = 1$. By our definition (cf. Step (1)), we know that V, $S \in \mathcal{P}^1$. So, V serves as the τ (note that $n + 1 = 0 + 1 = 1$) and S serves as the ψ in Operation (2.4) (note that $m = 1$). Consequently, the desired function I is the ϕ in Operation (2.4) (note that $n + m = 0 + 1 = 1$) and we can set

$$I(x) = V(S(x)) \ .$$

Hence, the identity function I is primitive recursive.

Example 16. *The binary addition function $\alpha\colon \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ given by $\alpha(n, m) = n + m$ for all $n, m \in \mathbb{N}$ is is primitive recursive.*

Proof. By assumption, $S \in \mathcal{P}$. As shown in Example 15, $I \in Prim$. First, we define some auxiliary functions by using the operations indicated below.

$$\begin{aligned}\psi(x_1,x_2) &=_{df} S(x_1) \quad \text{by using Operation (2.1)},\\ \tilde{\psi}(x_1,x_2) &=_{df} \psi(x_2,x_1) \quad \text{by using Operation (2.3)},\\ \tau(x_1,x_2,x_3) &=_{df} \tilde{\psi}(x_1,x_2) \quad \text{by using Operation (2.1)},\\ \tilde{\tau}(x_1,x_2,x_3) &=_{df} \tau(x_1,x_3,x_2) \quad \text{by using Operation (2.3)} \ .\end{aligned}$$

Now, we are ready to apply Operation (2.5) for defining α, i.e., we set

$$\begin{aligned} \alpha(n,0) &=_{df} I(n) \\ \alpha(n,m+1) &=_{df} \tilde{\tau}(n,m,\alpha(n,m)) \ . \end{aligned}$$

Since we only used Operations (2.1) through (2.5), we see that $\alpha \in \mathit{Prim}$.

So, let us compute $\alpha(n,1)$. Then we get

$$\begin{aligned} \alpha(n,1) &= \alpha(n,0+1) = \tilde{\tau}(n,0,\alpha(n,0)) \ , \\ &= \tilde{\tau}(n,0,I(n)) \quad \text{by using } \alpha(n,0) = I(n) \\ &= \tilde{\tau}(n,0,n) \quad \text{by using } I(n) = n \ , \\ &= \tau(n,n,0) \quad \text{by using the definition of } \tilde{\tau} \ , \\ &= \tilde{\psi}(n,n) \quad \text{by using the definition of } \tau \ , \\ &= \psi(n,n) \quad \text{by using the definition of } \tilde{\psi} \ , \\ &= S(n) = n+1 \quad \text{by using the definition of } \psi \text{ and } S \ . \end{aligned}$$

So, our definition may look more complex than necessary. In order to see, it is not, we compute $\alpha(n,2)$.

$$\begin{aligned} \alpha(n,2) &= \alpha(n,1+1) = \tilde{\tau}(n,0,\alpha(n,1)) \ , \\ &= \tilde{\tau}(n,0,n+1) \quad \text{by using } \alpha(n,0) = n+1 \ , \\ &= \tau(n,n+1,0) \\ &= \tilde{\psi}(n,n+1) \\ &= \psi(n+1,n) \\ &= S(n+1) = n+2 \ . \end{aligned}$$

In the following we shall often omit some of the tedious technical steps. For example, in order to clarify that binary multiplication is primitive recursive, we simply point out that is suffices to set

$$\begin{aligned} m(x,0) &=_{df} Z(x) \\ m(x,y+1) &=_{df} \alpha(x,m(x,y)) \ . \end{aligned}$$

Also note that the constant 1 function c is primitive recursive, i.e., $c(n) = 1$ for all $n \in \mathbb{N}$. For seeing this, we set

$$\begin{aligned} c(0) &= S(0) \ , \\ c(n+1) &= c(n) \ . \end{aligned}$$

In the following, instead of $c(n)$ we just write 1.

Now, it is easy to see that the signum function sg is primitive recursive, since we have

$$\begin{aligned} sg(0) &= 0 \ , \\ sg(n+1) &= 1 \ . \end{aligned}$$

Since the natural numbers are not closed under subtraction, one conventionally uses the so-called arithmetic difference defined as $n \dot{-} m = m - n$ if $m \geqslant n$ and 0 otherwise. The arithmetic difference is primitive recursive, too, since for all $n, m \in \mathbb{N}$ we have

$$\begin{aligned} m \dot{-} 0 &= I(m) \,, \\ m \dot{-} (n+1) &= V(m \dot{-} m) \,. \end{aligned}$$

Occasionally, we shall also need $\overline{sg}$ defined as

$$\overline{sg}(n) = 1 \dot{-} sg(n) \,.$$

As the above definition shows, $\overline{sg}$ is also primitive recursive.

Moreover, we can easily extend binary addition and multiplication to any fixed number k of arguments. For example, in order to define ternary the addition function α^3 ($k = 3$), we apply Operation (2.4) and set

$$\alpha^3(x_1, x_2, x_3) =_{df} \alpha(x_1, \alpha(x_2, x_3)) \,.$$

Iterating this idea yields k-ary addition. Of course, we can apply it *mutatis mutandis* to multiplication. For the sake of notational convenience we shall use the more common $\sum_{i=1}^{k} x_i$ and $\prod_{i=1}^{k} x_i$ to denote k-ary addition and k-ary multiplication, respectively. Also, from now on we shall use again the common $+$ and $\cdot$ to denote addition and multiplication whenever appropriate.

Example 17. *Let $g \in \mathcal{P}^{n+1}$ be any primitive recursive function. Then*

$$f(x_1, \ldots, x_n, k) = \sum_{i=1}^{k} g(x_1, \ldots, x_n, i)$$

is primitive recursive.

Proof. By Example 16 we know that α is primitive recursive. Then the function f is obtained by applying Operations (2.5) and (2.4), i.e.,

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n, 0) \\ f(x_1, \ldots, x_n, k+1) &= \alpha(f(x_1, \ldots, x_n, k), g(x_1, \ldots, x_n, k+1)) \,. \end{aligned}$$

Hence, f is primitive recursive. ∎

Analogously, one can show that the general multiplication is primitive recursive. That is, if g is as in Example 17 then $f(x_1, \ldots, x_n, k) = \prod_{i=1}^{k} g(x_1, \ldots, x_n, i)$ is primitive recursive.

Quite often one is defining functions by making case distinctions (cf., e.g., our definition of the predecessor function V). So, it is only natural to ask under what circumstances definitions by case distinctions do preserve primitive recursiveness. A

convenient way to describe properties is the usage of *predicates*. An $\mathfrak{n}$-ary predicate $\mathfrak{p}$ over the natural numbers is a subset of $\mathsf{N}^{\mathfrak{n}}$. Usually, one writes $\mathfrak{p}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}})$ instead of $(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) \in \mathfrak{p}$. The characteristic function of $\mathfrak{n}$-ary predicate $\mathfrak{p}$ is the function $\chi_{\mathfrak{p}} \colon \mathbb{N}^{\mathfrak{n}} \mapsto \{0, 1\}$ defined by

$$\chi_{\mathfrak{p}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) = \begin{cases} 1, & \text{if } \mathfrak{p}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) \\ 0, & \text{otherwise .} \end{cases}$$

A predicate $\mathfrak{p}$ is said to be *primitive recursive* if $\chi_{\mathfrak{p}}$ is primitive recursive. Let $\mathfrak{p}, \mathfrak{q}$ be $\mathfrak{n}$-ary predicates, then we define $\mathfrak{p} \wedge \mathfrak{q}$ to be the set $\mathfrak{p} \cap \mathfrak{q}$, $\mathfrak{p} \vee \mathfrak{q}$ to be the set $\mathfrak{p} \cup \mathfrak{q}$ and $\neg\mathfrak{p}$ to be the set $\mathsf{N}^{\mathfrak{n}} \setminus \mathfrak{p}$.

Lemma 11.3. *Let $\mathfrak{p}, \mathfrak{q}$ be any primitive recursive $\mathfrak{n}$-ary predicates. Then $\mathfrak{p} \wedge \mathfrak{q}$, $\mathfrak{p} \vee \mathfrak{q}$, and $\neg\mathfrak{p}$ are also primitive recursive.*

Proof. Obviously, it holds

$$\begin{aligned} \chi_{\mathfrak{p}\wedge\mathfrak{q}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) &= \chi_{\mathfrak{p}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) \cdot \chi_{\mathfrak{q}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) \ , \\ \chi_{\mathfrak{p}\vee\mathfrak{q}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) &= \chi_{\mathfrak{p}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) + \chi_{\mathfrak{q}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) \ , \\ \chi_{\neg\mathfrak{p}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) &= 1 \dot{-} \chi_{\mathfrak{p}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) \ . \end{aligned}$$

Since we already know addition, multiplication and the arithmetic difference to be primitive recursive, the assertion of the lemma follows. ∎

Exercise 39. *Show the binary predicates $=$, $<$, and $\leqslant$ defined as usual over $\mathbb{N} \times \mathbb{N}$ to be primitive recursive.*

Now, we can show our theorem concerning function definitions by making case distinctions.

Theorem 11.4. *Let $\mathfrak{p}_1, \ldots, \mathfrak{p}_{\mathfrak{k}}$ be pairwise disjoint $\mathfrak{n}$-ary primitive recursive predicates, and let $\psi_1, \ldots, \psi_{\mathfrak{k}} \in \mathcal{P}^{\mathfrak{n}}$ be primitive recursive functions. Then the function $\gamma \colon \mathbb{N}^{\mathfrak{n}} \mapsto \mathbb{N}$ defined by*

$$\gamma(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) = \begin{cases} \psi_1(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}), & \text{if } \mathfrak{p}_1(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) \\ \cdot & \\ \cdot & \\ \cdot & \\ \psi_{\mathfrak{k}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}), & \text{if } \mathfrak{p}_{\mathfrak{k}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) \\ 0, & \text{otherwise .} \end{cases}$$

is primitive recursive.

Proof. Since we can write γ as

$$\gamma(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) = \sum_{\mathfrak{i}=1}^{\mathfrak{k}} \chi_{\mathfrak{p}_{\mathfrak{i}}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) \cdot \psi_{\mathfrak{i}}(\mathfrak{x}_1, \ldots, \mathfrak{x}_{\mathfrak{n}}) \ ,$$

the theorem follows from the primitive recursiveness of general addition and multiplication. ∎

11.2. Pairing Functions

Quite often it would be very useful to have a bijection from $\mathbb{N}\times\mathbb{N}$ to $\mathbb{N}$. So, first we have to ask whether or not such a bijection does exist. This is indeed the case. Recall that the elements of $\mathbb{N}\times\mathbb{N}$ are ordered pairs of natural numbers. So, we may easily represent all elements of $\mathbb{N}\times\mathbb{N}$ in a two dimensional array, where row x contains all pairs (x,y), i.e., having x in the first component and $y = 0,1,2,\ldots$ (cf. Figure 11.1).

$$\begin{array}{llllll}
(0,0) & (0,1) & (0,2) & (0,3) & (0,4) & \ldots \\
(1,0) & (1,1) & (1,2) & (1,3) & (1,4) & \ldots \\
(2,0) & (2,1) & (2,2) & (2,3) & (2,4) & \ldots \\
(3,0) & (3,1) & (3,2) & (3,3) & (3,4) & \ldots \\
(4,0) & (4,1) & (4,2) & (4,3) & (4,4) & \ldots \\
(5,0) & \ldots & & & & \\
\ldots & \ldots & & & &
\end{array}$$

Figure 11.1: A two dimensional array representing $\mathbb{N}\times\mathbb{N}$.

Now, the idea is to arrange all these pairs in a sequence starting

$$(0,0),\ (0,1),\ (1,0),\ (0,2),\ (1,1),\ (2,0),\ (0,3),\ (1,2),\ (2,1),\ (3,0),\ldots \tag{11.1}$$

In this order, all pairs (x,y) appear before all pairs (x',y') if and only if $x+y < x'+y'$. That is, they are arranged in order of incrementally growing component sums. The pairs with the same component sum are ordered by the first component starting with the smallest one. That is, pair (x,y) is located in the segment

$$(0,x+y),\ \ (1,x+y-1),\ \ \ldots,\ \ (x,y),\ \ \ldots,\ \ (x+y,0)\ .$$

Note that there are $x+y+1$ many pairs having the component sum $x+y$. Thus, in front of pair $(0,x+y)$ we have in the Sequence (11.1) $x+y$ many segments containing a total of

$$1+2+3+\cdots+(x+y)$$

many pairs. Taking into account that

$$\sum_{i=0}^{n} i \ = \ \frac{n(n+1)}{2} \ = \ \sum_{i=1}^{n} i$$

we thus can define the desired bijection $c\colon \mathbb{N}\times\mathbb{N}\mapsto\mathbb{N}$ by setting

$$\begin{aligned} c(x,y) &= \frac{(x+y)(x+y+1)}{2} + x \\ &= \frac{(x+y)^2+3x+y}{2}\ . \end{aligned} \tag{11.2}$$

Note that we start counting with 0 in the Sequence (11.1), since otherwise we would not obtain a bijection. So, pair $(0,0)$ is at the 0th position, pair $(0,1)$ at the 1st,

.... Let us make a quick check by computing $c(1,0)$ and $c(0,2)$. We directly obtain $c(1,0) = ((1+0)^2+3+0)/2 = 2$ and $c(0,2) = ((0+2)^2+2)/2 = 3$.

All the operations involved in computing c have already been shown to be primitive recursive, thus we can conclude that c is primitive recursive, too. Note that c is usually referred to as to *Cantor's pairing function*.

Exercise 40. *Determine the functions* d_1 *and* d_2 *such that for all* $x, y \in \mathbb{N}$, *if* $z = c(x,y)$ *then* $x = d_1(z)$ *and* $y = d_2(z)$.

Exercise 41. *Show that for every fixed* $k \in \mathbb{N}$, $n > 2$, *there is a primitive recursive bijection* $c_k \colon \mathbb{N}^k \mapsto \mathbb{N}$.

Exercise 42. *Let* $\mathbb{N}^*$ *be the set of all finite sequences of natural numbers. Show that there is a primitive recursive bijection* $c_* \colon \mathbb{N}^* \mapsto \mathbb{N}$.

Furthermore, we can easily extend binary addition (i.e., the function α defined in Example 16) to finite sums by setting

$$\begin{aligned} \sum_{i=1}^{1} x_i &= I(x_1) \\ \sum_{i=1}^{n+1} x_i &= \alpha\left(x_{n+1}, \sum_{i=1}^{n} x_i\right) . \end{aligned}$$

Now, try it yourself.

Exercise 43. *Prove that*

(1) *every constant function,*

(2) $f(n) = 2^n$, *and*

(3) $d(n) = 2^{2^n}$

(4) $|m - n|$

(5) $\left[\frac{n}{m}\right]$ *(i.e., division with remainder)*

are primitive recursive.

11.3. General Recursive Functions

Next, we define the class of ***general recursive functions***.

Definition 37. *For all* $n \in \mathbb{N}^+$ *we define* $\mathcal{R}^n$ *to be the set of all functions* $f \in \mathcal{P}^n$ *such that* $f(x_1, \ldots, x_n)$ *is defined for all* $x_1, \ldots, x_n \in \mathbb{N}$. *Furthermore, we set* $\mathcal{R} = \bigcup_{n \in \mathbb{N}^+} \mathcal{R}^n$.

In other words, $\mathcal{R}$ is the set of all functions that are total and partial recursive. Now, we can show the following theorem.

Theorem 11.5. *Prim* $\subset \mathcal{R} \subset \mathcal{P}$.

Proof. Clearly $\mathsf{Z}, \mathsf{S}\,\mathsf{V} \in \mathcal{R}$. Furthermore, after a bit of reflection it should be obvious that any finite number of applications of Operations (2.1) through (2.5) results only in total functions. Hence, every primitive recursive function is general recursive, too. This shows *Prim* $\subseteq \mathcal{R}$. Also, $\mathcal{R} \subseteq \mathcal{P}$ is obvious by definition. So, it remains to show that the two inclusions are proper. This is done by the following claims.

Claim 1. $\mathcal{P} \setminus \mathcal{R} \neq \emptyset$.

By definition, $\mathsf{S} \in \mathcal{P}$ and using Operation (2.4) it is easy to see that $\delta(\mathfrak{n}) =_{\mathtt{df}} \mathsf{S}(\mathsf{S}(\mathfrak{n}))$ is in $\mathcal{P}$, too. Now, note that $\delta(\mathfrak{n}) = \mathfrak{n} + 2 > 1$ for all $\mathfrak{n} \in \mathbb{N}$.

Using Operation (2.1) we define $\tau(\mathfrak{x}, \mathfrak{y}) = \delta(\mathfrak{y})$, and thus $\tau \in \mathcal{P}$. Consequently,

$$\psi(\mathfrak{x}) = \mu[\tau(\mathfrak{x}, \mathfrak{y}) = 1]$$

is the nowhere defined function and hence $\psi \notin \mathcal{R}$. On the other hand, by construction $\psi \in \mathcal{P}$. Therefore, we get $\psi \in \mathcal{P} \setminus \mathcal{R}$, and Claim 1 is shown.

Claim 2. $\mathsf{R} \setminus \mathit{Prim} \neq \emptyset$.

Showing this claim is much more complicated. First, we define a function

$$\begin{aligned} \mathrm{a}(0, \mathfrak{m}) &= \mathfrak{m} + 1 \\ \mathrm{a}(\mathfrak{n} + 1, 0) &= \mathrm{a}(\mathfrak{n}, 1) \\ \mathrm{a}(\mathfrak{n} + 1, \mathfrak{m} + 1) &= \mathrm{a}(\mathfrak{n}, \mathrm{a}(\mathfrak{n} + 1, \mathfrak{m})) \end{aligned}$$

which is the so-called Ackermann-Péter function. Hilbert conjectured in 1926 that every total and computable function is also primitive recursive. This conjecture was disproved by Ackermann in 1928 and Péter simplified Ackermann's definition in 1955.

Now, we have to show that function a is *not* primitive recursive and that function a is general recursive. Both parts are not easy to prove. So, due to the lack of time, we must skip some parts. But before we start, let us confine ourselves that function a is *intuitively* computable. For doing this, consider the following fragment of pseudo-code implementing the function a as `peter`.

```
function peter(n, m)
      if n = 0
        return m + 1
      else if m = 0
        return peter(n - 1, 1)
      else
        return peter(n - 1, peter(n, m - 1))
```

Next, we sketch the proof that a cannot be primitive recursive. First, for every primitive recursive function ϕ, one defines a function f_ϕ as follows. Let k be the arity of ϕ; then we set

$$f_\phi(n) = \max\left\{\phi(x_1,\ldots,x_k) \mid \sum_{i=1}^{k} x_i \leqslant n\right\} .$$

Then, by using the inductive construction of the class *Prim* one can show by structural induction that for every primitive recursive function ϕ there is a number $n_\phi \in \mathbb{N}$ such that

$$f_\phi(n) < a(n_\phi, n) \text{ for all } n \geqslant n_\phi .$$

Intuitively, the latter statement shows that the Ackermann-Péter function grows faster than every primitive recursive function.

The rest is then easy. Suppose $a \in \mathit{Prim}$. Then, taking into account that the identity function I is primitive recursive, one directly sees by application of Operation (2.4) that

$$\kappa(n) = a(I(n), I(n))$$

is primitive recursive, too. Now, for κ there is a number $n_\kappa \in \mathbb{N}$ such that

$$f_\kappa(n) < a(n_\kappa, n) \text{ for all } n \geqslant n_\kappa .$$

But now,

$$\kappa(n_\kappa) \leqslant f_\kappa(n_\kappa) < a(n_\kappa, n_\kappa) = \kappa(n_\kappa) ,$$

a contradiction.

For the second part, one has to prove that $a \in \mathcal{R}$ which mainly means to provide a construction to express the function a using the Operations (2.1) through (2.5) *and* the μ-operator. We refer the interested reader to Hermes [2]. ∎

Exercise 44. *Compute* $a(1, m)$, $a(2, m)$, $a(3, m)$, *and* $a(4, m)$ *for* $m = 0, 1, 2$.

References

[1] K. Gödel (1931), Über formal unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme, *Monatshefte Mathematik Physik* **38**, 173 – 198.

[2] H. Hermes (1971), Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit. Springer, Berlin, Heidelberg, New York.

Lecture 12: Turing Machines

After having dealt with partial recursive functions, we turn our attention to Turing machines introduced by Alan Turing [2]. His idea was to formalize the notion of "intuitively computable" functions by using the four properties of an algorithm which we have stated at the beginning of Lecture 11. Starting from these properties, he observed that the primitive operations could be reduced to a level such that a machine can execute the whole algorithm. For the sake of simplicity, here we consider one-tape Turing machines.

12.1. One-tape Turing Machines

A one-tape Turing machine consists of an infinite tape which is divided into cells. Each cell can contain exactly one of the tape-symbols. Initially, we assume that all cells of the tape contain the symbol $*$ except those in which the actual input has been written. Moreover, we enumerate the tape cells as shown in Figure 12.1.

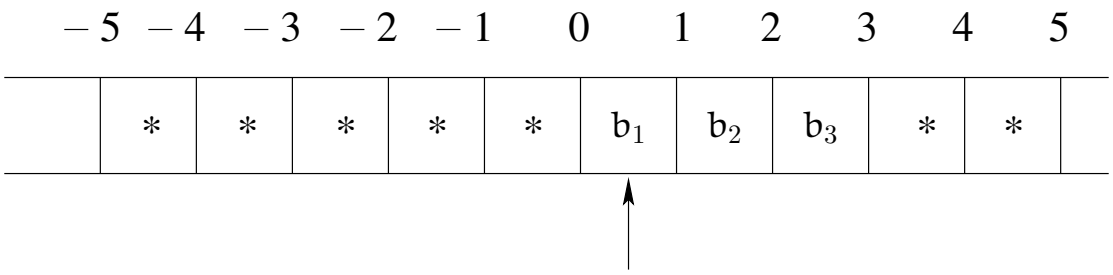

Figure 12.1: The tape of a Turing machine with input $\mathsf{b}_1\mathsf{b}_2\mathsf{b}_3$.

Furthermore, the Turing machine possesses a read-write head. This head can observe one cell at a time. Additionally, the machine has a finite number of states it can be in and a set of instructions it can execute. Initially, it is always in the start state z_s and the head is observing the leftmost symbol of the input, i.e., the cell 0. We indicate the position of the head by an arrow pointing to it.

Then, the machine works as follows. When in state z and reading tape symbol b, it writes tape symbol b' into the observed cell, changes its state to z' and moves the head either to the left (denoted by L) or to the right (denoted by R) or does not move the head (denoted by N) provided $(\mathsf{z}, \mathsf{b}, \mathsf{b}', \mathsf{m}, \mathsf{z}')$ is in the instruction set of the Turing machine, where $\mathsf{m} \in \{\mathsf{L}, \mathsf{N}, \mathsf{R}\}$. The execution of one instruction is called ***step***. When the machine reaches the *distinguished* state z_f (the *final* state), it stops. Thus, formally, we can define a Turing machine as follows.

