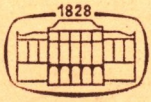
RECURSIVE FUNCTIONS IN COMPUTER THEORY

RÓZSA PÉTER



AKADÉMIAI KIADÓ · BUDAPEST

RECURSIVE FUNCTIONS IN COMPUTER THEORY

by

RÓZSA PÉTER

The present book, based mainly on the author's own research, gives an insight into some possible applications of recursive functions and their generalization in computer theory. Since both the input data and the sequential output of the results can be encoded into natural numbers, it follows that the functioning of a computer can always be considered as the computation of a value of a numeric function. In studying how the computation of partial recursive number-theoretic functions can be programmed, essentially all questions concerning the problems soluble by computer are studied. These problems always arise whenever a general mathematical theory is applied to practical problems