Definition 38. $\mathsf{M} = [\mathsf{B}, \mathsf{Z}, \mathsf{A}]$ *is called* ***deterministic one-tape Turing machine*** *if* B, Z, A *are non-empty finite sets such that* $\mathsf{B} \cap \mathsf{Z} = \emptyset$ *and*

(1) $\mathrm{card}(B) \geqslant 2$ ($B = \{*, |, \ldots\}$) (*tape-symbols*),

(2) $\mathrm{card}(Z) \geqslant 2$ ($Z = \{z_s, z_f, \ldots\}$) (*set of states*),

(3) $A \subseteq Z\backslash\{z_f\}\times B\times B\times\{L, N, R\}\times Z$ (*instruction set*), *where for every* $z \in Z\backslash\{z_f\}$ *and every* $b \in B$ *there is precisely one 5-tuple* $(z, b, \cdot, \cdot, \cdot)$.

Often, we represent the instruction set A in a table, e.g.,

	$*$	$\|$	b_2	$\ldots$	b_n
z_s	$b'Nz_3$				
z_1	$\cdot$				
$\cdot$	$\cdot$				
$\cdot$	$\cdot$				
$\cdot$	$\cdot$				
z_n	$\cdot$				

A Turing table

If the instruction set is small, it often convenient to write $zb \rightarrow b'Hz'$, where $H \in \{L, N, R\}$ instead of (z, b, b', H, z'). Also, we often refer to the instruction set of a Turing machine M as to the ***program*** of M.

12.2. Turing Computations

Next, we have to explain how a Turing machine is computing a function. Our primary concern are functions from $\mathbb{N}^n$ to $\mathbb{N}$, i.e., $f\colon \mathbb{N}^n \mapsto \mathbb{N}$. Therefore, the inputs are tuples $(x_1, \ldots, x_n) \in \mathbb{N}^n$. We shall reserve the special tape symbol # to separate x_i from x_{i+1}. Moreover, for the sake of simplicity, in the following we shall assume that numbers are unary encoded, e.g., number 0 is represented by $*$, number 1 by $|$, number 2 by $||$, number 3 by $|||$, a.s.o. Note that this convention is no restriction as long as we do not consider the *complexity* of a Turing computation.

Furthermore, it is convenient to introduce the following notations. Let $f\colon \mathbb{N}^n \mapsto \mathbb{N}$ be any function. If the value $f(x_1, \ldots, x_n)$ is not defined for a tuple $(x_1, \ldots, x_n) \in \mathbb{N}^n$ then we write $f(x_1, \ldots, x_n)\uparrow$. If $f(x_1, \ldots, x_n)$ is defined then we write $f(x_1, \ldots, x_n)\downarrow$.

Definition 39. *Let* M *be any Turing machine, let* $n \in \mathbb{N}^+$ *and let* $f\colon \mathbb{N}^n \mapsto \mathbb{N}$ *be any function. We say that* M ***computes*** *the function* f *if for all* $(x_1, \ldots, x_n) \in \mathbb{N}^n$ *the following conditions are satisfied:*

(1) *If* $f(x_1, \ldots, x_n)\downarrow$ *and if* $x_1\#\ldots\#x_n$ *is written on the empty tape of* M (*beginning in cell* 0) *and* M *is started on the leftmost symbol of* $x_1\#\ldots\#x_n$ *in state* z_s*, then* M *stops after having executed finitely many steps in state* z_f*. Moreover, if* $f(x_1, \ldots, x_n) = 0$*, then the symbol observed by* M *in state* z_f *is* $*$*. If* $f(x_1, \ldots, x_n) \neq 0$*, then the string beginning in the cell observed by* M *in state* z_f *(read from left to right) of consecutive* $|$ *denotes the results* (cf. Figure 12.2).

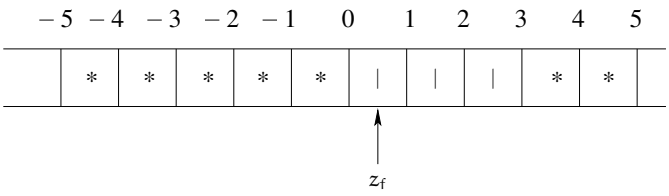

Figure 12.2: The tape of a Turing machine with result 3 (written as |||).

(2) *If* $f(x_1, \ldots, x_n) \uparrow$ *and if* $x_1 \# \ldots \# x_n$ *is written on the empty tape of* M (*beginning in cell* 0) *and* M *is started on the leftmost symbol of* $x_1 \# \ldots \# x_n$ *in state* z_s *then* M *does not stop.*

By f^n_M *we denote the function from* $\mathbb{N}^n$ *to* $\mathbb{N}$ *computed by Turing machine* M.

Definition 40. *Let* $n \in \mathbb{N}^+$ *and let* $f\colon \mathbb{N}^n \mapsto \mathbb{N}$ *be any function.* f *is said to be* ***Turing computable*** *if there exists a Turing machine* M *such that* $f^n_M = f$. *Furthermore, we set*

$$\begin{aligned} \mathcal{T}^n &= \text{set of all } n\text{-ary Turing computable functions.} \\ \mathcal{T} &= \bigcup_{n \geqslant 1} \mathcal{T}^n = \text{set of all Turing computable functions.} \end{aligned}$$

Furthermore, as usual we use $dom(f)$ to denote the domain of function f, and $range(f)$ to denote the range of function f.

Now, it is only natural to ask which functions are Turing computable. The answer is provided by the following theorem.

Theorem 12.1. *The class of Turing computable functions is equal to the class of partial recursive functions, i.e.,* $\mathcal{T} = \mathcal{P}$.

Proof. For showing $\mathcal{P} \subseteq \mathcal{T}$ it suffices to prove the following:

(1) The functions Z, S and V are Turing computable.

(2) The class of Turing computable functions is closed under the Operations (2.1) through (2.6) defined in Lecture 11.

A Turing machine computing the constant zero function can be easily defined as follows. Let $M = [\{*, |\}, \{z_s, z_f\}, A]$, where A is the following set of instructions:

$$\begin{aligned} z_s| &\rightarrow |\, L z_f \\ z_s* &\rightarrow *\, N z_f\ . \end{aligned}$$

That is, if the input is not zero, then M move its head one position to the left and stops. By our definition of a Turing machine, then M observes in cell -1 a $*$, and thus its output is 0. If the input is zero, then M observes in cell 0 a $*$, leaves it unchanged, does not move its head and stops. Clearly, $f^1_M(x) = 0$ for all $n \in \mathbb{N}$.

The successor function S is computed by the following Turing machine $\mathsf{M} = \{*,\ |\}, \{z_s, z_f\}, A]$, where A is the following set of instructions:

$$\begin{array}{rcl} z_s| & \to & |\,\mathsf{L}z_s \\ z_s* & \to & |\,\mathsf{N}z_f\ . \end{array}$$

That is, the machine just adds a | to its input and stops. Thus, we have $f^1_{\mathsf{M}}(x) = \mathsf{S}(x)$ for all $n \in \mathbb{N}$.

Furthermore, the predecessor function is computed by $\mathsf{M} = \{*,\ |\}, \{z_s, z_f\}, A]$, where A is the following set of instructions:

$$\begin{array}{rcl} z_s| & \to & *\,\mathsf{R}z_f \\ z_s* & \to & *\,\mathsf{N}z_f\ . \end{array}$$

Now, the Turing machine either observes a | in cell 0 which it removes and then the head goes one cell to the right or it observes a $*$, and stops without moving its head. Consequently, $f^1_{\mathsf{M}}(x) = \mathsf{V}(x)$ for all $n \in \mathbb{N}$. This proves Part (1).

Next, we sketch the proof of Part (2). This is done in a series of claims.

Claim 1. (Introduction of fictitious variables)
Let $n \in \mathbb{N}^+$*; then we have: if* $\tau \in \mathcal{T}^n$ *and* $\psi(x_1, \ldots, x_n, x_{n+1}) = \tau(x_1, \ldots, x_n)$*, then* $\psi \in \mathcal{T}^{n+1}$.

Intuitively, it is clear that Claim 1 holds. In order to compute ψ, all we have to do is to remove x_{n+1} from the input tape, then moving the head back into the initial position and then we start the Turing program for τ. Consequently, $\psi \in \mathcal{T}^{n+1}$. We omit the details.

Claim 2. (Identifying variables)
Let $n \in \mathbb{N}^+$*; then we have: if* $\tau \in \mathcal{T}^{n+1}$ *and* $\psi(x_1, \ldots, x_n) = \tau(x_1, \ldots, x_n, x_n)$*, then* $\psi \in \mathcal{T}^n$.

For proving Claim 2, we only need a Turing program that copies the last variable (that is, x_n). Thus, the initial tape inscription

$$* \ * \ x_1 \# \ldots \# x_n \ * \ *$$

is transformed into

$$* \ * \ x_1 \# \ldots \# x_n \# x_n \ * \ *$$

and the head is moved back into its initial position and M is put into the initial state of the program computing τ. Now, we start the program for τ. Consequently, $\psi \in \mathcal{T}^n$. Again, we omit the details.

Claim 3. (Permuting variables)
Let $n \in \mathbb{N}^+,\ n \geqslant 2$ *and let* $i \in \{1, \ldots, n\}$*; then we have:*
if $\tau \in \mathcal{T}^n$ *and* $\psi(x_1, \ldots, x_i, x_{i+1}, \ldots, x_n) = \tau(x_1, \ldots, x_{i+1}, x_i, \ldots, x_n)$*, then* $\psi \in \mathcal{T}^n$.

Claim 3 can be shown *mutatis mutandis* as Claim 2, and we therefore omit its proof here.

Claim 4. (Composition)
Let $n \in \mathbb{N}$ *and* $m \in \mathbb{N}^+$. *Furthermore, let* $\tau \in \mathcal{T}^{n+1}$, *let* $\psi \in \mathcal{T}^m$ *and let* $\phi(x_1, \ldots, x_n, y_1, \ldots, y_m) = \tau(x_1, \ldots, x_n, \psi(y_1, \ldots, y_m))$. *Then* $\phi \in \mathcal{T}^{n+m}$.

The proof of Claim 4 is a bit more complicated. Clearly, the idea is to move the head to the right until it observes the first symbol of y_1. Then we could start the Turing program for ψ. If $\psi(y_1, \ldots, y_m) \uparrow$, then the machine for ϕ also diverges on input $x_1, \ldots, x_n, y_1, \ldots, y_m$. But if $\psi(y_1, \ldots, y_m) \downarrow$, our goal would be to obtain the new tape inscription

$$* \; * \; x_1 \# \ldots \# x_n \# \psi(y_1, \ldots, y_m) * \; *$$

and then to move the head to the left such that it observes the first symbol of x_1. This would allow us to start then the Turing program for τ. So, the difficulty we have to overcome is to ensure that the computation of $\psi(y_1, \ldots, y_m)$ does not overwrite the x_i.

Thus, we need the following lemma.

Lemma M^+. *For every Turing machine* M *there exists a Turing machine* M^+ *such that*

(1) $f^n_M = f^n_{M^+}$ *for all* n.

(2) M^+ *is never moving left to the initial cell observed when starting its computation.*

(3) *For all* $x_1, \ldots, x_n$: *If* $f^n_{M^+}(x_1, \ldots, x_n) \downarrow$, *then the computation stops with the head observing the same cell it has observed when the computation started and right to the result computed there are only* $*$ *on the tape.*

The idea to show Lemma M^+ is as follows. M^+ does the following.

- It moves the whole input one cell to the right,
- it marks the initial cell with a special symbol, say L,
- it marks the first cell right to the moved input with a special symbol, say E,
- it works then as M does except the following three exceptions:
  - If M^+ reaches the cell marked with E, say in state z, then it moves the marker E one cell to the right, moves the head then one position to the left and writes a $*$ into this cell (that is into the cell that originally contained E) and continues to simulate M in state z with the head at this position (that is, the cell originally containing E and now containing $*$).

- If M in state z enters the cell containing L, then the whole tape inscription between L and E (including E but excluding L) is moved one position to the right. In the cell rightmost to L a $*$ is written and M + continues to simulate M observing this cell.
- If M stops, then M^+ moves the whole result left such that the first symbol of the result is now located in the cell originally containing L. Furthermore, into all cells starting from the first position right to the moved result and ending in E a $*$ is written and then the head is moved back to the leftmost cell containing the result. Then M^+ also stops.

This proves Lemma M^+.

Having Lemma M^+, now Claim 4 follows as described above.

The remaining claims for primitive recursion and μ-recursion are left as an exercise. This shows $\mathcal{P} \subseteq \mathcal{T}$.

Finally, we have to show $\mathcal{T} \subseteq \mathcal{P}$. Let $\mathfrak{n} \in \mathbb{N}+$, let $\mathsf{f} \in \mathcal{T}^{\mathfrak{n}}$ and let M be any Turing machine computing f. We define the functions t (time) and r (result) as follows.

$$\mathsf{t}(\mathsf{x}_1, \ldots, \mathsf{x}_\mathfrak{n}, \mathsf{y}) = \begin{cases} 1\,, & \text{if } \mathsf{M} \text{ when started on } \mathsf{x}_1, \ldots, \mathsf{x}_\mathfrak{n}, \\ & \text{stops after having executed at most } \mathsf{y} \text{ steps} \\ 0\,, & \text{otherwise.} \end{cases}$$

$$\mathsf{r}(\mathsf{x}_1, \ldots, \mathsf{x}_\mathfrak{n}, \mathsf{y}) = \begin{cases} \mathsf{f}(\mathsf{x}_1, \ldots, \mathsf{x}_\mathfrak{n})\,, & \text{if } \mathsf{t}(\mathsf{x}_1, \ldots, \mathsf{x}_\mathfrak{n}, \mathsf{y}) = 1, \\ 0\,, & \text{otherwise.} \end{cases}$$

Now, one can show that $\mathsf{t},\ \mathsf{r} \in \mathit{Prim}$. Furthermore, Kleene showed the following normal form theorem.

Theorem 12.2. *For every* $\mathsf{f} \in \mathcal{T}^{\mathfrak{n}}$, $\mathfrak{n} \in \mathbb{N}^+$ *there are functions* $\mathsf{t},\ \mathsf{r} \in \mathit{Prim}$ *such that*

$$\mathsf{f}(\mathsf{x}_1, \ldots, \mathsf{x}_\mathfrak{n}) = \mathsf{r}(\mathsf{x}_1, \ldots, \mathsf{x}_\mathfrak{n}, \mu\mathsf{y}[\mathsf{t}(\mathsf{x}_1, \ldots, \mathsf{x}_\mathfrak{n}, \mathsf{y}) = 1]) \tag{12.1}$$

for all $\mathsf{x}_1, \ldots, \mathsf{x}_\mathfrak{n} \in \mathbb{N}^{\mathfrak{n}}$. We do not prove this theorem here, since a proof is beyond the scope of this course.

Assuming Kleene's normal form theorem, the inclusion $\mathcal{T} \subseteq \mathcal{P}$ follows from the primitive-recursiveness of t and r and Equation (12.1), since the latter one shows that one has to apply the μ-operator exactly ones (Operation (2.6)) and the resulting function is composed with r by using Operation (2.4). Consequently, $\mathsf{f} \in \mathcal{P}$ and the theorem follows. ∎

The latter theorem is of fundamental epistemological importance. Though we started from completely different perspectives, we finally arrived the same set of computable functions. Subsequently, different approaches have been proposed to formalize the notion of "intuitively computable" functions. These approaches comprise, among

others, Post algorithms, Markov algorithms, Random-access machines (abbr. RAM), and Church' λ-calculus.

As it turned out, all these formalizations define the same set of computable functions, i.e., the resulting class of functions is equal to the Turing computable functions. This led Church to his famous thesis.

Church's Thesis. *The class of the "intuitively computable" functions is equal to the class of Turing computable functions.*

Note that Church's Thesis can neither be proved nor disproved in formal way, since it contains the not defined term "intuitively computable." Whenever this term is defined in a mathematical precise way, ones gets a precisely defined class of functions for which one then could try to prove a theorem as we did above.

Furthermore, it should be noted that the notion of a Turing machine contains some idealization such as potentially unlimited time and and having access to unlimited quantities of cells on the Turing tape. On the one hand, this idealization will limit the usefulness of theorems showing that some function is Turing computable. On the other hand, if one can prove that a particular function is *not* Turing computable, then such a result is very strong.

12.3. The Universal Turing Machine

Within this subsection we are going to show that there is *one* Turing machine which can compute all partial recursive functions.

First, using our results concerning pairing functions, it is easy to see that we can encode any $\mathfrak{n}$-tuple of natural numbers into a natural number. Moreover, as we have seen, this encoding is even primitive recursive. Thus, in the following, we use $\mathcal{P}$ to denote the set of all partial recursive functions from $\mathbb{N}$ to $\mathbb{N}$.

Next, let us consider any partial recursive function $\psi(\mathfrak{i},\mathfrak{x})$, i.e., $\psi\colon\mathbb{N}^2\mapsto\mathbb{N}$. Thus, if we fix the first argument $\mathfrak{i}$, then we obtain a partial recursive function of one argument. Usually, one uses the notation from the λ calculus to specify the argument which is not fixed, i.e., we write $\lambda\mathfrak{x}\psi(\mathfrak{i},\mathfrak{x})$ to denote the partial recursive function of the argument $\mathfrak{x}$. It is also common to write just $\psi_{\mathfrak{i}}$ instead of $\lambda\mathfrak{x}\psi(\mathfrak{i},\mathfrak{x})$. Thus, we can visualize all functions of one argument computed by ψ as follows (cf. Figure 12.3).

For having an example, consider $\psi(\mathfrak{i},\mathfrak{x})=\mathfrak{i}\mathfrak{x}$; then e.g., $\psi_7(\mathfrak{x})=7\mathfrak{x}$.

Therefore, it is justified to call every function $\psi\in\mathcal{P}^2$ a ***numbering***.

Definition 41. *A numbering $\psi\in\mathcal{P}^2$ is said to be **universal** for $\mathcal{P}$ if*

$$\{\psi_{\mathfrak{i}}\mid \mathfrak{i}\in\mathbb{N}\}=\mathcal{P}\ .$$

Clearly, now the interesting question is whether or not a universal $\psi\in\mathcal{P}^2$ for $\mathcal{P}$ does exist. If there is a universal ψ for $\mathcal{P}$, then, by Theorem 12.1, we know that ψ is

$$
\begin{array}{llllllll}
\psi_0 & \psi(0,0) & \psi(0,1) & \psi(0,2) & \psi(0,3) & \psi(0,4) & \dots \\
\psi_1 & \psi(1,0) & \psi(1,1) & \psi(1,2) & \psi(1,3) & \psi(1,4) & \dots \\
\psi_2 & \psi(2,0) & \psi(2,1) & \psi(2,2) & \psi(2,3) & \psi(2,4) & \dots \\
\psi_3 & \psi(3,0) & \psi(3,1) & \psi(3,2) & \psi(3,3) & \psi(3,4) & \dots \\
\psi_4 & \psi(4,0) & \psi(4,1) & \psi(4,2) & \psi(4,3) & \psi(4,4) & \dots \\
\psi_5 & \psi(5,0) & \dots & & & & \\
\cdot & & & & & & \\
\cdot & & & & & & \\
\cdot & & & & & & \\
\psi_i & \dots & \dots & & & & \\
\dots & & & & & &
\end{array}
$$

Figure 12.3: A two dimensional array representing all ψ_i.

Turing computable, too. Therefore, we could interpret any Turing machine computing ψ as a ***universal Turing machine***. The following theorem establishes the existence of a universal ψ.

Theorem 12.3. *There exists a universal numbering $\psi \in \mathcal{P}^2$ for $\mathcal{P}$.*

Proof. (Sketch) The idea is easily explained. By Theorem 12.1 we know that for every $\tau \in \mathcal{P}$ there is Turing machine M such that $f^1_M = \tau$. Therefore, we aim to encode every Turing machine into a natural number. Thus, we need an injective general recursive function *cod* such that $cod(M) \in \mathbb{N}$. Furthermore, in order to make this idea work we also need a general recursive function *decod* such that

$$decod(cod(M)) = \text{Program of } M\ .$$

If the input i to *decod* is not a correct encoding of some Turing machine, then we set $decod(i) = 0$.

The universal function ψ is then described by a Turing machine U taking two arguments as input, i.e., i and x. When started as usual, it first computes $decod(i)$. If $decod(i) = 0$, then it computes the function Z (constant zero). Otherwise, it should simulate the program of the machine M returned by $decod(i)$.

For realizing this behavior, the following additional conditions must be met:

(1) U is not allowed to overwrite the program obtained from the computation of $decod(i)$,

(2) U must be realized by using only finitely many tape symbols and a finite set of states.

Next, we shortly explain how all our conditions can be realized. For the sake of better readability, in the following we always denote the tape symbols by b_i and the state sets always starts with $z_s, z_f, \dots$. Let

$$M = [\{b_1, \dots, b_m\}, \{z_s, z_f, z_1, \dots, z_k\}, A]$$

be given. Then we use the following coding (here we write $0^{\mathfrak{n}}$ to denote the string consisting of exactly $\mathfrak{n}$ zeros):

$$\begin{aligned}
cod(\mathsf{L}) &= 101 \\
cod(\mathsf{R}) &= 1001 \\
cod(\mathsf{N}) &= 10001 \\
cod(\mathsf{z}_s) &= 10^4 1 \\
cod(\mathsf{z}_f) &= 10^6 1 \\
cod(\mathsf{z}_\ell) &= 10^{2(\ell+3)} 1 \qquad \text{for all } \ell \in \{1, \ldots, \mathsf{k}\} \\
cod(\mathsf{b}_\ell) &= 10^{2(\ell+1)+1} 1 \qquad \text{for all } \ell \in \{1, \ldots, \mathfrak{m}\}
\end{aligned}$$

The instruction set is then encoded by concatenating the codings of its parts, that is

$$cod(\mathsf{z}\mathfrak{b} \ \rightarrow \ \mathfrak{b}'\mathsf{H}\mathsf{z}') = cod(\mathsf{z})cod(\mathfrak{b})cod(\mathfrak{b}')cod(\mathsf{H})cod(\mathsf{z}') \ .$$

For example, $\mathsf{z}_s\mathfrak{b}_1 \ \rightarrow \ \mathfrak{b}_2\mathsf{N}\mathsf{z}_1$ is then encoded as

$$10^4 1 10^5 1 10^7 1 1000 1 10^8 1$$

Now, we have $\mathfrak{m}$ tape symbols and $\mathsf{k}+2$ states. Thus, there must be $\mathfrak{m}(\mathsf{k}+1)$ many instructions $\mathrm{I}_1, \ldots, \mathrm{I}_{\mathfrak{m}(\mathsf{k}+1)}$ (cf. Definition 38) which we assume to be written down in canonical order. Consequently, we finally encode the program of M by concatenating the encodings of all these instructions, i.e.,

$$cod(\mathsf{M}) = cod(\mathrm{I}_1) \cdots cod(\mathrm{I}_{\mathfrak{m}(\mathsf{k}+1)}) \ .$$

This string is interpreted as a natural number written in binary.

Now, it easy to see that *cod* is injective, i.e., if $\mathsf{M} \neq \mathsf{M}'$ then $cod(\mathsf{M}) \neq cod(\mathsf{M}')$.

Furthermore, if we use the function *cod* as described above, then *decode* reduces to check that an admissible string is given. If it is, the program of M can be directly read from the string.

Finally, we have to describe how the simulation is done. First, we have to ensure that U is not destroying the program of M. This is essentially done as outlined in Lemma M^+. Thus, it remains to explain how the Condition (2) is realized. Clearly, U cannot memorize the actual state of M during simulation in its state set, since this would potentially require an unlimited number of states. But U can mark the actual state in which M is on its tape (e.g., by using bold letters).

In order to ensure that U is only using finitely many tape symbols, U is not using directly $\mathfrak{b}_\ell$ from M's tape alphabet but just $cod(\mathfrak{b}_\ell) = 10^{2(\ell+1)+1}1$. This requires just two tape symbols for the simulation. We omit the details.

The Turing machine U can thus be expressed as a partial recursive function $\psi \in \mathcal{P}^2$ via Theorem 12.1 ∎

Summarizing, we have constructed a Turing machine U that can simulate every Turing machine computing a function of one argument. Since we can encode any tuple of natural numbers into a natural number, we thus have a *universal Turing machine*.

Corollary 12.4. *There exists a universal Turing machine* U *for* $\mathcal{T}$.

We finish this lecture by shortly explaining how Turing machines can accept formal languages.

12.4. Accepting Languages

Let Σ denote any finite alphabet. Again, we use Σ^* to denote the free monoid over Σ and λ to denote the empty string. Note that $\lambda \neq *$.

Next, we define what does it mean that a Turing machine is accepting a language L.

Definition 42. *A language* $\mathsf{L} \subseteq \Sigma^*$ *is* ***accepted*** *by Turing machine* M *if for every string* $w \in \Sigma^*$ *the following conditions are satisfied.*

If w *is written on the empty tape of* M (*beginning in cell* 0) *and the Turing machine* M *is started on the leftmost symbol of* w *in state* z_s *then* M *stops after having executed finitely many steps in state* z_f. *Moreover,*

(1) *if* $w \in \mathsf{L}$ *then the cell observed by* M *in state* z_f *contains a* $|$.
In this case we also write $\mathsf{M}(w) = |$.

(2) *If* $w \notin \mathsf{L}$ *then the cell observed by* M *in state* z_f *contains a* $*$.
In this case we also write $\mathsf{M}(w) = *$.

Of course, in order to accept a language $\mathsf{L} \subseteq \Sigma^*$ by a Turing machine $\mathsf{M} = [\mathsf{B}, \mathsf{Z}, \mathsf{A}]$ we always have to assume that $\Sigma \subseteq \mathsf{B}$.

Moreover, for every Turing machine M we define

$$\mathsf{L}(\mathsf{M}) =_{\mathrm{df}} \{w \mid w \in \Sigma^* \ \wedge \ \mathsf{M}(w) = |\} \ ,$$

and we refer to $\mathsf{L}(\mathsf{M})$ as to the language accepted by M.

Example 1. Let $\Sigma = \{\mathfrak{a}\}$ and $\mathsf{L} = \Sigma^+$.

We set $\mathsf{B} = \{*, \mathfrak{a}, |\}$, $\mathsf{Z} = \{z_s, z_f\}$ and define A as follows.

$$\begin{array}{rcl} z_s * & \longrightarrow & | \, \mathsf{N} z_f \\ z_s \mathfrak{a} & \longrightarrow & | \, \mathsf{N} z_f \\ z_s | & \longrightarrow & | \, \mathsf{N}, z_s \end{array}$$

where $z\mathfrak{b} \longrightarrow \mathfrak{b}'\mathfrak{m}z'$ if and only if $(z, \mathfrak{b}, \mathfrak{b}', \mathfrak{m}, z') \in \mathsf{A}$. Note that we have included the instruction $z_s | \longrightarrow | \, \mathsf{N}, z_s$ only for the sake of completeness, since this is required by Definition 38. In the following we shall often omit instructions that cannot be executed.

Example 2. Let $\Sigma = \{a\}$ and $L = \emptyset$.

Again, we set $B = \{*, a, |\}$, $Z = \{z_s, z_f\}$. The desired Turing machine M is defined as follows.

$$\begin{aligned} z_s * &\longrightarrow *Nz_f \\ z_s a &\longrightarrow *Nz_f \end{aligned}$$

Let us finish with a more complicated example, i.e., the language of all palindromes over a two letter alphabet, i.e., let $\Sigma = \{a, b\}$ and $L_{pal} = \{w \mid w \in \Sigma^*,\ w = w^T\}$.

For formally presenting a Turing machine M accepting L_{pal} we set $B = \{*, |,\ a,\ b\}$, $Z = \{z_s, z_1, z_2, z_3, z_4, z_5, z_6, z_f\}$ and A as given by the following table.

	a	b	$*$	$\|$
z_s	$*Rz_1$	$*Rz_2$	$\|Nz_f$	$\|Nz_f$
z_1	aRz_1	bRz_1	$*Lz_3$	$\|Nz_f$
z_2	aRz_2	bRz_2	$*Lz_4$	$\|Nz_f$
z_3	$*Lz_5$	$*Nz_f$	$\|Nz_f$	$\|Nz_f$
z_4	$*Nz_f$	$*Lz_5$	$\|Nz_f$	$\|Nz_f$
z_5	aLz_6	bLz_6	$\|Nz_f$	$\|Nz_f$
z_6	aLz_6	bLz_6	$*Rz_s$	$\|Nz_f$

Instruction set of a Turing machine accepting L_{pal}

So, this machine remembers the actual leftmost and rightmost symbol, respectively. Then it is checking whether or not it is identical to the rightmost and leftmost symbol, respectively.

References

[1] A. CHURCH (1936), An unresolvable problem of elementary number theory, *Am. J. Math.* **58**, 345 – 365.

[2] A.M. TURING (1936/37), On computable numbers with an application to the Entscheidungsproblem, *Proc. London Math. Soc.* **42**, 230 – 265.

Lecture 13: Algorithmic Unsolvability

In the last lecture we have shown that there is a universal numbering $\psi \in \mathcal{P}^2$ for $\mathcal{P}$ (cf. Theorem 12.3). Furthermore, we have provided a Turing machine U which is universal for $\mathcal{T}$ (see Corollary 12.4). These results can be used to establish the first problem which which is not algorithmically solvable.

13.1. The Halting Problem

Next, we explain the problem. Our universal Turing machine U is computing our numbering $\psi \in \mathcal{P}^2$. Now, if we wish to know whether or not $\psi(\mathtt{i},\mathtt{x})$ is defined, we could run U on input $\mathtt{i},\mathtt{x}$. Then, two cases have to be distinguished. First, the computation of $\mathsf{U}(\mathtt{i},\mathtt{x})$ stops. Then we know that $\psi(\mathtt{i},\mathtt{x})$ is defined, i.e., $\psi(\mathtt{i},\mathtt{x})\downarrow$. Second, the computation of $\mathsf{U}(\mathtt{i},\mathtt{x})$ does *not* stop.

However, while everything is clear when the computation of $\mathsf{U}(\mathtt{i},\mathtt{x})$ stops, the situation is different if it has not yet stopped. Just by observing the Turing machine U on input $\mathtt{i},\mathtt{x}$ for a finite amount of time, the only thing we can tell for sure is that has not yet stopped. Of course, there is still a chance that it will stop later. But it is also possible that it will never stop. Compare this to the situation when you are working with a computer. You have started a program and it does not terminate. What should you do? If you kill the execution of the program, then maybe it would have terminated its execution within the next hour, and now everything is lost. But if you let it run, and check the next day, and it still did not stop, again you have no idea what is better, to wait or to kill the execution.

Thus, it would be very nice if we could construct an algorithm *deciding* whether or not the computation of $\mathsf{U}(\mathtt{i},\mathtt{x})$ will stop. So, we ask whether or not such an algorithm does exist. This problem is usually referred to as to the ***general halting problem***. This equivalent to asking whether or not the following function $\tilde{\mathsf{h}}\colon \mathsf{N}\times\mathbb{N}\mapsto\mathbb{N}$ is computable, where

$$\tilde{\mathsf{h}}(\mathtt{i},\mathtt{x}) \;=\; \begin{cases} 1\,, & \text{if } \psi(\mathtt{i},\mathtt{x})\downarrow \\ 0\,, & \text{if } \psi(\mathtt{i},\mathtt{x})\uparrow \end{cases}.$$

Clearly, $\tilde{\mathsf{h}}$ is total. Thus, we have to figure out whether or not $\tilde{\mathsf{h}}\in\mathcal{R}^2$.

For answering this question, we look at a restricted version of it, usually just called the ***halting problem***, i.e., we consider

$$\mathsf{h}(\mathtt{x}) \;=\; \begin{cases} 1\,, & \text{if } \psi(\mathtt{x},\mathtt{x})\downarrow \\ 0\,, & \text{if } \psi(\mathtt{x},\mathtt{x})\uparrow \end{cases}.$$

Thus, now we have to find out whether or not $\mathsf{h}\in\mathcal{R}$. The negative answer is provided by our next theorem. We shall even proof a stronger result which is independent on the particular choice of ψ.

Theorem 13.1. *Let $\psi \in \mathcal{P}^2$ be any function which is universal for $\mathcal{P}$. Then for*

$$h(x) = \begin{cases} 1\,, & \text{if } \psi(x,x)\downarrow \\ 0\,, & \text{if } \psi(x,x)\uparrow\,, \end{cases}$$

we always have $h \notin \mathcal{R}$.

Proof. The proof is done by diagonalization. Suppose the converse, i.e., $h \in \mathcal{R}$. Now, define a function $\overline{h}\colon \mathbb{N} \mapsto \mathbb{N}$ as follows.

$$\overline{h}(x) = \begin{cases} 1 \dot{-} \psi(x,x)\,, & \text{if } h(x) = 1 \\ 0\,, & \text{if } h(x) = 0\,. \end{cases}$$

Since by supposition $h \in \mathcal{R}$, we can directly conclude that $\overline{h} \in \mathcal{R}$, too, by using the same technique as in the proof of Theorem 11.4. Furthermore, $\overline{h} \in \mathcal{R}$ directly implies $\overline{h} \in \mathcal{P}$, since $\mathcal{R} \subseteq \mathcal{P}$. Since ψ is universal for $\mathcal{P}$, there must exist an $i \in \mathbb{N}$ such that $\overline{h} = \psi_i$. Consequently,

$$\psi_i(i) = \overline{h}(i)\,.$$

We distinguish the following cases.

Case 1. $\psi_i(i)\downarrow$.

Then, by the definition of the function h we directly get that $h(i) = 1$. Therefore, the definition of the function $\overline{h}$ directly implies

$$\overline{h}(i) = 1 \dot{-} \psi(i,i) = 1 \dot{-} \psi_i(i) \neq \psi_i(i)\,,$$

a contradiction to $\psi_i(i) = \overline{h}(i)$. Thus, Case 1 cannot happen.

Case 2. $\psi_i(i)\uparrow$.

Now, the definition of the function h directly implies that $h(i) = 0$. Therefore, the definition of the function $\overline{h}$ yields $\overline{h}(i) = 0$. So, we have

$$0 = \overline{h}(i) = \psi_i(i)\,,$$

a contradiction to $\psi_i(i)\uparrow$. Thus, Case 2 can not happen either.

Now, the only resolution is that our supposition $h \in \mathcal{R}$ must be wrong. Consequently, we get $h \notin \mathcal{R}$, and the theorem is proved. ∎

So, we have seen that the halting problem is *algorithmically unsolvable*. This directly allows the corollary that the general halting problem is algorithmically unsolvable, too.

Corollary 13.2. *Let $\psi \in \mathcal{P}^2$ be any function which is universal for $\mathcal{P}$. Then for*

$$\tilde{h}(i,x) = \begin{cases} 1\,, & \text{if } \psi(i,x)\downarrow \\ 0\,, & \text{if } \psi(i,x)\uparrow\,. \end{cases}$$

we always have $\tilde{h} \notin \mathcal{R}^2$.

Proof. Suppose the converse, i.e., $\tilde{h} \in \mathcal{R}^2$. Then, we directly see that $h(x) = \tilde{h}(x,x)$ and thus, $h \in \mathcal{R}$, too. But this is a contradiction to Theorem 13.1. Thus, the corollary follows. ∎

Next, we aim to attack some of the problems from formal language theory. But this is easier said than done. Therefore, we have to study another problem which turns out to be very helpful, i.e., Post's correspondence problem.

13.2. Post's Correspondence Problem

While we have provided a direct proof for the undecidability of the halting problem, in the following we shall use a different approach. Within this subsection we begin to *reduce* undecidable problems concerning Turing machines to "real" problems.

Let us first explain what is meant by *reduction*. Let A and B be any subsets of $\mathbb{N}$. Furthermore, let us assume that the characteristic function χ_A of A is general recursive. So, for every $x \in N$ we can compute $\chi_A(x)$. Thus, if $\chi_A(x) = 1$, then we know that $x \in A$ and if $\chi_A(x) = 0$, then $x \notin A$. But now we get the new problem to decide for every $x \in \mathbb{N}$ whether or not $x \in B$. Instead of starting from scratch, we also have the possibility to search for a function *red* having the following two properties:

(1) $red \in \mathcal{R}$,

(2) $x \in B$ if and only if $red(x) \in A$.

Property (1) ensures that *red* is computable and total. Now, given any $x \in \mathbb{N}$, we compute $red(x)$ and then we run the algorithm computing χ_A on input $red(x)$. By Property (2) we know that $x \in B$ if and only if $\chi_A(red(x)) = 1$ (cf. Figure 13.1). If B is reducible to A via *red* then we also write $B \leqslant_{red} A$.

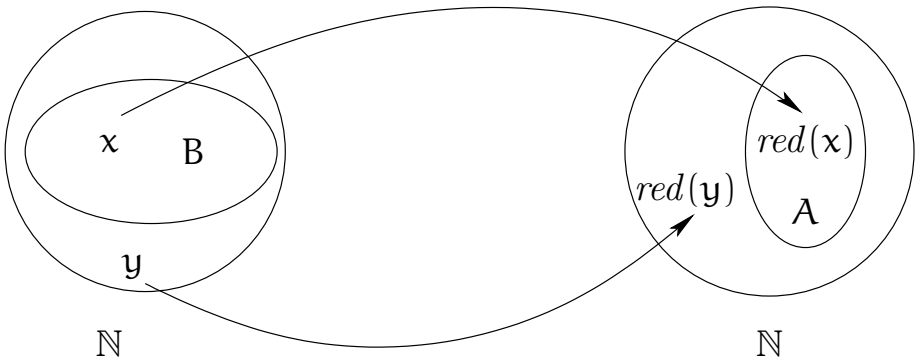

Figure 13.1: Reducing B to A.

Reductions are also very important as a proof technique. Let $\psi \in \mathcal{P}^2$ be any numbering universal for $\mathcal{P}$. As shown above, then the set $K = \{i \mid i \in \mathbb{N},\ \psi_i(i)\downarrow\}$ is undecidable, i.e., $\chi_K \notin \mathcal{R}$. We refer to K as to the ***halting set***. Now, let a set $P \subseteq \mathbb{N}$ be given. If we can find a reduction function $red \in \mathcal{R}$ such that $K \leqslant_{red} P$, then P must be undecidable, too. For seeing this, suppose the converse, i.e., $\chi_P \in \mathcal{R}$. Given any $x \in \mathbb{N}$, we can compute $\chi_P(red(x))$. By Property (2) of a reduction function we then have $x \in K$ if and only if $\chi_P(red(x)) = 1$. Thus, K would be decidable, a contradiction.

We continue with Post's correspondence problem. Though Post's correspondence problem may seem rather abstract to you, it has the advantage to be defined over strings. In the following we shall abbreviate Post's correspondence problem by PCP. Next, we formally define what a PCP is.

Definition 43. *A quadruple* $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ *is called* ***PCP*** *if*

(1) $\Sigma \neq \emptyset$ *is a finite alphabet,*

(2) $n \in \mathbb{N}^+$

(3) $\mathfrak{P}, \mathfrak{Q} \in (\Sigma^+)^n$, *i.e.,*

$$\begin{aligned} \mathfrak{P} &= [p_1, \ldots, p_n] \\ \mathfrak{Q} &= [q_1, \ldots, q_n] \ , \quad \text{where } p_i, q_i \in \Sigma^+ \ . \end{aligned}$$

Definition 44. *Let* $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ *be any PCP.* $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ *is said to be* ***solvable*** *if there is a finite sequence* $i_1, i_2 \ldots, i_k$ *of natural numbers such that*

(1) $i_j \leqslant n$ *for all* $1 \leqslant j \leqslant k$,

(2) $p_{i_1} p_{i_2} \cdots p_{i_k} = q_{i_1} q_{i_2} \cdots q_{i_k}$.

$[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ *is said to be* ***unsolvable*** *if it is not solvable.*

Next, we provide an example for a solvable and an unsolvable PCP, respectively.

Example 18. Consider the PCP $[\{a, b\}, 3, [a^2, b^2, ab^2],\ [a^2b, ba, b]]$. This PCP is solvable, since

$$p_1 p_2 p_1 p_3 = a^2 b^2 a^2 a b^2 = a^2 b b a a^2 b b = q_1 q_2 q_1 q_3 \ .$$

Example 19. Consider the PCP $[\{a\}, 2, [a^3, a^4],\ [a^2, a^3]]$. This PCP is unsolvable.

Furthermore, it is necessary to have the following definition.

Definition 45. *Let* $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ *be any PCP.* $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ *is said to be* ***1-solvable*** *if it is solvable and*

$$p_1 p_{i_2} \cdots p_{i_k} = q_1 q_{i_2} \cdots q_{i_k} \ ,$$

that is, if $i_1 = 1$.

Now we are ready to state the main theorem of this subsection, i.e., the property of a PCP to be solvable is undecidable.

Theorem 13.3. *There does not exist any Turing machine* M *such that*

$$M(p_1 \# p_2 \# \cdots \# p_n \# q_1 \# q_2 \# \cdots \# q_n) = \begin{cases} 1 \ , & \text{if } [\Sigma, n, \mathfrak{P}, \mathfrak{Q}] \text{ is solvable,} \\ 0 \ , & \text{if } [\Sigma, n, \mathfrak{P}, \mathfrak{Q}] \text{ is unsolvable,} \end{cases}$$

for every PCP $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$.

Proof. The proof idea is as follows. We shall show that deciding the solvability of any PCP is at least as hard as deciding the halting problem.

This goal is achieved in two steps, i.e., we show:

Step 1. If we could decide the solvability of any PCP, then we can also decide the 1-solvability of any PCP.

Step 2. If we could decide the 1-solvability of any PCP, then we can also decide the halting problem.

For proving Step 1, we show that for every PCP $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ we can effectively construct a PCP $[\Sigma \cup \{A, E\}, n+2, \mathfrak{P}', \mathfrak{Q}']$ such that

$$[\Sigma, n, \mathfrak{P}, \mathfrak{Q}] \text{ is 1-solvable iff } [\Sigma \cup \{A, E\}, n+2, \mathfrak{P}', \mathfrak{Q}'] \text{ is solvable .} \tag{13.1}$$

Here, by *effective* we mean that there is an algorithm (a Turing machine) which on input any PCP $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ outputs $[\Sigma \cup \{A, E\}, n+2, \mathfrak{P}', \mathfrak{Q}']$.

We define the following two homomorphisms h_R and h_L. Let A, E be any fixed symbols such that A, $E \notin \Sigma$. Then we set for all $x \in \Sigma$

$$\begin{aligned} h_R(x) &= xA \ , \\ h_L(x) &= Ax \ . \end{aligned}$$

Next, let $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ be given as input, where $\mathfrak{P} = [p_1, \ldots, p_n]$ and $\mathfrak{Q} = [q_1, \ldots, q_n]$. Then we compute

$$\begin{array}{rclcrcl} p'_1 &=& Ah_R(p_1) & & q'_1 &=& h_L(q_1) \\ p'_{i+1} &=& h_R(p_i) & 1 \leqslant i \leqslant n & q'_{i+1} &=& h_L(q_i) \\ p'_{n+2} &=& E & & q'_{n+2} &=& AE \end{array}$$

For the sake of illustration, let us exemplify the construction. Let the PCP $[\{a, b\}, 2, [a^2, b],\ [a, ab]]$ be given. It is obviously 1-solvable, since $p_1p_2 = q_1q_2$. Now, we compute

$$\begin{array}{ll} p'_1 = AaAaA & q'_1 = Aa \\ p'_2 = aAaA & q'_2 = Aa \\ p'_3 = bA & q'_3 = AaAb \\ p'_4 = E & q'_4 = AE \end{array}$$

Now, it is easy to see that $[\{a, b, A, E\}, 4, \mathfrak{P}', \mathfrak{Q}']$ is solvable, since

$$\begin{aligned} p'_1p'_3p'_4 &= AaAaAbAE \\ &= q'_1q'_3q'_4 \ . \end{aligned}$$

For the general case, we show the following claim stating that if $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ is 1-solvable then $[\Sigma \cup \{A, E\}, n+2, \mathfrak{P}', \mathfrak{Q}']$ is solvable.

Claim 1. *Let* $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ *be any PCP and let* $[\Sigma \cup \{A, E\}, n+2, \mathfrak{P}', \mathfrak{Q}']$ *be the PCP constructed as described above. If there is a finite sequence* $1, i_1, \ldots, i_r$ *such that* $p_1 p_{i_1} \cdots p_{i_r} = q_1 q_{i_1} \cdots q_{i_r}$ *then* $p'_1 p'_{i_1+1} \cdots p'_{i_r+1} p'_{n+2} = q'_1 q'_{i_1+1} \cdots q'_{i_r+1} q'_{n+2}$.

The claim is proved by calculating the strings involved. For seeing how the proof works, let us start with the simplest case, i.e., $p_1 = q_1$.

Let $p_1 = x_1 \cdots x_k$. By construction we get

$$\begin{aligned} p'_1 &= Ax_1Ax_2A \cdots Ax_kA & \qquad q'_1 &= Ax_1Ax_2A \cdots Ax_k \\ p'_2 &= x_1Ax_2A \cdots Ax_kA & \qquad q'_2 &= Ax_1Ax_2A \cdots Ax_k \\ p'_{n+2} &= E & \qquad q'_{n+2} &= AE \end{aligned}$$

So, we see that p'_2 and q'_2 are almost equal, that is they are equal except the missing leading A in p'_2 and the A at the end of q_2. Therefore, we replace p'_2 and q'_2 by p'_1 and q'_1, respectively. This solves the problem of the missing leading A. Now, q'_{n+2} gives the missing A on the rightmost part of q'_1 plus an E and this E is obtained from p'_{n+2} which concatenated to p'_2. Hence, $p'_1 p'_{n+2} = q'_1 q'_{n+2}$. Thus the assertion follows.

Now, for the general case the same idea works. Assume that

$$p_1 p_{i_1} \cdots p_{i_r} = q_1 q_{i_1} \cdots q_{i_r} \ . \tag{13.2}$$

We have to show that

$$p'_1 p'_{i_1+1} \cdots p'_{i_r+1} p'_{n+2} = q'_1 q'_{i_1+1} \cdots q'_{i_r+1} q'_{n+2}$$

Since h_R and h_L are homomorphisms, by construction we get

$$\begin{aligned} h_R(p_1 p_{i_1} \cdots p_{i_r}) &= h_R(p_1) h_R(p_{i_1}) \cdots h_R(p_{i_r}) \\ &= p'_2 p'_{i_1+1} \cdots p'_{i_r+1} \ , \text{ and} \\ h_L(q_1 q_{i_1} \cdots q_{i_r}) &= h_L(q_1) h_L(q_{i_1}) \cdots h_L(q_{i_r}) \\ &= q'_2 q'_{i_1+1} \cdots q'_{i_r+1} \ . \end{aligned}$$

Thus by (13.2), applying h_R to $p_1 p_{i_1} \cdots p_{i_r}$ and h_L to $q_1 q_{i_1} \cdots q_{i_r}$ gives again almost the same strings, except the same problems mentioned above. So we again replace p'_2 and q'_2 by p'_1 and q'_1, respectively, solving the problem of the leading A in $h_R(p_1 p_{i_1} \cdots p_{i_r})$. Appending p'_{n+2} and q'_{n+2} to $p'_1 p'_{i_1+1} \cdots p'_{i_r+1}$ and $q'_1 q'_{i_1+1} \cdots q'_{i_r+1}$, respectively, gives on the left hand side an E at the end of the string and an AE on the right hand side at the end of the string. Hence,

$$p'_1 p'_{i_1+1} \cdots p'_{i_r+1} p'_{n+2} = q'_1 q'_{i_1+1} \cdots q'_{i_r+1} q'_{n+2} \ .$$

This proves the claim.

Thus, we have shown the necessity of (13.1). For proving the sufficiency of (13.1), assume that $[\Sigma \cup \{A, E\}, n+2, \mathfrak{P}', \mathfrak{Q}']$ is solvable. We have to show that $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ is 1-solvable.

By assumption, there is a finite sequence $\mathfrak{i}_1, \ldots, \mathfrak{i}_r$ such that

$$\mathfrak{p}'_{\mathfrak{i}_1} \cdots \mathfrak{p}'_{\mathfrak{i}_r} = \mathfrak{q}'_{\mathfrak{i}_1} \cdots \mathfrak{q}'_{\mathfrak{i}_r} \ .$$

Now, all strings $\mathfrak{q}'_{\mathfrak{i}}$ start with an A. Since $\mathfrak{p}'_1$ is the only string among all $\mathfrak{p}'_{\mathfrak{i}}$ which start with A, we directly see that $\mathfrak{i}_1 = 1$.

Furthermore, since all $\mathfrak{p}'_{\mathfrak{i}}$ end with an A and none of the $\mathfrak{q}'_{\mathfrak{i}}$ does end with A, we can directly conclude that $\mathfrak{i}_r = n + 2$. Thus, by deleting all A and E we directly get the desired solution, i.e.,

$$\mathfrak{p}_1\mathfrak{p}_{\mathfrak{i}_2-1} \cdots \mathfrak{p}_{\mathfrak{i}_{r-1}} = \mathfrak{q}_1\mathfrak{q}_{\mathfrak{i}_2-1} \cdots \mathfrak{q}_{\mathfrak{i}_{r-1}} \ .$$

Note that deleting all A and E corresponds to applying the erasing homomorphism h_e defined as $h_e(x) = x$ for all $x \in \Sigma$ and $h_e(\mathsf{A}) = h_e(\mathsf{E}) = \lambda$. Therefore, we have provided a 1-solution of $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$. This completes the proof of Step 1.

We continue with Step 2. The idea is to transform the instruction set of any Turing machine M and its actual input x into a PCP such that the PCP is 1-solvable if and only if M stops its computation on input x.

For realizing this idea, it is technically advantageous to assume a normalization of Turing machines such as we did when proving Lemma M^+ in Lecture 12. Therefore, we need the following Lemma M^*.

Lemma M^*. *For every Turing machine* M *there exists a Turing machine* M^* *such that*

(1) $f_M = f_{M^*}$, *i.e.,* M *and* M^* *compute the same function. for all* n.

(2) M^* *is never moving left to the initial cell observed when starting its computation.*

(3) *In each step of the computation of* M^* *the head is either moving one cell to the left or one cell to the right.*

(4) M^* *is not allowed to write the symbol* $*$ *on its tape (but of course, it is allowed to read it).*

Proof. Property (2) of Lemma M^* can be shown in the same way as we proved it in the demonstration of Lemma M^+. But instead of L, we know use another symbol, e.g., $\boldsymbol{L}$.

For showing Property (3), we have to deal with all instructions of the form $zb \to b'Nz'$ of M's instruction set. So, if the instruction $zb \to b'Nz'$ belongs to M's instruction set, then we introduce a new state $z_{z'}$ to the state set of M^* and replace $zb \to b'Nz'$ by

$$\begin{aligned} zb &\to b'Lz_{z'} \\ z_{z'}b &\to bRz' \text{ for all tape symbols } b \in B \ . \end{aligned}$$

That is, now the head first moves left and M^* changes its state to $z_{z'}$. And then it moves the head just right and switches it state back to z'. Thus, it behaves exactly as M does except the two additional moves of the head.

Finally, we have to realize Property (4). Without loss of generality we can assume that $? \notin B$. Then instead of writing $*$ on its tape, M^* writes the new symbol ? on its tape. That is, we replace all instructions $zb \to *Hz'$, where $H \in \{L, R\}$, of the so far transformed instruction set by $zb \to ?Hz'$, where $H \in \{L, R\}$. Additionally, we have to duplicate all instructions of the form $z* \to \ldots$ by $z? \to \ldots$. That is, when reading a ?, M^* works as M would do when reading a $*$.

Now, it is easy to see that Property (1) is also satisfied. We omit details. This proves Lemma M^*.

Introducing the ? in Lemma M^* has the advantage that M^*, when reading a $*$ knows that it is visiting a cell it has not visited before.

In the same way as we did in Lecture 12, one can show that there is a universal Turing machine U^* which can simulate all Turing machines M^*. Since the machines M^* have special properties, it is conceivable that the halting problem for U^* is decidable. However, using the same technique as we did above, one can show that the general halting problem and the halting problem for U^* are undecidable, too.

This is a good point to recall that we have introduced instantaneous descriptions for pushdown automata in order to describe their computations. For describing computations of Turing machines we use *configurations* defined as follows. Let M^* be a Turing machine.

A string $b_1 b_2 \cdots b_{i-1} z b_i \cdots b_n$ is said to be a ***configuration*** of M^* if

1. z is the actual state of M^*,
2. the head is observing cell i,
3. $b_1 b_2 \cdots b_n$ is the portion of the tape between the leftmost and rightmost $*$.

So, the initial configuration is $z_s x$, where x represents the input. If we use an unary representation for the inputs, then, in case $x = 0$, we omit the $*$. We write $c_i \vdash c_{i+1}$ provided configuration c_{i+1} is reached from configuration c_i in one step of computation performed by M^*.

Example 20. Let $M^* = [\{*, |\}, \{z_s, z_f, z_1\}, A]$, where A is defined as follows.

$$\begin{aligned} z_s| &\to |Rz_1 \\ z_1| &\to |Rz_1 \\ z_s* &\to |Rz_f \\ z_1* &\to |Lz_f \end{aligned}$$

Then, on input | we get the following sequence of configurations.

$$z_s\,|\vdash|\,z_1\vdash z_f\,||\tag{13.3}$$

Now, we are ready to perform the final step of our proof, i.e., reducing the halting problem to PCP. Given any Turing machine $M^*=[B,Z,A]$ and input x represented in unary notation, we construct a PCP as follows. Note that, by Lemma M^* we have that if $zb\ \to\ z'Hb'\in A$, where $H\in\{L,R\}$, then $b'\in B\setminus\{*\}$. In the table below, we always assume that $b,\ b',\ b''\in B\setminus\{*\}$. Also, we assume that $\#\notin B$. That is, here we use # as a separator (of configurations). Thus, we assume that $\#\notin B$.

	$\mathfrak{P}$		$\mathfrak{Q}$	if
p_1	#	q_1	$\#z_s\vert\cdots\vert\#$	$x=\vert\cdots\vert$
	zb		$b'z$	$zb\ \to\ b'Rz'\in A$
	bzb'		$z'bb''$	$zb'\ \to\ b''Lz'\in A$, $b\in B\setminus\{*\}$
	$z\#$		$bz'\#$	$z*\ \to\ bRz'\in A$
p_i	$bz\#$	q_i	$zbb'\#$	$z*\ \to\ b'Lz'\in A$, $b\in B\setminus\{*\}$
	b		b	$b\in B\setminus\{*\}$
	#		#	always
	z_fb		z_f	$b\in B\setminus\{*\}$
	bz_f		z_f	$b\in B\setminus\{*\}$
	$z_f\#\#$		#	always

Figure 13.2: The PCP corresponding to M^* on input x.

Before continuing the proof, let us return to Example 20. It is easy to see that $f_{M^*}(0)=0$ and $f_{M^*}(n)=2$ for all $n\in\mathbb{N}^+$. Thus, this machine M^* stops on every input.

Then for any $x\in\mathbb{N}$, we get the following PCP, where again x is the input to M^*.

	$\mathfrak{P}$		$\mathfrak{Q}$	if
p_1	#	q_1	$\#z_s\vert\cdots\vert\#$	$x=\vert\cdots\vert$
p_2	$z_s\vert$	q_2	$\vert z_1$	$z_s\vert\ \to\ \vert Rz_1\in A$
p_3	$z_1\vert$	q_3	$\vert z_1$	$z_1\vert\ \to\ \vert Rz_1\in A$
p_4	$z_s\#$	q_4	$\vert z_f\#$	$z_s*\vert Rz_f\in A$
p_5	$\vert z\#$	q_5	$z_f\vert\vert\#$	$z_1*\ \to\ \vert Lz_f\in A$
p_6	$\vert$	q_6	$\vert$	$\vert\in B\setminus\{*\}$
p_7	#	q_7	#	always
p_8	$z_f\vert$	q_8	z_f	$\vert\in B\setminus\{*\}$
p_9	$\vert z_f$	q_9	z_f	$\vert\in B\setminus\{*\}$
p_{10}	$z_f\#\#$	q_{10}	#	always

Looking at the table above, we see that $|p_i|\leqslant|q_i$ for all $i\in\{1,\ldots,7\}$. In order to find a 1-solution of the PCP we concatenate p's and q's, respectively, that correspond to the computation of M^* when started in z_s on input x. Thus, initially, the concatenation of the p's will result in a string that is shorter than string obtained when

concatenating the corresponding $\mathfrak{q}$'s. This length difference can be resolved if and only if the computation terminates. Then, one can use the strings one the last group in the table to resolve the length difference.

In order to see how it works, we consider input $|$. Thus, $\mathfrak{p}_1 = \#$ and $\mathfrak{q}_1 = \#\mathfrak{z}_s|\#$. Therefore, in order to have any chance to find a 1-solution of the PCP, we must use $\mathfrak{p}_2$ next. Consequently, we have to use $\mathfrak{q}_2$, too, and this gives

$$\begin{array}{cc} \underbrace{\#}_{\mathfrak{p}_1} & \underbrace{\mathfrak{z}_s|}_{\mathfrak{p}_2} \\ \underbrace{\#\mathfrak{z}_s|\#}_{\mathfrak{q}_1} & \underbrace{|\mathfrak{z}_1}_{\mathfrak{q}_2} \end{array}$$

Next, in the sequence of the $\mathfrak{p}$'s, we need a $\#$. We could use $\mathfrak{p}_1$ or $\mathfrak{p}_7$. So, we try $\mathfrak{p}_7$, and hence, we have to use $\mathfrak{q}_7$, too. Then we must use $\mathfrak{p}_5$ and $\mathfrak{q}_5$. Thus, we have

$$\begin{array}{cccc} \underbrace{\#}_{\mathfrak{p}_1} & \underbrace{\mathfrak{z}_s|}_{\mathfrak{p}_2} & \underbrace{\#}_{\mathfrak{p}_7} & \underbrace{|\mathfrak{z}_1\#}_{\mathfrak{p}_5} \\ \underbrace{\#\mathfrak{z}_s|\#}_{\mathfrak{q}_1} & \underbrace{|\mathfrak{z}_1}_{\mathfrak{q}_2} & \underbrace{\#}_{\mathfrak{q}_7} & \underbrace{\mathfrak{z}_f||\#}_{\mathfrak{q}_5} \end{array}$$

The idea of our construction should be clear by now. Before reaching the final state, the sequence of $\mathfrak{q}$'s describes successive configurations of the computation of M^* started on input $\mathfrak{x}$. That is, if we have $\#\alpha_i\#\alpha_{i+1}$ then α_i and α_{i+1} are configurations of M^* and $\alpha_i \vdash \alpha_{i+1}$ holds (see (13.3)). Also, we see that the sequence of $\mathfrak{q}'s$ is always one configuration ahead to the sequence generated by the $\mathfrak{p}$'s as long as M^* did not reach its final state.

Looking at the sequences generated so far, we see that after $\mathfrak{z}_f$ has been reached the strings $\mathfrak{p}_8$, $\mathfrak{p}_9$ and $\mathfrak{p}_{10}$ can be used, too. So, we get the following 1-solution by appending $\mathfrak{p}_8\mathfrak{p}_6\mathfrak{p}_7\mathfrak{p}_8\mathfrak{p}_7\mathfrak{p}_{10}$ to the already used sequence $\mathfrak{p}_1\mathfrak{p}_2\mathfrak{p}_7\mathfrak{p}_5$ and $\mathfrak{q}_8\mathfrak{q}_6\mathfrak{q}_7\mathfrak{q}_8\mathfrak{q}_7\mathfrak{q}_{10}$ to $\mathfrak{q}_1\mathfrak{q}_2\mathfrak{q}_7\mathfrak{q}_5$ resulting in

$$\begin{aligned} \mathfrak{p}_1\mathfrak{p}_2\mathfrak{p}_7\mathfrak{p}_5\mathfrak{p}_8\mathfrak{p}_6\mathfrak{p}_7\mathfrak{p}_8\mathfrak{p}_7\mathfrak{p}_{10} &= \#\mathfrak{z}_s|\#|\,\mathfrak{z}_1\#\mathfrak{z}_f||\#\mathfrak{z}_f|\#\mathfrak{z}_f\#\# \\ \mathfrak{q}_1\mathfrak{q}_2\mathfrak{q}_7\mathfrak{q}_5\mathfrak{q}_8\mathfrak{q}_6\mathfrak{q}_7\mathfrak{q}_8\mathfrak{q}_7\mathfrak{q}_{10} &= \#\mathfrak{z}_s|\#|\,\mathfrak{z}_1\#\mathfrak{z}_f||\#\mathfrak{z}_f|\#\mathfrak{z}_f\#\# \end{aligned}$$

Exercise 45. *Provide the 1-solution for the PCP obtained for* M^* *when started on input* $||$.

Next, we finish the proof by showing the following claim.

Claim 2. *The computation of* M^* *on input* $\mathfrak{x}$ *stops if and only if the corresponding PCP is 1-solvable.*

We start with the sufficiency. If the corresponding PCP is 1-solvable then the solution must start with $\mathfrak{p}_1$ and $\mathfrak{q}_1$. By construction, $\mathfrak{p}_1 = \#$ and $\mathfrak{q}_1 = \#\mathfrak{z}_s|\cdots|\#$. Looking at Figure 13.2 we see that $|\mathfrak{p}_i| \leqslant |\mathfrak{q}_i|$ for all strings except the ones in the last

group. Thus, as long as M^* does not reach its final state, the concatenation of p's is shorter than the concatenation of q's. Furthermore, $z_s|\cdots|$ corresponds to the initial configuration of M^* on input x. By construction, if the sequence of p's is s and the sequence of q's is sy then s is a sequence of configurations of M^* representing the computation of M^* on input x possibly followed by # and the beginning of the next configuration. Consequently, if the PCP is 1-solvable, then M^* has reached its final state. This proves the sufficiency.

Necessity. If M^* on input x reaches its final state, then, after having started with p_1 and q_1, we can use the p_i's and the corresponding q_i's from the first and second group to obtain the sequence of configurations as well as the separator #. Having done this, we finally use the p_i's and the corresponding q_i's from the second and third group to get the desired solution of the PCP. ∎

Finally, we mention that Theorem 13.3 does not remain true if $\mathrm{card}(\Sigma) = 1$. That is, we have the following.

Exercise 46. *PCP* $[\Sigma, n, \mathfrak{P}, \mathfrak{Q}]$ *is decidable for all* n *provided* $\mathrm{card}(\Sigma) = 1$.

As we mentioned above, PCP can be used to show many interesting results. The following exercise provides an example which is easy to prove. For stating it, we recall the notion of subfunction. Let $f, g: \mathbb{N} \mapsto \mathbb{N}$ be any functions. We say that f is a subfunction of g (written $f \subseteq g$) if for all $x \in \mathbb{N}$ we have: If $f(x)$ is defined then $f(x) = g(x)$.

Exercise 47. *There exists a function* $f \in \mathcal{P}$ *such that there is no function* $g \in \mathcal{R}$ *with* $f \subseteq g$.

Lecture 14: Applications of PCP

Our goal is to present some typical undecidability results for problems arising naturally in formal language theory. Here our main focus is on context-free languages but we shall also look at regular languages and the family $\mathcal{L}_0$.

14.1. Undecidability Results for Context-free Languages

As we have seen in Lecture 6, there are context-free languages L_1 and L_2 such that $L_1 \cap L_2 \notin \mathcal{CF}$. So, it would be nice to have an algorithm which, on input any two context-free grammars $\mathcal{G}_1$, $\mathcal{G}_2$, returns 1 if $L(\mathcal{G}_1) \cap L(\mathcal{G}_2) \in \mathcal{CF}$ and 0 otherwise. Unfortunately, such an algorithm does not exist. Also, some closely related problems cannot be solved algorithmically as our next theorem states.

Theorem 14.1. *The following problems are undecidable for any context-free grammars $\mathcal{G}_1$, $\mathcal{G}_2$:*

(1) $L(\mathcal{G}_1) \cap L(\mathcal{G}_2) = \emptyset$,

(2) $L(\mathcal{G}_1) \cap L(\mathcal{G}_2)$ *is infinite,*

(3) $L(\mathcal{G}_1) \cap L(\mathcal{G}_2) \in \mathcal{CF}$,

(4) $L(\mathcal{G}_1) \cap L(\mathcal{G}_2) \in \mathcal{REG}$.

Proof. The general proof idea is as follows. We construct a context-free language L_S (here S stands for standard) and for any PCP $[\{a, b\}, n, \mathfrak{P}, \mathfrak{Q}]$ a language $L(\mathfrak{P}, \mathfrak{Q})$ such that $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) \neq \emptyset$ if and only if $[\{a, b\}, n, \mathfrak{P}, \mathfrak{Q}]$ is solvable.

Let $\Sigma = \{a, b, c\}$ and define

$$L_S \;=_{df} \{pcqcq^Tcp^T \mid p,\ q \in \{a, b\}^+,\ c \in \Sigma\}\ . \tag{14.1}$$

Furthermore, for any $p_1, \ldots, p_n$, where $p_i \in \{a, b\}^+$, we set

$$L(p_1, \ldots, p_n) \;=_{df} \{ba^{i_k}b \cdots ba^{i_1}cp_{i_1}p_{i_2} \cdots p_{i_k} \mid k \geqslant 1,\ \forall j[1 \leqslant j \leqslant k \ \rightarrow\ 1 \leqslant i_j \leqslant n]\}\ .$$

Here the idea is to encode in a^{i_j} the index i_j. Now, let $[\{a, b\}, n, \mathfrak{P}, \mathfrak{Q}]$ be any PCP, then we define the language $L(\mathfrak{P}, \mathfrak{Q})$ as follows:

$$L(\mathfrak{P}, \mathfrak{Q}) \;=_{df} L(p_1, \ldots, p_n)\{c\}L^T(q_1, \ldots, q_n)\ . \tag{14.2}$$

Claim 1. L_S, $L(p_1, \ldots, p_n)$ *and* $L(\mathfrak{P}, \mathfrak{Q})$ *are context-free.*

First, we define a grammar $\mathcal{G}_S = [\{a, b, c\}, \{\sigma, h\}, \sigma, P]$, where the production set P is defined as follows:

$$\begin{aligned} \sigma &\to a\sigma a \\ \sigma &\to b\sigma b \\ \sigma &\to chc \\ h &\to aha \\ h &\to bhb \\ h &\to c \end{aligned}$$

Now, it is easy to see that $\mathcal{G}_S$ is context-free and that

$$\sigma \overset{*}{\Rightarrow} w\sigma w^T \;\Rightarrow\; wchcw^T \overset{*}{\Rightarrow} wcvhv^Tcw^T \;\Rightarrow\; wcvcv^Tcw^T \;,$$

where $w, v \in \{a, b\}^+$. Hence $L(\mathcal{G}_S) \subseteq L_S$. The inclusion $L_S \subseteq L(\mathcal{G}_S)$ is obvious. Consequently, $L_S \in \mathcal{CF}$.

Next, let $\mathfrak{P} = [p_1, \ldots, p_n]$. We define a grammar $\mathcal{G}_{\mathfrak{P}} = [\{a, b, c\}, \{\sigma\}, \sigma, P]$, where

$$P = \{\sigma \;\to\; ba^i\sigma p_i \mid\; i = 1, \ldots, n\} \cup \{\sigma \;\to\; c\} \;.$$

Clearly, $\mathcal{G}_{\mathfrak{P}}$ is context-free, $L(\mathcal{G}_{\mathfrak{P}}) = L(p_1, \ldots, p_n)$ and thus $L(p_1, \ldots, p_n) \in \mathcal{CF}$.

By Theorem 6.4 we know that $\mathcal{CF}$ is closed under transposition. Therefore, we can conclude that $L^T(p_1, \ldots, p_n) \in \mathcal{CF}$, too. Moreover, $\mathcal{CF}$ is closed under product (cf. Theorem 6.2). Consequently, $L(\mathfrak{P}, \mathfrak{Q})$ is context-free for any PCP. This proves Claim 1.

Claim 2. *For every PCP we have:* $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) \neq \emptyset$ *if and only if* $[\{a, b\}, n, \mathfrak{P}, \mathfrak{Q}]$ *is solvable.*

Necessity. Let $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) \neq \emptyset$ and consider any string $r \in L_S \cap L(\mathfrak{P}, \mathfrak{Q})$, i.e.,

$$r = \underbrace{ba^{i_k}b \cdots ba^{i_1}}_{w_1} c \underbrace{p_{i_1}p_{i_2} \cdots p_{i_k}}_{w_3} c \underbrace{(q_{j_1} \cdots q_{j_m})^T}_{w_4} c \underbrace{(ba^{j_m}b \cdots ba^{j_1})^T}_{w_2} \;.$$

Since $r \in L_S$, we directly see that $w_1 = w_2^T$ and $w_3 = w_4^T$. Consequently, we get $k = m$ and $i_\ell = j_\ell$ for $\ell = 1, \ldots, k$. Thus, the equality $w_3 = w_4^T$ provides a solution of $[\{a, b\}, n, \mathfrak{P}, \mathfrak{Q}]$.

Sufficiency. Let $[\{a, b\}, n, \mathfrak{P}, \mathfrak{Q}]$ be solvable. Then there exist a finite sequence $i_1, i_2 \ldots, i_k$ of natural numbers such that $i_j \leqslant n$ for all $1 \leqslant j \leqslant k$, and $p_{i_1}p_{i_2} \cdots p_{i_k} = q_{i_1}q_{i_2} \cdots q_{i_k}$. So, one directly gets a string $r \in L_S \cap L(\mathfrak{P}, \mathfrak{Q})$. This proves Claim 2.

Claim 1 and 2 together directly imply Assertion (1) via Theorem 13.3.

For showing Assertion (2), we can use the same ideas plus the following Claim.

Claim 3. $L_S \cap L(\mathfrak{P}, \mathfrak{Q})$ *is infinite if and only if* $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) \neq \emptyset$.

The necessity is trivial. For showing the sufficiency, let $i_1, i_2 \ldots, i_k$ be a finite sequence of natural numbers such that $i_j \leqslant n$ for all $1 \leqslant j \leqslant k$, and $p_{i_1}p_{i_2} \cdots p_{i_k} =$

$q_{i_1} q_{i_2} \cdots q_{i_k}$. Therefore, we also have $(p_{i_1} p_{i_2} \cdots p_{i_k})^m = (q_{i_1} q_{i_2} \cdots q_{i_k})^m$, that is, $(i_1, i_2 \ldots, i_k)^m$ is a solution of $[\{a, b\}, n, \mathfrak{P}, \mathfrak{Q}]$ for every $m \geqslant 1$. But this means, if

$$w_1 c w_3 c w_4 c w_2 \in L_S \cap L(\mathfrak{P}, \mathfrak{Q}) \ ,$$

then also

$$w_1^m c w_3^m c w_4^m c w_2^m \in L_S \cap L(\mathfrak{P}, \mathfrak{Q})$$

for every $m \in \mathbb{N}^+$. This proves Claim 3, and thus Assertion (2) is shown.

It remains to prove Assertions (3) and (4). This done via the following claim.

Claim 4. $L_S \cap L(\mathfrak{P}, \mathfrak{Q})$ *does not contain any infinite context-free language.*

Assuming Claim 4, Assertions (3) and (4) can be obtained, since the following assertions are equivalent.

(α) $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) = \emptyset$,

(β) $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) \in \mathcal{RES}$,

(γ) $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) \in \mathcal{CF}$.

Obviously, (α) implies (β) and (β) implies (γ). Thus, we have only to prove that (γ) implies (α). This is equivalent to showing that the negation of (α) implies the negation of (γ).

So, let us assume the negation of (α), i.e., $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) \neq \emptyset$. By Claim 3, we then know that $L_S \cap L(\mathfrak{P}, \mathfrak{Q})$ is infinite. Now Claim 4 tells us that $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) \notin \mathcal{CF}$. Thus, we have shown the negation of (γ).

Under the assumption that Claim 4 is true, we have thus established the equivalence of (α), (β) and (γ). Consequently, by Claim 2 we then know that $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) \in \mathcal{CF}$ if and only if $[\{a, b\}, n, \mathfrak{P}, \mathfrak{Q}]$ is not solvable and also that $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) \in \mathcal{RES}$ if and only if $[\{a, b\}, n, \mathfrak{P}, \mathfrak{Q}]$ is not solvable. This proves Assertions (3) and (4).

So, it remains to show Claim 4. Let $[\{a, b\}, n, \mathfrak{P}, \mathfrak{Q}]$ be arbitrarily fixed. Suppose there is a language $L \subseteq L_S \cap L(\mathfrak{P}, \mathfrak{Q})$ such that L is infinite and context-free. Now we apply Theorem 7.4. Thus, there exists $k \in \mathbb{N}$ such that for all $w \in L$ with $|w| > k$ there are strings q, r, s, u, v such that $w = qrsuv$ and $ru \neq \lambda$ and $qr^i su^i v \in L$ for all $i \in \mathbb{N}^+$.

Since by supposition L is infinite there must exist a string $w \in L$ with $|w| > k$. Furthermore, $L \subseteq L(\mathfrak{P}, \mathfrak{Q})$ and therefore w must have the form

$$w = w_1 c w_2 c w_3 c w_4 \ , \text{ where } w_i \in \{a, b\}^+, \ i = 1, 2, 3, 4 \ . \tag{14.3}$$

That is, w contains exactly three times the letter c. Now, let $w = qrsuv$. We distinguish the following cases.

Case 1. c is a substring of ru.

Then $\mathfrak{q}\mathfrak{r}^i\mathfrak{s}\mathfrak{u}^i\mathfrak{v} \in L$ for every $i \in \mathbb{N}^+$ and hence $\mathfrak{q}\mathfrak{r}^i\mathfrak{s}\mathfrak{u}^i\mathfrak{v}$ contains at least four $\mathfrak{c}$'s. for every $i \geqslant 4$. So, Case 1 cannot happen.

Case 2. $\mathfrak{c}$ is not a substring of $\mathfrak{r}\mathfrak{u}$.

Then, neither $\mathfrak{r}$ nor $\mathfrak{u}$ could be substrings of $\mathfrak{b}\mathfrak{a}^{i_1}\mathfrak{b}\cdots\mathfrak{b}\mathfrak{a}^{i_k}$. If $\mathfrak{r}$ or $\mathfrak{u}$ start and end with $\mathfrak{b}$, then $\mathfrak{q}\mathfrak{r}^2\mathfrak{s}\mathfrak{u}^2\mathfrak{v} \notin L(\mathfrak{P}, \mathfrak{Q})$. If both $\mathfrak{r}$ and $\mathfrak{u}$ do not start and end with $\mathfrak{b}$, then $\mathfrak{r} \in \{\mathfrak{a}\}^+$ or $\mathfrak{u} \in \{\mathfrak{a}\}^+$ is impossible, since $\mathfrak{q}\mathfrak{r}^{n+1}\mathfrak{s}\mathfrak{u}^{n+1}\mathfrak{v}$ then violates the condition that we have at most n consecutive $\mathfrak{a}$'s. Otherwise, we get a contradiction to the definition of $L(\mathfrak{P}, \mathfrak{Q})$, since then we have more blocks of $\mathfrak{a}$'s as there are $\mathfrak{p}_i$'s or $\mathfrak{q}_j$'s.

Finally, the only remaining possibility is $\mathfrak{r}$ is a substring of w_2 or $\mathfrak{r}$ is a substring of w_3 (cf. (14.3)). In either case, then $\mathfrak{q}\mathfrak{r}^i\mathfrak{s}\mathfrak{u}^i\mathfrak{v} \notin L(\mathfrak{P}, \mathfrak{Q})$ for $i \geqslant 2$, since the length of w_1 and w_2 as well as the length of w_3 and w_4 are related. This is one of the reasons we have included the $\mathfrak{a}^{i_j}$ into the definition of $L(\mathfrak{P}, \mathfrak{Q})$. This proves Claim 4, and thus the theorem is shown. ∎

Next, we turn our attention to problems involving the complement of context-free languages. Recall that $\mathfrak{cf}$ is not closed under complement (cf. Corollary 6.6). However, due to lack of time, we have to omit a certain part of the following proof.

Theorem 14.2. *The following problems are undecidable for any context-free grammar $\mathcal{G}$:*

(1) $\overline{L(\mathcal{G})} = \emptyset$,

(2) $\overline{L(\mathcal{G})}$ *is infinite,*

(3) $\overline{L(\mathcal{G})} \in \mathcal{CF}$,

(4) $\overline{L(\mathcal{G})} \in \mathcal{RE}\mathcal{G}$,

(5) $L(\mathcal{G}) \in \mathcal{RE}\mathcal{G}$.

Proof. We use the notions from the demonstration of Theorem 14.1. Consider $L =_{df} \overline{L_S \cap L(\mathfrak{P}, \mathfrak{Q})}$. Note that L is always context-free. We do not prove this assertion here. The interested reader is referred to Ginsburg [1].

For showing (1), suppose the converse. Then, given the fact that L is context-free, there is a context-free grammar $\mathcal{G}$ such that $L = L(\mathcal{G})$. So, we could run the algorithm on input $\mathcal{G}$. On the other hand, $\overline{L} = L_S \cap L(\mathfrak{P}, \mathfrak{Q})$. Thus, we could decide whether or not $L_S \cap L(\mathfrak{P}, \mathfrak{Q}) = \emptyset$. By Claim 2 in the proof of Theorem 14.1, this implies that we can decided whether or not a PCP is solvable; a contradiction to Theorem 13.3.

Assertion (2) is shown analogously via (2) of Theorem 14.1.

Assertion (3) and (4) also follow directly from Theorem 14.1 by using its Assertions (3) and (4), respectively.

Finally, Assertion (5) is a direct consequence of Assertion (4) and the fact that $L \in \mathcal{RE}\mathcal{G}$ if and only if $\overline{L} \in \mathcal{RE}\mathcal{G}$ (cf. Problem 4.2). ∎

Furthermore, the theorems shown above directly allow the following corollary.

Corollary 14.3. *The following problems are undecidable for any context-free grammars $\mathcal{G}_1$, $\mathcal{G}_2$:*

(1) $\mathsf{L}(\mathcal{G}_1) = \mathsf{L}(\mathcal{G}_2)$,

(2) $\mathsf{L}(\mathcal{G}_1) \subseteq \mathsf{L}(\mathcal{G}_2)$.

Proof. Suppose (1) is decidable. Let $\mathcal{G}_1$ be any context-free grammar such that $\mathsf{L}(\mathcal{G}_1) = \mathsf{L} = \overline{\mathsf{L}_S \cap \mathsf{L}(\mathfrak{P}, \mathfrak{Q})}$ (see the proof of Theorem 14.2). Furthermore, let $\mathcal{G}_2$ be any context-free grammar such that $\mathsf{L}(\mathcal{G}_2) = \{\mathfrak{a}, \mathfrak{b}, \mathfrak{c}\}^*$. Then

$$\begin{aligned} \mathsf{L} = \{\mathfrak{a}, \mathfrak{b}, \mathfrak{c}\}^* &\iff \overline{\mathsf{L}_S \cap \mathsf{L}(\mathfrak{P}, \mathfrak{Q})} = \{\mathfrak{a}, \mathfrak{b}, \mathfrak{c}\}^* \iff \mathsf{L}_S \cap \mathsf{L}(\mathfrak{P}, \mathfrak{Q}) = \emptyset \\ &\iff [\{\mathfrak{a}, \mathfrak{b}\}, \mathfrak{n}, \mathfrak{P}, \mathfrak{Q}] \text{ is not solvable ,} \end{aligned}$$

a contradiction to Theorem 13.3. This proves Assertion (1).

If (2) would be decidable, then (1) would be decidable, too, since

$$\mathsf{L}(\mathcal{G}_1) = \mathsf{L}(\mathcal{G}_2) \iff \mathsf{L}(\mathcal{G}_1) \subseteq \mathsf{L}(\mathcal{G}_2) \text{ and } \mathsf{L}(\mathcal{G}_2) \subseteq \mathsf{L}(\mathcal{G}_1) \ , \tag{14.4}$$

a contradiction. Therefore, we obtain that (2) is not decidable and Assertion (2) is shown. ∎

14.2. Back to Regular Languages

This is a good place to summarize our knowledge about regular languages and to compare the result obtained to the undecidability results for context-free languages shown above.

By Theorem 3.2 we know that $\mathsf{L} \in \mathcal{RES}$ if and only if there exists a deterministic finite automaton $\mathcal{A}$ such that $\mathsf{L} = \mathsf{L}(\mathcal{A})$. Therefore, we can directly get the following corollary.

Corollary 14.4. *The regular languages are closed under complement.*

Proof. Let $\mathsf{L} \in \mathcal{RES}$ be any language. By Theorem 3.2 there exists a deterministic finite automaton $\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ such that $\mathsf{L} = \mathsf{L}(\mathcal{A})$. Let $\overline{\mathcal{A}} = [\Sigma, Q, \delta, q_0, Q \setminus F]$. Then, we obviously have $\overline{\mathsf{L}} = \mathsf{L}(\overline{\mathcal{A}})$. ∎

Furthermore, by Theorem 2.1, the regular languages are closed under union, product and Kleene closure. So, recalling a bit set theory, we know that $\mathsf{L}_1 \cap \mathsf{L}_2 = \overline{\overline{\mathsf{L}_1} \cup \overline{\mathsf{L}_2}}$. Hence we directly get the following corollary.

Corollary 14.5. *The regular languages are closed under intersection.*

Moreover, as shown in Lecture 4, there is an algorithm which on input any regular grammar $\mathcal{G}$ decides whether or not $\mathsf{L}(\mathcal{G})$ is infinite (cf. Theorem 4.5). Using the algorithm given in the proof of Theorem 4.5, we can also conclude that there is an

algorithm which on input any regular grammar $\mathcal{G}$ decides whether or not $L(\mathcal{G})$ is finite. Looking at the proof of Theorem 4.5 we also get the following corollary.

Corollary 14.6. *There is an algorithm which on input any regular grammar $\mathcal{G}$ decides whether or not $L(\mathcal{G}) = \emptyset$.*

Proof. Let $\mathcal{G}$ be a regular grammar. The algorithm first constructs a deterministic finite automaton $\mathcal{A} = [\Sigma, Q, \delta, q_0, F]$ such that $L(\mathcal{G}) = L(\mathcal{A})$. Let $\mathrm{card}(Q) = n$. Then, the algorithm checks whether or not there is a string s such that $n+1 \leqslant |s| \leqslant 2n+2$ with $s \in L(\mathcal{A})$.

If there is no such string, then by the proof of Theorem 4.5 we already know that $L(\mathcal{G})$ is finite. Thus, it suffices to check whether or not there is a string s such that $|s| \leqslant n$ and $s \in L(\mathcal{A})$.

If there is such string, output $\mathcal{G} \neq \emptyset$. Otherwise, output $\mathcal{G} = \emptyset$. ∎

Of course, this is not the most efficient algorithm.

Finally, you have shown that $\mathcal{REG}$ is closed under set difference (cf. Problem 4.2 of our Exercise sets). Thus, we also have the following corollary.

Corollary 14.7. *There is an algorithm which on input any regular grammars $\mathcal{G}_1$ and $\mathcal{G}_2$ decides whether or not $L(\mathcal{G}_1) \subseteq L(\mathcal{G}_2)$.*

Proof. Since $\mathcal{REG}$ is closed under set difference, we know that $L(\mathcal{G}_1) \setminus L(\mathcal{G}_2) \in \mathcal{REG}$. Furthermore, a closer inspection of the proof given shows that we can construct a grammar $\mathcal{G}$ such that $L(\mathcal{G}) = L(\mathcal{G}_1) \setminus L(\mathcal{G}_2)$. Again recalling a bit set theory, we have

$$L(\mathcal{G}_1) \subseteq L(\mathcal{G}_2) \text{ if and only if } L(\mathcal{G}) = \emptyset \ .$$

By Corollary 14.6 it is decidable whether or not $L(\mathcal{G}) = \emptyset$. ∎

Using the same idea as in the proof of Corollary 14.3 (see (14.4)), we also see that there is an algorithm which on input any regular grammars $\mathcal{G}_1$ and $\mathcal{G}_2$ decides whether or not $L(\mathcal{G}_1) = L(\mathcal{G}_2)$.

Thus, we can conclude that all the problems considered in Theorems 14.1 and 14.2 and Corollary 14.3 when stated *mutatis mutandis* for regular languages are *decidable*.

14.3. Results concerning $\mathcal{L}_0$

We need some more notations which we introduce below.

Let $A \subseteq \mathbb{N}$ be any set. We have already defined the characteristic function χ_A, i.e., $\chi_A(x) = 1$ if $x \in A$ and $\chi_A(x) = 0$ if $x \notin A$. Next, we introduce the ***partial characteristic function*** π_A defined as follows.

$$\pi_A(x) = \begin{cases} 1 \ , & \text{if } x \in A \\ \text{not defined} \ , & \text{otherwise} \ . \end{cases}$$

Now we can formally define what does it mean that a set is recursive and recursively enumerable, respectively.

Definition 46. *Let $A \subseteq \mathbb{N}$; then A is said to*

(1) *be* ***recursive*** *if* $\chi_A \in \mathcal{R}$,

(2) *be* ***recursively enumerable*** *if* $\pi_A \in \mathcal{P}$.

We continue with some examples.

(1) The set of all prime numbers is recursive.

(2) The set of all odd numbers is recursive.

(3) The set of all even numbers is recursive.

(4) $\emptyset$ is recursive.

(5) Every finite set is recursive.

(6) Every recursive set is recursively enumerable.

(7) Let A be any recursive set. Then its complement $\overline{A}$ is recursive, too.

The following theorem establishes a characterization of recursive sets.

Theorem 14.8. *Let* $A \subseteq \mathbb{N}$*; then we have:* A *is recursive if and only if* A *is recursively enumerable and* $\overline{A}$ *is recursively enumerable.*

Proof. Necessity. If A is recursive then A is recursively enumerable. Furthermore, as mentioned above, the complement $\overline{A}$ of every recursive set A is recursive, too. Thus, $\overline{A}$ is also recursively enumerable.

Sufficiency. If both A and $\overline{A}$ are recursively enumerable, then $\pi_A \in \mathcal{P}$ and $\pi_{\overline{A}} \in \mathcal{P}$. Thus, we can express χ_A as follows.

$$\chi_A(x) = \begin{cases} 1\,, & \text{if } \pi_A(x) = 1 \\ 0\,, & \text{if } \pi_{\overline{A}}(x) = 1\,. \end{cases}$$

Consequently, $\chi_A \in \mathcal{R}$. ∎

The latter theorem can be used to show that there are recursively enumerable sets which are not recursive.

Corollary 14.9. *The halting set* K *is recursively enumerable but not recursive.*

Proof. Recall that $K = \{i \mid i \in \mathbb{N},\ \psi_i(i)\downarrow\}$, where ψ is universal for $\mathcal{P}$. So, we consider here the universal numbering ψ that corresponds to the universal Turing machine U (cf. Theorem 12.3). Hence, we have the following equivalence:

$$\pi_K(i) = 1 \iff \psi_i(i)\downarrow \iff U(i,i) \text{ stops}\,.$$

Thus, in order to compute $\pi_K(i)$ it suffices to start the universal Turing machine U on input (i,i). If the computation terminates, the algorithm for computing $\pi_K(i)$ outputs 1. Otherwise, the algorithm does not terminate. Consequently, $\pi_K \in \mathcal{P}$ and thus K is recursively enumerable.

Suppose K to be recursive. Then, $\chi_K \in \mathcal{R}$, a contradiction to Theorem 13.1. ∎

By Theorem 14.8, we directly arrive at the following corollary.

Corollary 14.10. *The complement* $\overline{\mathsf{K}}$ *of the halting set* K *is not recursively enumerable.*

So, it remains to relate the language family $\mathcal{L}_0$ to Turing machines. This is done by the following theorems. In the following, by "any type-0 grammar" we always mean a grammar as defined in Definition 6 without any restrictions.

Theorem 14.11. *Let* $\mathcal{G}$ *be any type-0 grammar. Then there exists a Turing machine* M *such that* $\mathsf{L}(\mathcal{G}) = \mathsf{L}(\mathsf{M})$.

The opposite is also true.

Theorem 14.12. *For every Turing machine* M *there exists a type-0 grammar* $\mathcal{G}$ *such that* $\mathsf{L}(\mathsf{M}) = \mathsf{L}(\mathcal{G})$.

Theorems 14.11 and 14.12 directly allow the following corollary.

Corollary 14.13. $\mathcal{L}_0$ *is equal to the family of all recursively enumerable sets.*

Now, we are ready to show the theorem already announced at the end of Lecture 10, i.e., that Theorem 10.12 does not generalize to $\mathcal{L}_0$.

Theorem 14.14. *There does not exist any algorithm that on input any type-0 grammar* $\mathcal{G} = [\mathsf{T}, \mathsf{N}, \sigma, \mathsf{P}]$ *and any string* $\mathsf{s} \in \mathsf{T}^*$ *decides whether or not* $\mathsf{s} \in \mathsf{L}(\mathcal{G})$.

Proof. By Corollary 14.9 we know that K is recursively enumerable but not recursive. Since $\pi_{\mathsf{K}} \in \mathcal{P}$, there exists a Turing machine computing π_{K}. By Theorem 14.12 there is a grammar $\mathcal{G}$ such that $\mathsf{L}(\mathcal{G}) = \mathsf{K}$. Thus, if we could decide $\mathsf{s} \in \mathsf{L}(\mathcal{G})$ for every $\mathsf{s} \in \mathbb{N}$, then K would be recursive, a contradiction. ∎

Since $\overline{\mathsf{K}}$ is not recursively enumerable (cf. Corollary 14.10), we also have the following corollary.

Corollary 14.15. $\mathcal{L}_0$ is not closed under complement and set difference.

Interestingly, the ideas used in the proof of Theorem 2.1 directly yield the following theorem.

Theorem 14.16. *The language family* $\mathcal{L}_0$ *is closed under union, product and Kleene closure.*

Finally, we show further undecidability results.

Theorem 14.17. *There does not exist any algorithm which, on input any type-0 grammar* $\mathcal{G}$ *decides whether or not* $\mathsf{L}(\mathcal{G}) = \emptyset$.

Proof. Suppose the converse. Let $\mathcal{G} = [\mathsf{T}, \mathsf{N}, \sigma, \mathsf{P}]$ be any grammar and let $w \in \mathsf{T}^*$ be arbitrarily fixed. We construct a grammar $\tilde{\mathcal{G}} = [\mathsf{T}, \mathsf{N} \cup \{\tilde{\sigma}, \#\}, \tilde{\sigma}, \tilde{\mathsf{P}}]$, where $\tilde{\mathsf{P}}$ is as follows:

$$\tilde{\mathsf{P}} \;=\; \mathsf{P} \cup \{\tilde{\sigma} \;\to\; \#\sigma\#,\; \#w\# \;\to\; \lambda\}\,.$$

Note that $\#w\# \rightarrow \lambda$ is the only production which can remove the # symbols. Thus, we get

$$\mathsf{L}(\tilde{\mathcal{G}}) = \begin{cases} \{\lambda\}, & \text{if } w \in \mathsf{L}(\mathcal{G}) \\ \emptyset, & \text{otherwise.} \end{cases}$$

Consequently, $w \notin \mathsf{L}(\mathcal{G})$ if and only if $\mathsf{L}(\tilde{\mathcal{G}}) = \emptyset$. Thus, we can decide $\mathsf{L}(\tilde{\mathcal{G}}) = \emptyset$ if and only if we can decide $w \notin \mathsf{L}(\mathcal{G})$. But the latter problem is undecidable (cf. Theorem 14.14). Since the construction of $\tilde{\mathcal{G}}$ can be done algorithmically, we obtain a contradiction. ∎

Corollary 14.18. *The following problems are undecidable for any type-0 grammars $\mathcal{G}_1$, $\mathcal{G}_2$:*

(1) $\mathsf{L}(\mathcal{G}_1) = \mathsf{L}(\mathcal{G}_2)$,

(2) $\mathsf{L}(\mathcal{G}_1) \subseteq \mathsf{L}(\mathcal{G}_2)$.

Proof. Let $\mathcal{G}_\emptyset$ be any type-0 grammar such that $\mathsf{L}(\mathcal{G}_\emptyset) = \emptyset$. Then we have for any type-0 grammar $\mathcal{G}$:

$$\mathsf{L}(\mathcal{G}) = \emptyset \iff \mathsf{L}(\mathcal{G}) \subseteq \mathsf{L}(\mathcal{G}_\emptyset) \iff \mathsf{L}(\mathcal{G}) = \mathsf{L}(\mathcal{G}_\emptyset),$$

and thus both the inclusion and equivalence problem, respectively, are reduced to the decidability of the emptiness problem. Since the emptiness problem is undecidable by Theorem 14.17, the corollary is shown. ∎

As a general "rule of thumb" you should memorize that every *non-trivial* problem is undecidable for type-0 grammars. Here by non-trivial we mean that there are infinitely many grammars satisfying the problem and infinitely many grammars not satisfying the problem.

Exercise 48. *Prove or disprove:*

The emptiness problem is undecidable for context-sensitive grammars.

14.4. Summary

Finally, we present a table summarizing many important results. All problems in this table should be read as follows:

Does there exist an algorithm which on input any grammar $\mathcal{G}$ and any string s or any grammar $\mathcal{G}$ or any two grammars $\mathcal{G}_1$, $\mathcal{G}_2$, respectively, returns 1 if the property on hand is fulfilled and 0 if it is not fulfilled.

We use + to indicate that the desired algorithm exists and − to indicate that the desired algorithm does *not* exist.

		$\mathcal{RES}$	$\mathcal{CF}$	$\mathcal{CS}$	Type 0
1)	$s \in L(\mathcal{G})$	+	+	+	−
2)	$L(\mathcal{G}_1) \subseteq L(\mathcal{G}_2)$	+	−	−	−
3)	$L(\mathcal{G}_1) = L(\mathcal{G}_2)$	+	−	−	−
4)	$L(\mathcal{G}) = \emptyset$	+	+	−	−
5)	$L(\mathcal{G})$ finite	+	+	−	−
6)	$L(\mathcal{G})$ infinite	+	+	−	−
7)	$\overline{L(\mathcal{G})} = \emptyset$	+	−	−	−
8)	$\overline{L(\mathcal{G})}$ infinite	+	−	−	−
9)	$\overline{L(\mathcal{G})}$ has the same type as $L(\mathcal{G})$	+	−	+	−
10)	$\overline{L(\mathcal{G})} \in \mathcal{RES}$	+	−	−	−
11)	$\overline{L(\mathcal{G})} \in \mathcal{CF}$	+	−	−	−
12)	$L(\mathcal{G}_1) \cap L(\mathcal{G}_2) = \emptyset$	+	−	−	−
13)	$L(\mathcal{G}_1) \cap L(\mathcal{G}_2)$ finite	+	−	−	−
14)	$L(\mathcal{G}_1) \cap L(\mathcal{G}_2)$ infinite	+	−	−	−
15)	$L(\mathcal{G}) = T^*$	+	−	−	−
16)	$L(\mathcal{G}) \in \mathcal{RES}$	+	−	−	−

References

[1] S. GINSBURG (1966), *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, Inc, New York, NY, USA.

Lecture 15: Numberings, Complexity

As we have seen, there are numberings that are universal for $\mathcal{P}$ (cf. Theorem 12.3). Within this lecture we study further properties of numberings. Some more notations are needed.

For every $\psi \in \mathcal{P}^2$ we set $\mathcal{P}_\psi = \{\psi_i \mid i \in \mathbb{N}\}$. Next, we introduce the notion of reducibility.

Definition 47. *Let ψ, $\psi' \in \mathcal{P}^2$ be any numberings. We say that ψ is **reducible** to ψ' (written $\psi \leqslant \psi'$) if there exists a function $c \in \mathcal{R}$ such that $\psi_i = \psi'_{c(i)}$ for all $i \in \mathbb{N}$.*

Clearly, if $\psi \leqslant \psi'$ then $\mathcal{P}_\psi \subseteq \mathcal{P}_{\psi'}$. If we interpret every i as a program, then the function c reducing ψ to ψ' can be interpreted as a *compiler*. This is the reason we called the reduction function c.

Exercise 49. *Prove reducibility to be reflexive and transitive.*

Furthermore, it should be noted that $\leqslant$ is not trivial. In particular, Friedberg has shown the following.

Theorem 15.1. *There are numberings ψ, $\psi' \in \mathcal{P}^2$ such that $\mathcal{P}_\psi = \mathcal{P}_{\psi'}$ but neither $\psi \leqslant \psi'$ nor $\psi' \leqslant \psi$.*

We do not prove this theorem here, since we do not need it. However, we mention that in Theorem 15.1 $\psi\ \psi' \in \mathsf{Pa}^2$ may be chosen such that $\mathcal{P}_\psi = \mathcal{P}_{\psi'} = \mathcal{P}$, i.e., $\psi\ \psi'$ can be universal for $\mathcal{P}$.

So it is only natural to ask whether or not there are numberings that are maximal with respect to reducibility. The answer is provided in the following subsection.

15.1. Gödel Numberings

Definition 48. *Let $\varphi \in \mathcal{P}^2$; we call φ a **Gödel numbering** if*

(1) $\mathcal{P}_\varphi = \mathcal{P}$, *and*

(2) $\psi \leqslant \varphi$ *for every numbering $\psi \in \mathcal{P}^2$.*

The following theorem establishes the existence of Gödel numberings.

Theorem 15.2. *There exists a Gödel numbering.*

Proof. In fact, we have already constructed a Gödel numbering when proving Theorem 12.3. So, let φ be the numering corresponding to the universal Turing machine U. Thus, we already know $\mathcal{P}_\varphi = \mathcal{P}$.

Claim 1. *For every numbering $\psi \in \mathcal{P}^2$ there is a function $c \in \mathcal{R}$ such that $\psi_i(x) = \varphi_{c(i)}(x)$ for all i, $x \in \mathbb{N}$.*

Since $\psi \in \mathcal{P}^2$, there exists a Turing machine M computing ψ. Now, if we fix the first argument i then we get a Turing machine M_i computing ψ_i. Then we set $c(i) = \mathit{cod}(M_i)$, where *cod* is the computable function from Theorem 12.3. Consequently, $c(i)$ is defined for every $i \in \mathbb{N}$ and $\mathit{cod}(M_i)$ can be computed from the knowledge of M. Finally, we have

$$\psi_i(x) = f_M(i,x) = f_{M_i}(x) = f_U(\mathit{cod}(M_i), x) = f_U(c(i), x) = \varphi_{c(i)}(x) \ ,$$

and Claim 1 follows. ∎

Now that we know that there is one Gödel numbering, it is only natural to ask if it is the only one. The negative answer is provided by the next theorem.

Theorem 15.3. *There are countably many Gödel numberings for $\mathcal{P}$.*

Proof. Let φ be the Gödel numbering from Theorem 15.2 and let $\psi \in \mathcal{P}^2$ be any function. We can thus define $\varphi' \in \mathcal{P}^2$ by setting $\varphi'_{2i} = \varphi_i$ and $\varphi'_{2i+1} = \psi_i$ for all $i \in \mathbb{N}$.

Clearly, φ' is universal for $\mathcal{P}$. Moreover, using $c(i) = 2i$ we get $\varphi \leqslant \varphi'$. Since $\leqslant$ is transitive, we thus have shown φ' to be a Gödel numbering. ∎

So, the Gödel numberings are the maximal elements of the lattice $[\mathcal{P}, \leqslant]$. Moreover all Gödel numberings are equivalent in the following sense.

Theorem 15.4 (Theorem of Rogers). *For any two Gödel numberings φ and ψ of $\mathcal{P}$ there exists a permutation $\pi \in \mathcal{R}$ such that for all $i \in \mathbb{N}$ we have $\varphi_i = \psi_{\pi(i)}$ and $\psi_i = \varphi_{\pi^{-1}(i)}$.*

15.2. The Recursion and the Fixed Point Theorem

Theorem 15.5 (Fixed Point Theorem). *Let $\varphi \in \mathcal{P}^2$ be any Gödel numbering. Then, for every function $h \in \mathcal{R}$ there exists a number a such that $\varphi_{h(a)} = \varphi_a$.*

Proof. First we define a function τ as follows. For all $i, x \in \mathbb{N}$ let

$$\tau(i,x) = \begin{cases} \varphi_{\varphi_i(i)}(x) \ , & \text{if } \varphi_i(i) \downarrow \\ \text{not defined} \ , & \text{otherwise} \ . \end{cases}$$

Clearly, $\tau \in \mathcal{P}^2$ and thus τ is a numbering, too. By construction we obtain

$$\tau_i = \begin{cases} \varphi_{\varphi_i(i)} \ , & \text{if } \varphi_i(i) \downarrow \\ e \ , & \text{otherwise} \ , \end{cases}$$

where e denotes the nowhere defined function. By the definition of a Gödel numbering there exists a function $c \in \mathcal{R}$ such that

$$\tau_i = \varphi_{c(i)} \text{ for all } i \in \mathbb{N} \ . \tag{15.1}$$

Next, we consider $g(x) = h(c(x))$ for all $x \in \mathbb{N}$. Since $h,\ c \in \mathcal{R}$, we also know that $g \in \mathcal{R}$. Since φ is universal for $\mathcal{P}$, there exists a $\nu \in \mathbb{N}$ such that

$$g = \varphi_\nu \ . \tag{15.2}$$

Consequently, $\varphi_\nu(\nu) \downarrow$ because of $g \in \mathcal{R}$. Hence, we obtain

$$\begin{aligned}
\tau_\nu &= \varphi_{c(\nu)} \quad \text{by the definition of } c \text{ , see (15.1)} \\
&= \varphi_{\varphi_\nu(\nu)} \quad \text{by the definition of } \tau \text{ and because of } \varphi_\nu(\nu) \downarrow \\
&= \varphi_{g(\nu)} \quad \text{since } g = \varphi_\nu \text{ , see (15.2)} \\
&= \varphi_{h(c(\nu))} \quad \text{by the definition of } g \text{ .}
\end{aligned}$$

Thus, we have $\varphi_{c(\nu)} = \varphi_{h(c(\nu))}$ and hence setting $a = c(\nu)$ proves the theorem. ∎

The proof directly allows the following corollary telling us that fixed points can be even computed.

Corollary 15.6. *For every Gödel numbering φ here exists a function $fix \in \mathcal{R}$ such that for all $z \in \mathbb{N}$ we have:*

If $\varphi_z \in \mathcal{R}$ then $\varphi_{\varphi_z(fix(z))} = \varphi_{fix(z)}$.

Note that the fixed point theorem has been discovered by Rogers. The following theorem is due to Kleene.

Theorem 15.7 (Recursion Theorem). *For every numbering $\psi \in \mathcal{P}^2$ and every Gödel numbering $\varphi \in \mathcal{P}^2$ there exists a number $a \in \mathbb{N}$ such that $\varphi_a(x) = \psi(a, x)$ for all $x \in \mathbb{N}$.*

Proof. Since $\psi \in \mathcal{P}^2$ and since $\varphi \in \mathcal{P}^2$ is a Gödel numbering, there exists a function $c \in \mathcal{R}$ such that

$$\psi_i = \varphi_{c(i)} \text{ for all } i \in \mathbb{N} \text{ .}$$

By the fixed point theorem and the choice of function c, there is a number $a \in \mathbb{N}$ such that

$$\psi_a = \varphi_{c(a)} = \varphi_a \text{ ,}$$

and consequently we have $\psi(a, x) = \varphi_a(x)$ for all $x \in \mathbb{N}$. ∎

Note that the recursion theorem also implies the fixed point theorem. We leave the proof as an exercise.

There are also important generalizations of the fixed point and recursion theorem. We mention here only one which has been found by Smullyan. Further generalizations can be found in Smith [5].

Theorem 15.8 (Smullyan's Double Fixed Point Theorem). *Let φ be any Gödel numbering for $\mathcal{P}$ and let $h_1, h_2 \in \mathcal{R}^2$. Then there exist $i_1, i_2 \in \mathbb{N}$ such that simultaneously*

$$\varphi_{i_1} = \varphi_{h_1(i_1,i_2)} \text{ and } \varphi_{i_2} = \varphi_{h_2(i_1,i_2)}$$

are satisfied.

Exercise 50. *Let φ be any Gödel numbering for $\mathcal{P}$. Prove that there always exists an $i \in \mathbb{N}$ such that $\varphi_i = \varphi_{i+1}$.*

15.2.1. The Theorem of Rice

Next we show that all nontrivial properties of programs are undecidable. For stating the corresponding theorem, we must define what is commonly called an index set.

Definition 49. *Let $\varphi \in \mathcal{P}^2$ be any Gödel numbering and let $\mathcal{P}' \subseteq \mathcal{P}$. Then the set*

$$\Theta_\varphi \mathcal{P}' = \{i |\ i \in \mathbb{N} \text{ and } \varphi_i \in \mathcal{P}'\}$$

*is called **index set of $\mathcal{P}'$**.*

The term "index" in the definition above is synonymous in its use to the term "program." However, we follow here the traditional terminology, since *program set* could lead to confusion. One may be tempted to interpret *program set* as set of programs, i.e., a collection of programs that may or may not be index set. Furthermore, the technical term "undecidable" is used as a synonym for "not recursive."

Theorem 15.9 (Theorem of Rice). *Let $\varphi \in \mathcal{P}^2$ be any Gödel numbering. Then $\Theta_\varphi \mathcal{P}'$ is undecidable for every set $\mathcal{P}'$ with $\emptyset \subset \mathcal{P}' \subset \mathcal{P}$.*

Proof. Suppose the converse, i.e., there is a set $\mathcal{P}'$ such that $\emptyset \subset \mathcal{P}' \subset \mathcal{P}$ and $\Theta_\varphi \mathcal{P}'$ is decidable. Thus $\chi_{\Theta_\varphi \mathcal{P}'} \in \mathcal{R}$. Since $\emptyset \subset \mathcal{P}' \subset \mathcal{P}$, there exists a function $g \in \mathcal{P} \setminus \mathcal{P}'$ and a function $f \in \mathcal{P}'$. Let u be any program for g, i.e., $\varphi_u = g$ and let z be any program for f, i.e., $\varphi_z = f$. Next, we define a function h as follows. For all $i \in \mathbb{N}$ we set

$$h(i) = \begin{cases} u\,, & \text{if } \chi_{\Theta_\varphi \mathcal{P}'}(i) = 1 \\ z\,, & \text{otherwise}\,, \end{cases}$$

Since $\chi_{\Theta_\varphi \mathcal{P}'} \in \mathcal{R}$, we can directly conclude that $h \in \mathcal{R}$, too. Hence, by the fixed point theorem, there exists an $a \in \mathbb{N}$ such that $\varphi_{h(a)} = \varphi_a$. We distinguish the following cases.

Case 1. $\varphi_a \in \mathcal{P}'$

Then $\chi_{\Theta_\varphi \mathcal{P}'}(a) = 1$, and thus $h(a) = u$. Consequently,

$$\varphi_a = \varphi_{h(a)} = \varphi_u = g \notin \mathcal{P}'\,,$$

a contradiction to $\chi_{\Theta_\varphi \mathcal{P}'}(a) = 1$.

Case 2. $\varphi_a \notin \mathcal{P}'$

Then $\chi_{\Theta_\varphi \mathcal{P}'}(a) = 0$, and therefore $h(a) = z$. Consequently,

$$\varphi_a = \varphi_{h(a)} = \varphi_z = f \in \mathcal{P}'\,,$$

a contradiction to $\chi_{\Theta_\varphi \mathcal{P}'}(a) = 0$.

Hence, our supposition $\chi_{\Theta_\varphi \mathcal{P}'} \in \mathcal{R}$ must be wrong, and thus the set $\Theta_\varphi \mathcal{P}'$ is undecidable. ∎

Example 21. Let $\mathcal{P}' = Prim$. As we have seen, many functions are primitive recursive, and thus $Prim \neq \emptyset$. Since every primitive recursive function is total, we also have $Prim \subset \mathcal{P}$. Consequently, $\Theta_\varphi Prim$ is undecidable. That is, there does not exist any algorithm which on input any program $\mathfrak{i} \in \mathbb{N}$ can decide whether or not it computes a primitive recursive function.

The theorem of Rice also directly allows the following corollary.

Corollary 15.10. *Let $\varphi \in \mathcal{P}^2$ be any Gödel numbering. Then for every function $\mathfrak{f} \in \mathcal{P}$ the set $\{\mathfrak{i} \mid \mathfrak{i} \in \mathbb{N} \text{ and } \varphi_\mathfrak{i} = \mathfrak{f}\}$ is infinite.*

Proof. Setting $\mathcal{P}' = \{\mathfrak{f}\}$, we directly get $\emptyset \subset \mathcal{P}' \subset \mathcal{P}$. Thus the assumptions of the theorem of Rice are fulfilled and therefore $\Theta_\varphi \mathcal{P}'$ is undecidable. On the other hand, $\Theta_\varphi \mathcal{P}' = \{\mathfrak{i} \mid \mathfrak{i} \in \mathbb{N} \text{ and } \varphi_\mathfrak{i} = \mathfrak{f}\}$. Since every finite set is decidable, we can conclude that $\{\mathfrak{i} \mid \mathfrak{i} \in \mathbb{N} \text{ and } \varphi_\mathfrak{i} = \mathfrak{f}\}$ is infinite. ∎

15.3. Complexity

Next, we turn our attention to complexity. Using the setting of recursive functions, one can show several results that turn out to be very useful and insightful. So, we shall define abstract complexity measure and prove some fundamental results. However, due to the lack of time, more advanced results have been put into the Appendix.

Before defining abstract complexity measures, we make the following conventions.

The quantifiers '$\overset{\infty}{\forall}$' and '$\overset{\infty}{\exists}$' are interpreted as 'for all but finitely many' and 'there exists infinitely many, respectively. For any set $\mathsf{S} \subseteq \mathbb{N}$, by $\max \mathsf{S}$ and $\min \mathsf{S}$ we denote the maximum and minimum of a set S, respectively, where, by convention, $\max \emptyset = 0$ and $\min \emptyset = \infty$.

Definition 50 (Blum [1]).
Let φ be a Gödel numbering of $\mathcal{P}$ and let $\Phi \in \mathcal{P}^2$. $[\varphi, \Phi]$ is said to be a ***complexity measure*** *if*

(1) *$dom(\varphi_\mathfrak{i}) = dom(\Phi_\mathfrak{i})$ for all $\mathfrak{i} \in \mathbb{N}$, and*

(2) *There exist a recursive predicate M such that*

$$\forall \mathfrak{i} \forall \mathfrak{n} \forall \mathfrak{y} [\mathsf{M}(\mathfrak{i}, \mathfrak{n}, \mathfrak{y}) = 1 \iff \Phi_\mathfrak{i}(\mathfrak{n}) = \mathfrak{y}]$$

Note that Condition (1) and (2) are independent from one another. This can be seen as follows. Let φ be any Gödel numbering and define $\Phi_\mathfrak{i}(\mathfrak{n}) = 0$ for all $\mathfrak{i}, \mathfrak{n} \in \mathbb{N}$. Then $\Phi \in \mathcal{P}^2$ (in fact $\Phi \in \mathcal{R}^2$) and it obviously satisfies Condition (2). Of course, Condition (1) is violated.

Next consider

$$\Phi_\mathfrak{i}(\mathfrak{n}) = \begin{cases} 0, & \text{if } \varphi_\mathfrak{i}(\mathfrak{n}) \text{ is defined,} \\ \text{not defined,} & \text{otherwise.} \end{cases}$$

Now, Condition (1) is obviously fulfilled but Condition (2) is not satisfied.

A well-known example of a complexity measures is the number of steps required by the ith Turing machine (in a standard enumeration of all Turing machines) to converge on input x, i.e., the standard time measure. Another well-known example is the space measure, i.e., the number of squares containing non-blank symbols or visited by the read-write head during a computation by a Turing machine, provided the latter is considered undefined if the machine loops on a bounded tape segment.

Further complexity measures comprise

(a) ***reversal***, i.e. the number of times during the computation of the ith Turing machine that the head must change direction.

(b) ***ink***, i.e. the number of times during the computation of the ith Turing machine that a symbol has to be overwritten by a different symbol.

Exercise 51. *Prove or disprove that* ***carbon****, i.e., the number of times during the computation of the* i*th Turing machine that a symbol has to be overwritten by the same symbol is a complexity measure.*

Next, we establish several basic properties. The following theorem shows that the function values of all partial recursive functions are uniformly boundable by their complexities. But before we can state the result formally, we have to explain what does it mean that $\psi(x) = \theta(x)$ for partial functions ψ and θ and $x \in \mathbb{N}$. $\psi(x) = \theta(x)$ means that either both values $\psi(x)$ and $\theta(x)$ are defined and $\psi(x) = \theta(x)$ or else both values $\psi(x)$ and $\theta(x)$ are undefined. $\psi(x) \leqslant \theta(x)$ is then defined analogously.

Theorem 15.11. *Let* $[\varphi, \Phi]$ *be a complexity measure. Then there exists a function* $h \in \mathcal{R}^2$ *such that for all* $i \in \mathbb{N}$ *and all but finitely many* $n \in \mathbb{N}$ *we have* $\varphi_i(n) \leqslant h(n, \Phi_i(n))$.

Proof. We define the desired function h as follows. For all $i, n \in \mathbb{N}$, let

$$h(n, t) =_{df} \max\{\varphi_i(n) \mid i \leqslant n \ \wedge \ \Phi_i(n) = t\} .$$

Since the predicate $\Phi_i(n) = t$ is recursive, we directly obtain that $h \in \mathcal{R}^2$ (here we need our convention that $\max \emptyset = 0$).

It remains to show that h fulfills the desired property. First, suppose that $\varphi_i(n)$ is undefined. Then, by Condition (1) of the definition of a complexity measure, we also know that $\Phi_i(n)$ is undefined. Hence, the value $h(n, \Phi_i(n))$ is undefined, too, and thus the inequality $\varphi_i(n) \leqslant h(n, \Phi_i(n))$ is satisfied.

Next, assume $\varphi_i(n)$ to be defined. By the same argument as above we conclude that $\Phi_i(n)$ is defined, and thus so is $h(n, \Phi_i(n))$. Moreover, if $i \leqslant n$ then, by construction, we have $\varphi_i(n) \leqslant h(n, \Phi_i(n))$. Since there are only finitely many $n < i$, we are done. ∎

Intuitively speaking, Theorem 15.11 says that rapidly growing functions must be also "very" complex. This is of course obvious for a measure like space, since it takes a huge amount of time to write very large numbers down. It is, however, by no means obvious for complexity measures like reversal. So, maybe there is a deeper reason for this phenomenon. This is indeed the case, since all complexity measures can be recursively related to one another as our next theorem shows (cf. Blum [1]).

Theorem 15.12 (Recursive Relatedness of Complexity Measure). *Let* $[\varphi, \Phi]$ *and* $[\psi, \Psi]$ *be any two complexity measures. Furthermore, let* π *be a recursive permutation such that* $\varphi_i = \psi_{\pi(i)}$ *for all* $i \in \mathbb{N}$. *Then there exists a function* $h \in \mathcal{R}^2$ *such that*

$$\forall i \overset{\infty}{\forall} n \, [\, \Phi_i(n) \leqslant h(n, \Psi_{\pi(i)}(n)) \quad \text{and} \quad \Psi_{\pi(i)}(n) \leqslant h(n, \Phi_i(n)) \,] \; .$$

Proof. The desired function h is defined as follows. For all $n, t \in \mathbb{N}$ we set

$$h(n,t) = \max\{\Phi_i(n) + \Psi_{\pi(i)}(n) \mid i \leqslant n \wedge (\Phi_i(n) = t \vee \Psi_{\pi(i)}(n) = t)\} \; .$$

It remains to show that h fulfills the properties stated. Since $[\varphi, \Phi]$ and $[\psi, \Psi]$ are complexity measures, the predicates $\Phi_i(n) = t$ and $\Psi_{\pi(i)}(n) = t$ are both uniformly recursive in i, n, t. Moreover, by Condition (2) of Definition 50, if $\Phi_i(n) = t$ then, in particular, $\varphi_i(n)$ is defined. Because of $\varphi_i = \psi_{\pi(i)}$, we can conclude that $\psi_{\pi(i)}(n)$ is defined, too. Now another application of Condition (2) of Definition 50 directly yields that also $\Psi_{\pi(i)}(n)$ must be defined. Analogously it can be shown that $\Psi_{\pi(i)}(n) = t$ implies that $\Phi_i(n)$ is defined. Thus, if $\Phi_i(n) = t$ or $\Psi_{\pi(i)}(n) = t$ for some $i \leqslant n$, then $h(n,t)$ is defined. If neither $\Phi_i(n) = t$ nor $\Psi_{\pi(i)}(n) = t$ for all $i \leqslant n$, then we again the maximum of the empty set, i.e., in this case we have $h(n,t) = 0$. Hence, h is recursive.

Finally,

$$\forall i \overset{\infty}{\forall} n \, [\, \Phi_i(n) \leqslant h(n, \Psi_{\pi(i)}(n)) \quad \text{and} \quad \Psi_{\pi(i)}(n) \leqslant h(n, \Phi_i(n)) \,]$$

follows directly from our construction. We omit the details. ∎

With the following theorem we shall establish a fundamental basic result, i.e., we are going to prove that *no recursive amount* of computational resources is sufficient to compute *all* recursive functions. Furthermore, the proof below will use three basic proof techniques: *finite extension*, *diagonalization* and *cancellation*. The idea of finite extension is to construct a function by defining a finite piece at a time. Diagonalization is used to ensure that all functions not fulfilling a certain property must differ from our target function. And cancellation is a technique to keep track of all those programs we have already diagonalized against, and which ones we have yet to consider. Once a program i is diagonalized against, we shall cancel it. Canceled programs are never considered later for future diagonalization. Now, we are ready for our next theorem (cf. Blum [1]).

Theorem 15.13. *Let* $[\varphi, \Phi]$ *be any complexity measure. Then for every function* $h \in \mathcal{R}$ *there exists a function* $f \in \mathcal{R}_{0,1}$ *such that for all* φ*-programs* i *with* $\varphi_i = f$ *we have* $\Phi_i(n) > h(n)$ *for all but finitely many* $n \in \mathbb{N}$.

Proof. We shall use $\oplus$ as a symbol for addition modulo 2. Furthermore, we are going to define sets C_n in which we keep track of the programs already canceled. Finally, by μ we denote the well-known μ-operator. The desired function f is defined as follows.

$$f(0) = \begin{cases} \varphi_0(0) \oplus 1, & \text{if } \Phi_0(0) \leqslant h(0), \text{ then set } C_0 = \{0\} \\ 0, & \text{otherwise, then set } C_0 = \emptyset\ . \end{cases}$$

Now suppose, $f(0), \ldots, f(n)$ are already defined. We set

$$f(n+1) = \begin{cases} \varphi_{i^*}(n+1) \oplus 1, & \text{if } i^* = \mu i[i \leqslant n+1,\ i \notin C_n,\ \Phi_i(n+1) \leqslant h(n+1)] \\ & \text{exists, then set } C_{n+1} = C_n \cup \{i^*\} \\ 0, & \text{otherwise, then set } C_{n+1} = C_n\ . \end{cases}$$

It remains to show that f satisfies the stated properties. Since the predicate $\Phi_i(n) = y$ is uniformly recursive, so is the predicate $\Phi_i(n) \leqslant y$. Now, since $h \in \mathcal{R}$, we can effectively test whether or not $\Phi_0(0) \leqslant h(0)$. If it is, by Condition (1) of Definition 50 we can conclude that $\varphi_0(0)$ is defined. Hence, in this case $f(0)$ is defined and takes a value from $\{0, 1\}$. Otherwise, $f(0) = 0$, and thus again $f(0) \in \{0, 1\}$. Consequently, our initialization is recursive.

Using the same arguments as above and taking into account that we only have to check finitely many programs, it is obvious that the induction step is recursive, too. Moreover, by construction we again obtain $f(n+1) \in \{0, 1\}$. Therefore, $f \in \mathcal{R}_{0,1}$, and the first of f's properties is shown.

The remaining part is shown indirectly. Suppose there is program i such that $\varphi_i = f$ and

$$\overset{\infty}{\exists}\, n\ [\Phi_i(n) \leqslant h(n)]\ .$$

We set $C = \bigcup_{n \in \mathbb{N}} C_n$. By construction, it is easy to see that $i \notin C$, since otherwise $\varphi_i \neq f$ should hold.

Next, we consider $C^{(i)} =_{df} \{j \mid j < i \wedge j \in C\}$. Then we directly obtain $C^{(i)} \subseteq C$ and $\mathrm{card}(C^{(i)})$ is finite. Therefore, there must be an $m \geqslant i$ such that $C^{(i)} \subseteq C_n$ for all $n \geqslant m$. Furthermore, since there are infinitely many $n \in \mathbb{N}$ with $\Phi_i(n) \leqslant h(n)$ there exists an $n^* \geqslant m$ such that $\Phi_i(n^*) \leqslant h(n^*)$. But now

$$i = \mu j[j \leqslant n^* \wedge j \notin C_{n^*-1} \wedge \Phi_i(n^*) \leqslant h(n^*)]\ .$$

Hence, we have to cancel i when constructing $f(n^*)$, and thus $i \in C_{n^*} \subseteq C$, a contradiction. Consequently, for all but finitely many $n \in \mathbb{N}$

$$\Phi_i(n) > h(n)$$

must hold. Since i was any program for f, we are done. ∎

Theorem 15.11 showed that we can recursively bound the function values of all partial recursive functions by their complexities. Thus, it is only natural to ask whether or not we can do the converse, i.e., bound the complexity values of all partial recursive functions by their function values. The negative answer is provided next.

Theorem 15.14. *Let* $[\varphi, \Phi]$ *be any complexity measure. Then there is no recursive function* $h \in \mathcal{R}^2$ *such that*

$$\forall i \overset{\infty}{\forall} n[\Phi_i(n) \leqslant h(n, \varphi_i(n))] .$$

Proof. Suppose the converse, i.e., there is a function $h \in \mathcal{R}^2$ such that

$$\forall i \overset{\infty}{\forall} n[\Phi_i(n) \leqslant h(n, \varphi_i(n))] .$$

Consider the recursive function h^* defined as

$$h^*(n) = h(n, 0) + h(n, 1)$$

for all $n \in \mathbb{N}$. Then, by Theorem 15.13 there exists a function $f \in \mathcal{R}_{0,1}$ such that

$$\forall i[\varphi_i = f \longrightarrow \overset{\infty}{\forall} n[\Phi_i(n) > h^*(n)]] .$$

Consequently,

$$\overset{\infty}{\forall} n[\Phi_i(n) > h^*(n) = h(n, 0) + h(n, 1) \geqslant h(n, \varphi_i(n))]$$

for every program i with $\varphi_i = f$, a contradiction to the choice of h. ∎

Next, we ask a slight modification of the question posed before Theorem 15.14, i.e., are there functions $h \in \mathcal{R}^2$ such that there are also functions $f \in \mathcal{P}$ satisfying

$$\exists i[\varphi_i = f \wedge \overset{\infty}{\forall} n[\Phi_i(n) \leqslant h(n, \varphi_i(n))]] ?$$

Before providing the affirmative answer, we make the following definition.

Definition 51. *Let* $[\varphi, \Phi]$ *be a complexity measure, let* $h \in \mathcal{R}^2$ *and let* $\psi \in \mathcal{P}$. *Function* ψ *is said to be* ***h-honest*** *if*

$$\exists i[\varphi_i = \psi \wedge \overset{\infty}{\forall} n[\Phi_i(n) \leqslant h(n, \varphi_i(n))]]$$

.

By $H(h)$ we denote the set of all h-honest functions from $\mathcal{P}$. Now, we are ready to answer the latter question.

Theorem 15.15. *Let* $[\varphi, \Phi]$ *be a complexity measure and let* $\Psi \in \mathcal{P}^2$ *be any function such that the predicate* $\Psi_i(x) = y$ *is uniformly recursive. Then there exists a function* $h \in \mathcal{R}^2$ *such that* $\Psi_i \in H(h)$ *for all* $i \in \mathbb{N}$.

Proof. Since φ is a Gödel numbering of $\mathcal{P}$ and $\Psi \in \mathcal{P}^2$, there exists a function $c \in \mathcal{R}$ such that $\Psi_i = \varphi_{c(i)}$ for all $i \in \mathbb{N}$. Moreover, by assumption the predicate $\Psi_i(x) = y$ is uniformly recursive. Therefore, we can define a recursive function $h \in \mathcal{R}^2$ as follows:

$$h(n,t) = \max\{\Phi_{c(i)}(n) \mid i \leqslant n \ \wedge \ \Psi_i(n) = t\} \ .$$

Finally, consider any Ψ_i. Then we have for the φ-program $c(i)$ of Ψ_i the following:

$$\Psi_i = \varphi_{c(i)} \wedge \overset{\infty}{\forall} n[\Phi_{c(i)}(n) \leqslant h(n, \varphi_{c(i)}(n))]$$

by construction. ∎

The latter theorem immediately allows the following corollary.

Corollary 15.16. *Let* $[\varphi, \Phi]$ *be a complexity measure. Then there exists a function* $h \in \mathcal{R}^2$ *such that* $\{\Phi_i \mid i \in \mathbb{N}\} \subseteq H(h)$.

Furthermore, Corollary 15.16 and Theorem 15.14 together directly imply the following set theoretical properties of $S = \{\Phi_i \mid i \in \mathbb{N}\}$.

Corollary 15.17. *Let* $[\varphi, \Phi]$ *be a complexity measure. Then we have*

(1) $S \subset \mathcal{P}$,

(2) $S \cap (\mathcal{P} \setminus \mathcal{R}) \subset \mathcal{P} \setminus \mathcal{R}$, *and*

(3) $S \cap \mathcal{R} \subset \mathcal{R}$.

Finally, it should be noted that it is not meaningful to consider the following stronger version of h-honest functions ψ

$$\forall i[\varphi_i = \psi \wedge \overset{\infty}{\forall} n[\Phi_i(n) \leqslant h(n, \varphi_i(n))]] \ .$$

This is due to the fact, well-known to everybody who has already written a couple of programs, that there are arbitrarily bad ways of computing any function with respect to any complexity measure. More formally, we have the following theorem (cf. Blum [1]).

Theorem 15.18. *Let* $[\varphi, \Phi]$ *be any complexity measure. Then, for every program* i *and every recursive function* $h \in \mathcal{R}$ *there exists a program* e *such that* $\varphi_e = \varphi_i$ *and* $\Phi_e(x) > h(x)$ *for all* $x \in dom(\varphi_i)$.

Proof. Let $g \in \mathcal{R}$ be any function such that for all $i, n \in \mathbb{N}$

$$\varphi_{g(i)}(n) = \begin{cases} \varphi_i(n), & \text{if } \neg[\Phi_i(n) \leqslant h(n)] \\ \varphi_i(n) + 1, & \text{if } \Phi_i(n) \leqslant h(n) \ . \end{cases}$$

By the fixed point theorem, there exists an e such that $\varphi_{g(e)} = \varphi_e$. It remains to show that $\varphi_e = \varphi_i$ and $\Phi_e(x) > h(x)$ for all $x \in dom(\varphi_i)$.

Suppose there is an $x \in dom(\varphi_i)$ such that $\Phi_e(x) \leqslant h(x)$. By Condition (1) of Definition 50 we conclude that $\varphi_e(x)$ must be defined. Since $\varphi_{g(e)} = \varphi_e$ and because of $\Phi_e(x) \leqslant h(x)$ we then have $\varphi_e(x) = \varphi_{g(e)}(x) = \varphi_e(x)+1$. Thus, the case $\Phi_e(x) \leqslant h(x)$ cannot happen. Consequently, for all $x \in dom(\varphi_i)$ the case $\Phi_e(x) > h(x)$ must occur, and thus, by construction $\varphi_e(x) = \varphi_i(x)$. But if $x \notin dom(\varphi_i)$, it also holds $\neg[\Phi_e(x) \leqslant h(x)]$ and therefore again $\varphi_e(x) = \varphi_i(x)$. This implies $\varphi_i = \varphi_e$ and completes the proof. ∎

APPENDIX

Now, we turn our attention to some more advanced topics. We start with complexity classes.

16.1. Complexity Classes

Probably, you have already heard a bit about some complexity classes such as the class of all problems decidable in deterministic polynomial time, the class of all problems decidable in non-deterministic polynomial time, the class of all languages acceptable in deterministic logarithmic space the class of all languages acceptable in non-deterministic logarithmic space and the class of all languages acceptable in polynomial space.

For the time being, we want to take a more abstract view of complexity classes such as the ones mentioned above and study properties they have or do not have in common. Therefore, we continue by defining abstract complexity classes (cf. [3]).

Definition 52. *Let* $[\varphi, \Phi]$ *be a complexity measure and let* $t \in \mathcal{R}$. *Then we set*

$$\mathcal{R}_t^{[\varphi,\Phi]} =_{df} \{f \mid f \in \mathcal{R} \wedge \exists i[\varphi_i = f \wedge \overset{\infty}{\forall} n\ \Phi_i(n) \leqslant t(n)]\} .$$

and call $\mathcal{R}_t^{[\varphi,\Phi]}$ *the* ***complexity class*** *generated by* t.

First, we deal with the structure of the classes $\mathcal{R}_t^{[\varphi,\Phi]}$. For this purpose, let us recall the common two definitions of enumerability.

Definition 53. *Let* $\mathcal{U} \subseteq \mathcal{R}$ *be any class of recursive functions.* $\mathcal{U}$ *is said to be recursively enumerable if there exists a function* $g \in \mathcal{R}^2$ *such that* $\mathcal{U} \subseteq \{\lambda x g(i, x) \mid i \in \mathbb{N}\}$.

If a class $\mathcal{U} \subseteq \mathcal{R}$ is recursively enumerable, we also write $\mathcal{U} \in$ NUM, i.e., we use NUM to denote the collection of all classes $\mathcal{U} \subseteq \mathcal{R}$ that are recursively enumerable.

Definition 54. *Let* $\mathcal{U} \subseteq \mathcal{R}$ *be any class of recursive functions.* $\mathcal{U}$ *is said to be sharply recursively enumerable if there exists a function* $g \in \mathcal{R}^2$ *such that* $\mathcal{U} = \{\lambda x g(i, x) \mid i \in \mathbb{N}\}$.

If a class $\mathcal{U} \subseteq \mathcal{R}$ is sharply recursively enumerable, we also write $\mathcal{U} \in \mathrm{NUM}!$, i.e., we use NUM! to denote the collection of all classes $\mathcal{U} \subseteq \mathcal{R}$ that are sharply recursively enumerable.

Exercise 52. *Prove the following definitions to be equivalent to Definition 53 and Definition 54, respectively. Let φ be any Gödel numbering; then we have:*

(1) $\mathcal{U} \in \mathrm{NUM}$ *if and only if there exists a function* $h \in \mathcal{R}$ *such that*

$$\mathcal{U} \subseteq \{\varphi_{h(i)} \mid i \in \mathbb{N}\} \subseteq \mathcal{R} .$$

(2) $\mathcal{U} \in \mathrm{NUM}!$ *if and only if there exists a function* $h \in \mathcal{R}$ *such that*

$$\mathcal{U} = \{\varphi_{h(i)} \mid i \in \mathbb{N}\} \subseteq \mathcal{R} .$$

The following theorem shows in particular that every class $\mathcal{U} \subseteq \mathcal{R}$ can be embedded into a complexity class.

Theorem 16.1. *Let $[\varphi, \Phi]$ be a complexity measure and let $\mathcal{U} \subseteq \mathcal{R}$ be any class of recursive functions such that $\mathcal{U} \in \mathrm{NUM}$. Then, we have:*

(1) *There exists a function $b \in \mathcal{R}$ such that for all $f \in \mathcal{U}$ the condition $f(n) \leqslant b(n)$ for all but finitely many $n \in \mathbb{N}$ is satisfied.*

(2) *There exists a function $t \in \mathcal{R}$ such that $\mathcal{U} \subseteq \mathcal{R}_t^{[\varphi,\Phi]}$.*

Proof. Since $\mathcal{U} \in \mathrm{NUM}$ there is a function $g \in \mathcal{R}^2$ such that $\mathcal{U} \subseteq \{\lambda x g(i,x) \mid i \in \mathbb{N}\}$. We therefore define the desired function b as follows: For all $n \in \mathbb{N}$ let

$$b(n) =_{df} \max\{g(i,n) \mid i \leqslant n\} .$$

Then b obviously satisfies the properties stated in (1).

For showing Condition (2) it suffices to apply Exercise 52, i.e., since $\mathcal{U} \in \mathrm{NUM}$ there is $h \in \mathcal{R}$ such that $\mathcal{U} \subseteq \{\varphi_{h(i)} \mid i \in \mathbb{N}\} \subseteq \mathcal{R}$. Therefore, for all $n \in \mathbb{N}$ we can define

$$t(n) =_{df} \max\{\Phi_{h(i)}(n) \mid i \leqslant n\} .$$

By the choice of h we know that $\varphi_{h(i)} \in \mathcal{R}$ for all $i \in \mathbb{N}$. Using Condition (1) of the definition of a complexity measure, we can conclude $\Phi_{h(i)} \in \mathcal{R}$ for all $i \in \mathbb{N}$, too. Thus, our function t defined above satisfies $t \in \mathcal{R}$. Moreover, by construction we directly obtain

$$\forall i \overset{\infty}{\forall} n[\Phi_{h(i)}(n) \leqslant t(n)] ,$$

and therefore $\mathcal{U} \subseteq \mathcal{R}_t^{[\varphi,\Phi]}$. ∎

16.2. Recursive Enumerability of Complexity Classes

Now, let $[\varphi, \Phi]$ be any complexity measure. We are interested in learning for which $t \in \mathcal{R}$ the complexity class $\mathcal{R}_t^{[\varphi,\Phi]}$ is recursively enumerable. This would be a nice property, since $\mathcal{R}_t^{[\varphi,\Phi]} \in \mathrm{NUM}$ would give us an effective overview about at least one program for every function $f \in \mathcal{R}_t^{[\varphi,\Phi]}$. We are going to show that $\mathcal{R}_t^{[\varphi,\Phi]} \in \mathrm{NUM}$ for all functions $t \in \mathcal{R}$ and every complexity measure $[\varphi, \Phi]$. On the other hand, there are complexity measures $[\varphi, \Phi]$ and functions $t \in \mathcal{R}$ such that $\mathcal{R}_t^{[\varphi,\Phi]} \notin \mathrm{NUM}$!, as we shall see.

Theorem 16.2. *Let $[\varphi, \Phi]$ be any complexity measure. Then, for every function $t \in \mathcal{R}$ we have $\mathcal{R}_t^{[\varphi,\Phi]} \in \mathrm{NUM}$.*

Proof. We start our proof by asking under which conditions a function f belongs to $\mathcal{R}_t^{[\varphi,\Phi]}$. By the definition of a complexity class, this is the case if and only if there exists a φ-program k computing f such that the complexity Φ_k fulfills the condition

$$\overset{\infty}{\forall}\, n[\Phi_k(n) \leqslant t(n)] \,.$$

But this condition means nothing else than there exist a $\tau \in \mathbb{N}$ and an n_0 such that $\Phi_k(n) \leqslant \tau$ for all $n < n_0$ and $\Phi_k(n) \leqslant t(n)$ for all $n \geqslant n_0$.

Therefore, we choose an effective enumeration of all triples of natural numbers. For the sake of presentation, let c_1, c_2, $c_3 \in \mathcal{R}$ such that

$$\mathbb{N} \times \mathbb{N} \times \mathbb{N} = \{[c_1(i), c_2(i), c_3(i)] \mid i \in \mathbb{N}\} \,.$$

Now, we are ready to define the wanted function $g \in \mathcal{R}^2$ as follows. For all $i, n \in \mathbb{N}$ let

$$g(i,n) = \begin{cases} \varphi_{c_1(i)}(n), & \text{if } [n < c_2(i) \ \wedge\ \Phi_{c_1(i)}(n) \leqslant c_3(i)] \\ & \quad \vee\, [n \geqslant c_2(i) \ \wedge\ \Phi_{c_1(i)}(n) \leqslant t(n)] \\ 0, & \text{otherwise} \,. \end{cases}$$

It remains to show that $\mathcal{R}_t^{[\varphi,\Phi]} \subseteq \{\lambda n g(i,n) \mid i \in \mathbb{N}\}$. Let $f \in \mathcal{R}_t^{[\varphi,\Phi]}$; then there exists a triple $[k, n_0, \tau]$ such that

(1) $\varphi_k = f$,

(2) $\forall n < n_0[\Phi_k(n) \leqslant \tau]$,

(3) $\forall n \geqslant n_0[\Phi_k(n) \leqslant t(n)]$.

Hence, there must exist an $i \in \mathbb{N}$ such that $[c_1(i), c_2(i), c_3(i)] = [k, n_0, \tau]$.

But now, by construction we can immediately conclude that $\lambda n g(i,n) = \varphi_k = f$, and thus, by (1) through (3), the theorem follows. ∎

The proof above also shows why it may be difficult to prove $\mathcal{R}_t^{[\varphi,\Phi]} \in \mathrm{NUM}!$, i.e., we have defined $g(i,n) = 0$ in case the stated condition is not satisfied. Since we have to ensure $g \in \mathcal{R}^2$, we have to define $g(i,n)$ for all those i,n for which the stated condition is not fulfilled somehow, thus running into the danger to include functions into our enumeration that are *not* members of the complexity class $\mathcal{R}_t^{[\varphi,\Phi]}$. But if we have a bit prior knowledge about $\mathcal{R}_t^{[\varphi,\Phi]}$, then we can show $\mathcal{R}_t^{[\varphi,\Phi]} \in \mathrm{NUM}!$.

Therefore, we set

$$\mathcal{U}_0 = \{f \in \mathcal{R} \mid f(n) = 0 \text{ for all but finitely } n\} ,$$

i.e., $\mathcal{U}_0$ is the class of functions of ***finite support***. Moreover, for any $f \in \mathcal{R}$ we let

$$\mathcal{U}_f = \{\hat{f} \in \mathcal{R} \mid \hat{f}(n) = f(n) \text{ for all but finitely } n\} ,$$

i.e., $\mathcal{U}_f$ is the class of all finite variations of function f.

Theorem 16.3. *Let $[\varphi,\Phi]$ be any complexity measure and let $t \in \mathcal{R}$. Then we have:*

(1) *If $\mathcal{U}_0 \subseteq \mathcal{R}_t^{[\varphi,\Phi]}$ then $\mathcal{R}_t^{[\varphi,\Phi]} \in \mathrm{NUM}!$.*

(2) *If $\mathcal{U}_f \subseteq \mathcal{R}_t^{[\varphi,\Phi]}$ for some $f \in \mathcal{R}$ then $\mathcal{R}_t^{[\varphi,\Phi]} \in \mathrm{NUM}!$.*

Proof. Though the proof is conceptually very similar to the proof of Theorem 16.2, an important modification is necessary, since we have to ensure that $g \in \mathcal{R}^2$ enumerates exclusively functions belonging to $\mathcal{R}_t^{[\varphi,\Phi]}$. First, we show Assertion (1). Let c_1, c_2, $c_3 \in \mathcal{R}$ be as in the proof of Theorem 16.2. Now, we define the desired function $g \in \mathcal{R}^2$ as follows. For all $i,n \in \mathbb{N}$ let

$$g(i,n) = \begin{cases} \varphi_{c_1(i)}(n), & \text{if } [\forall x[x < c_2(i) \longrightarrow \Phi_{c_1(i)}(x) \leqslant c_3(i)] \wedge \\ & \quad \forall x[c_2(i) \leqslant x \leqslant n \longrightarrow \Phi_{c_1(i)}(x) \leqslant t(x)]] \\ 0, & \text{otherwise .} \end{cases}$$

At this point, one easily verifies that for all $i \in \mathbb{N}$ the condition $\lambda n g(i,n) = \varphi_{c_1(i)}$ or $\overset{\infty}{\forall}\, n g(i,n) = 0$ is satisfied. Consequently, now Assertion (1) follows as in the proof of Theorem 16.2. We omit the details.

Assertion (2) is proved *mutatis mutandis*, i.e., we define for all $i,n \in \mathbb{N}$

$$g(i,n) = \begin{cases} \varphi_{c_1(i)}(n), & \text{if } [\forall x[x < c_2(i) \longrightarrow \Phi_{c_1(i)}(x) \leqslant c_3(i)] \wedge \\ & \quad \forall x[c_2(i) \leqslant x \leqslant n \longrightarrow \Phi_{c_1(i)}(x) \leqslant t(x)]] \\ f(n), & \text{otherwise .} \end{cases}$$

∎

Next, we can show the following general result.

Theorem 16.4. *Let* $[\varphi, \Phi]$ *be any complexity measure. Then there exists a function* $t \in \mathcal{R}$ *such that for all functions* $\tilde{t}$ *satisfying* $\tilde{t}(n) \geqslant t(n)$ *for all but finitely* n *we have* $\mathcal{R}_{\tilde{t}}^{[\varphi,\Phi]} \in \text{NUM!}$.

Proof. The proof is done via the following claim.

Claim. Let $[\varphi, \Phi]$ *be any complexity measure. Then there exists a function* $\hat{t} \in \mathcal{R}$ *such that* $U_0 \subseteq \mathcal{R}_{\hat{t}}^{[\varphi,\Phi]}$.

First, recall that $U_0 \in \text{NUM}$, since all finite tuples of natural numbers are recursively enumerable. Thus, we can apply Theorem 16.1. Consequently, there is a function $\hat{t} \in \mathcal{R}$ such that $U_0 \subseteq \mathcal{R}_{\hat{t}}^{[\varphi,\Phi]}$. This proves the claim.

Now, we set $t = \hat{t}$. Consequently, we have $U_0 \subseteq \mathcal{R}_{t}^{[\varphi,\Phi]}$. Finally, assume any function $\tilde{t}$ satisfying $\tilde{t}(n) \geqslant t(n)$ for all but finitely n. By the definition of complexity classes, we directly get $\mathcal{R}_{t}^{[\varphi,\Phi]} \subseteq \mathcal{R}_{\tilde{t}}^{[\varphi,\Phi]}$, and therefore we have $U_0 \subseteq \mathcal{R}_{\tilde{t}}^{[\varphi,\Phi]}$, too. Thus, by Theorem 16.3 we arrive at $\mathcal{R}_{\tilde{t}}^{[\varphi,\Phi]} \in \text{NUM!}$. ∎

The latter theorem allows a nice corollary. Consider all 3-tape Turing machines (with input-tape, work-tape, and output tape) and let $\tilde{\varphi}$ be the canonical Gödel numbering of all these 3-tape Turing machines. Moreover, we let Φ_i be the number of cells used by φ_i on its work-tape. Thus, $[\tilde{\varphi}, \tilde{\Phi}]$ is a complexity measure.

Corollary 16.5. *Let* $[\tilde{\varphi}, \tilde{\Phi}]$ *be the complexity measure defined above. Then* $\mathcal{R}_{t}^{[\tilde{\varphi},\tilde{\Phi}]} \in \text{NUM!}$ *for all* $t \in \mathcal{R}$.

Proof. The corollary immediately follows from Theorem 16.4, since $U_0 \subseteq \mathcal{R}_{t}^{[\tilde{\varphi},\tilde{\Phi}]}$ for all $t \in \mathcal{R}$, i.e, in particular for $t(n) = 0$ for all n. ∎

Next we show that the theorems concerning the recursive enumerability which we have obtained in the last lecture, cannot be improved. That is, we are going to show that there are complexity measures and functions $t \in \mathcal{R}$ such that $\mathcal{R}_{t}^{[\varphi,\Phi]} \notin \text{NUM!}$. For the sake of presentation, it will be convenient to identify a function with the sequence of its values. For example, we shall use 0^∞ to denote the everywhere zero function t, i.e., $t(n) = 0$ for all $n \in \mathbb{N}$.

Theorem 16.6. *There are a complexity measure and a function* $t \in \mathcal{R}$ *such that* $\mathcal{R}_{t}^{[\varphi,\Phi]} \notin \text{NUM!}$.

Proof. Since the theorem can only hold for small functions $t \in \mathcal{R}$, we choose the smallest possible one, i.e., $t(n) = 0$ for all $n \in \mathbb{N}$. Now, let $[\varphi, \Phi']$ be any complexity measure. Furthermore, let $h \in \mathcal{R}$ be a function such that

$$\varphi_{h(i)}(n) = i \text{ for all } i \in \mathbb{N} \text{ and all } n \in \mathbb{N} .$$

That is, $\varphi_{h(i)} = i^\infty$. Without loss of generality, by the padding lemma, we can assume h to strictly monotone. Thus, *range*(h) is decidable.

Now, the proof idea is easily explained. Let K be the halting set, i.e.,

$$K = \{i \mid \varphi_i(i) \text{ is defined } \} .$$

Recall that K is recursively enumerable but its complement $\overline{K}$ is not, since otherwise K would be decidable.

We are going to construct a complexity measure $[\varphi, \Phi]$ such that

$$\mathcal{R}_{0^\infty}^{[\varphi,\Phi]} = \{i^\infty \mid i \in \overline{K}\} .$$

This is all we need. For seeing this, assume we have already shown Property 15.3 to hold. Suppose to the contrary that $\mathcal{R}_{0^\infty}^{[\varphi,\Phi]} \in$ NUM!. Then there must be a function $g \in \mathcal{R}^2$ such that

$$\mathcal{R}_{0^\infty}^{[\varphi,\Phi]} = \{\lambda x g(j,x) \mid j \in \mathbb{N}\} .$$

The latter property directly implies that

$$\{g(j,0) \mid j \in \mathbb{N}\} = \overline{K} ,$$

i.e., $\overline{K}$ would be recursively enumerable, a contradiction.

Hence, it indeed suffices to show Property 15.3 which is done next. We set for all $j \in \mathbb{N}$ and all $n \in \mathbb{N}$

$$\Phi_j(n) =_{df} \begin{cases} 0, & \text{if } j = h(i) \text{ and } \neg[\Phi'_i(i) \leqslant n], \\ 1 + \Phi'_j(n), & \text{if } j = h(i) \text{ and } \Phi'_i(i) \leqslant n, \\ 1 + \Phi'_j(n), & \text{if } j \notin \mathit{range}(h) . \end{cases}$$

Note that in the first and second case $j \in \mathit{range}(h)$. Since h is strictly monotone, there can be at most one such i satisfying $j = h(i)$. In the third case, we assume $\Phi_j(n)$ to be not defined if and only if $\Phi'_j(n)$ is not defined.

Now, it is easy to see that $[\varphi, \Phi]$ is a complexity measure. Since we did not change the Gödel numbering, our construction directly implies that Condition (1) of the definition of complexity measure remains valid.

Condition (2) of Definition 50 is also satisfied. In order to decide whether or not "$\Phi_j(n) = y$," we have to check if $j \in \mathit{range}(h)$. This is decidable. If it is not and $y = 0$ then the answer is "no." If $j \notin \mathit{range}(h)$ and $y > 0$, we output $M'(j, n, y-1)$, where M' is the predicate for $[\varphi, \Phi']$. Moreover, if $j \in \mathit{range}(h)$, then let $j = h(i)$. We then first check if $M'(i, i, k) = 0$ for all $k = 0, \ldots, n$. If this true and $y = 0$, we output 1. If $M'(i, i, k) = 0$ for all $k = 0, \ldots, n$ and $y > 0$, we output 0. Otherwise, there is a $k \leqslant n$ such that $M'(i, i, k) = 1$. Again, if $y = 0$, we output 0, and if $y > 0$ we output $M'(j, n, y-1)$.

Finally, it is easy to see that for all $j \in \mathbb{N}$ our construction directly yields $\Phi_j(n) > 0$ for all but finitely many n or $\Phi_j(n) = 0$ for all $n \in \mathbb{N}$. But the latter case happens if and only if

$$\exists i[j = h(i) \ \wedge \ \varphi_i(i) \text{ not defined }] ,$$

i.e.,

$$\mathcal{R}_{0^\infty}^{[\varphi,\Phi]} = \{i^\infty \mid i \in \overline{K}\} .$$

This proves the theorem. ∎

16.3. An Undecidability Result

As we have seen, it can happen that $\mathcal{R}_t^{[\varphi,\Phi]} \notin \text{NUM!}$. Thus, it is only natural to ask if one can decide, for any given complexity measure $[\varphi, \Phi]$ and any given function $t \in \mathcal{R}$ whether or not $\mathcal{R}_t^{[\varphi,\Phi]} \in \text{NUM!}$. Our next theorem shows that there is such a decision procedure if and only if it is not necessary, i.e., if and only if $\mathcal{R}_t^{[\varphi,\Phi]} \in \text{NUM!}$ for all $t \in \mathcal{R}$. Thus, conceptually, the following theorem should remind you to the Theorem of Rice.

Theorem 16.7. *Let $[\varphi, \Phi]$ be any complexity measure. Then we have:*

$$\exists \psi \in \mathcal{P}\, \forall j[\varphi_j \in \mathcal{R} \longrightarrow \psi(j) \text{ defined } \wedge \left(\mathcal{R}_{\varphi_j}^{[\varphi,\Phi]} \in \text{NUM!} \longrightarrow \psi(j) = 1\right) \wedge \left(\mathcal{R}_{\varphi_j}^{[\varphi,\Phi]} \notin \text{NUM!} \longrightarrow \psi(j) = 0\right) \leftrightarrow \forall t \in \mathcal{R}\left[\mathcal{R}_t^{[\varphi,\Phi]} \in \text{NUM!}\right]$$

Proof. The sufficiency is obvious, since we can set $\psi = 1^\infty$.

Necessity. We set

$$\mathcal{R}^+ =_{df} \{t \mid t \in \mathcal{R} \wedge \mathcal{R}_t^{[\varphi,\Phi]} \in \text{NUM!}\}$$

as well as $\mathcal{R}^- =_{df} \mathcal{R} \setminus \mathcal{R}^+$. As we have already seen, $\mathcal{R}^+ \neq \emptyset$ (cf. Theorem 16.4). Moreover, if $\mathcal{R}^- = \emptyset$, then we are already done. Thus, suppose $\mathcal{R}^- \neq \emptyset$. We continue by showing that such a desired ψ cannot exist. For this purpose, let $t^+ \in \mathcal{R}^+$ and $t^- \in \mathcal{R}^-$. Furthermore, for any $f \in \mathcal{R}$ we again set

$$\mathcal{U}_f =_{df} \{f' \mid \varphi' \in \mathcal{R} \wedge \overset{\infty}{\forall} n f'(n) = f(n)\}\,.$$

Now, suppose to the contrary that there is a $\psi \in \mathcal{P}$ such that

$$\forall j[\varphi_j \in \mathcal{R} \longrightarrow \psi(j) \text{ defined } \wedge \left(\mathcal{R}_{\varphi_j}^{[\varphi,\Phi]} \in \text{NUM!} \longrightarrow \psi(j) = 1\right) \wedge \left(\mathcal{R}_{\varphi_j}^{[\varphi,\Phi]} \notin \text{NUM!} \longrightarrow \psi(j) = 0\right)$$

Then, this function ψ should in particular satisfy the following

$$\forall j[\varphi_j \in \mathcal{U}_{t^+} \longrightarrow \psi(j) = 1] \wedge \forall j[\varphi_j \in \mathcal{U}_{t^-} \longrightarrow \psi(j) = 0]\,.$$

Next, let k be arbitrarily fixed such the $\varphi_k = \psi$. Furthermore, let $f \in \mathcal{R}$ be chosen such that for all $j, n \in \mathbb{N}$

$$\varphi_{f(j)}(n) = \begin{cases} t^+(n), & \text{if } \Phi_k(j) \leqslant n \wedge \psi(j) = 0 \\ t^-(n), & \text{if } \Phi_k(j) \leqslant n \wedge \psi(j) = 1 \\ 0, & \text{otherwise}\,. \end{cases}$$

By the choice of k, if $\Phi_k(j) \leqslant n$, then we can compute $\varphi_k(j) = \psi(j)$. Therefore, $\varphi_{f(j)} \in \mathcal{R}$ for all $j \in \mathbb{N}$. By the fixed point theorem, there is a number a such that

$\varphi_{f(a)} = \varphi_a$. Thus, $\varphi_a \in \mathcal{R}$. By assumption, we can conclude that $\psi(a)$ must be defined. Now, we distinguish the following cases.

Case 1. $\psi(a) = 0$.

Then, by construction, $\varphi_{f(a)} \in \mathcal{U}_{t^+}$. Since $\varphi_{f(a)} = \varphi_a$, we thus obtain $\varphi_a \in \mathcal{U}_{t^+}$. But this implies $\mathcal{R}_{\varphi_a}^{[\varphi,\Phi]} \in$ NUM!, and consequently $\psi(a) = 1$, a contradiction. Thus, Case 1 cannot happen.

Case 2. $\psi(a) = 1$.

Then, by construction, $\varphi_{f(a)} \in \mathcal{U}_{t^-}$ and thus $\varphi_a \in \mathcal{U}_{t^-}$. By the choice of t^- we can conclude $\mathcal{R}_{\varphi_a}^{[\varphi,\Phi]} \notin$ NUM!, and consequently $\psi(a) = 0$, a contradiction. Thus, this case cannot happen either. Hence, the desired function ψ cannot exist. ∎

Further results concerning recursive properties of abstract complexity classes can be found in Landweber and Robertson [4] as well as in McCreight and Meyer [3] and the references therein.

Next, we turn our attention to a different problem. We are going to ask how much resources must be added to some given resources in order to make more functions computable than before. That is, given any $t \in \mathcal{R}$, we are interested in learning how to choose $\hat{t}$ such that

$$\mathcal{R}_t^{[\varphi,\Phi]} \subset \mathcal{R}_{\hat{t}}^{[\varphi,\Phi]} .$$

Before we are trying to answer this question in more depth, we confine ourselves that there is always such a function $\hat{t}$.

Lemma 16.8. *Let $[\varphi, \Phi]$ be any complexity measure. For every recursive function $t \in \mathcal{R}$ there is always another function $\hat{t}$ such that $\mathcal{R}_t^{[\varphi,\Phi]} \subset \mathcal{R}_{\hat{t}}^{[\varphi,\Phi]}$.*

Proof. Let $t \in \mathcal{R}$ be arbitrarily fixed. By Theorem 15.13, there exists a function $f \in \mathcal{R}_{0,1}$ such that for all φ-programs i with $\varphi_i = f$ we have $\Phi_i(n) > t(n)$ for all but finitely many $n \in \mathbb{N}$. Therefore, we get $f \notin \mathcal{R}_t^{[\varphi,\Phi]}$.

Let i be any φ-program for f. We define for all $n \in \mathbb{N}$

$$\hat{t}(n) = \max\{t(n),\, \Phi_i(n)\} .$$

Consequently, $f \in \mathcal{R}_{\hat{t}}^{[\varphi,\Phi]}$ and $\mathcal{R}_t^{[\varphi,\Phi]} \subset \mathcal{R}_{\hat{t}}^{[\varphi,\Phi]}$. ∎

The latter lemma directly implies the following corollary.

Corollary 16.9. *Let $[\varphi, \Phi]$ be any complexity measure. There is no function $t \in \mathcal{R}$ such that $\mathcal{R}_t^{[\varphi,\Phi]} = \mathcal{R}$.*

So, the more interesting question is whether or not we can choose $\hat{t}$ in dependence of t effectively in order to obtain

$$\mathcal{R}_t^{[\varphi,\Phi]} \subset \mathcal{R}_{\hat{t}}^{[\varphi,\Phi]} .$$

This is done in the next part of our lecture.

16.4. The Gap-Theorem

Let us try the following approach. For the sake of presentation, for any $h \in \mathcal{R}^2$ we shall write $h \circ t$ to denote the function

$$(h \circ t)(n) =_{df} h(n, t(n))$$

for all $n \in \mathbb{N}$. Now, we can imagine that $h \in \mathcal{R}^2$ is a very rapidly growing function. Thus, it is quite natural to ask whether or not we may expect

$$\mathcal{R}_t^{[\varphi,\Phi]} \subset \mathcal{R}_{h\circ t}^{[\varphi,\Phi]} \ .$$

for all $t \in \mathcal{R}$. The surprising answer is no as the following theorem shows. Since this theorem establishes a "gap" in which nothing more can be computed than before, it is called Gap-Theorem. The Gap-Theorem has been discovered by Trakhtenbrot [6] and later independently by Borodin [2]. The proof given below, however, follows Young [7].

Theorem 16.10 (*Gap-Theorem*).
Let $[\varphi, \Phi]$ *be any complexity measure. Then we have*

$$\forall h \in \mathcal{R}^2 \left[\forall n \forall y [h(n, y) \geqslant y \longrightarrow \exists t \in \mathcal{R} \left[\mathcal{R}_t^{[\varphi,\Phi]} = \mathcal{R}_{h\circ t}^{[\varphi,\Phi]}\right]\right] \ .$$

Proof. We shall even show a somehow stronger result, i.e., that t can made arbitrarily large. Let $a \in \mathcal{R}$ be any function. We are going to construct t such that

(1) $t(n) \geqslant a(n)$ for all but finitely many n.

(2) For all $n > j$, if $\Phi_j(n) > t(n)$ then $\Phi_j(n) > h(n, t(n))$.

To define t we set

$$t_{n+1} = a(n) \text{ and for } 0 < i \leqslant n+1, \text{ let } t_{i-1} = h(n, t_i) + 1 \ .$$

Thus, we directly get from $h(n, y) \geqslant y$ that

$$t_{n+1} < t_n < t_{n-1} < \cdots < t_1 < t_0 \ .$$

So, we have $n+2$ many points. Next, we consider all the $n+1$ many intervals $[t_i, t_{i-1})$, $i = n+1, \ldots, 1$. Moreover, consider the n many points $\Phi_j(n)$ for $j = 0, \ldots, n-1$. Since we have more intervals than points, we can effectively find at least one interval, say $[t_{i_0}, t_{i_0-1})$ that does not contain any of these points, i.e., for no $j < n$ do we have

$$t_{i_0} \leqslant \Phi_j(n) \leqslant h(n, t_{i_0}) < t_{i_0-1} \ .$$

Therefore, we are setting $t(n) = t_{i_0}$. Then, by construction $t(n) = t_{i_0} \geqslant t_{n+1} = a(n)$.

Moreover, for $n > j$ and $\Phi_j(n) \geqslant t(n) = t_{i_0}$ we obtain from $\Phi_j(n) \notin [t_{i_0}, t_{i_0-1})$ directly that

$$\Phi_j(n) \geqslant t_{i_0-1} = h(n, t_{i_0}) + 1 > h(n, t(n)) \ .$$

Finally, since $h(n, y) \geqslant y$ the condition $\Phi_j(n) \leqslant t(n)$ also implies $F_j(n) \leqslant h(n, t(n))$. Consequently, $\mathcal{R}_t^{[\varphi,\Phi]} = \mathcal{R}_{h\circ t}^{[\varphi,\Phi]}$. ∎

So, we have just seen that there are indeed gaps in the complexity hierarchy which do not contain any new function from $\mathcal{R}$. These gaps are described by the functions $\mathfrak{t}$ and $\mathfrak{h} \circ \mathfrak{t}$ from the Gap-Theorem.

References

[1] M. BLUM. A Machine Independent Theory of Complexity of Recursive Functions, *Journal of the ACM*, 14(2):322–336, 1967.

[2] A. BORODIN. Computational complexity and the existence of complexity gaps, *Journal of the ACM*, 19(1):158–174, 1972

[3] E. M. MCCREIGHT AND A. R. MEYER. Classes of Computable Functions Defined by Bounds on Computation: Preliminary Report, in *Proceedings of the first annual ACM symposium on Theory of computing, Marina del Rey, California, United States*, pp. 79–88, 1969, ACM Press.

[4] L. H. LANDWEBER AND E. L. ROBERTSON. Recursive properties of abstract complexity classes, *Journal of the ACM* 19(2):296–308, 1972.

[5] C.H. SMITH. *A Recursive Introduction to the Theory of Computation*, Springer-Verlag, New York, Berlin, Heidelberg, 1994.

[6] B. TRAKHTENBROT. Turing computations with logarithmic delay, *Algebra i Logika*, 3, 3–48, 1964

[7] P. YOUNG. Easy Constructions in Complexity Theory: Gap and Speed-up Theorems, *Proceedings of the American Mathematical Society*, 37(2):555–563, 1973.

Index

List of Symbols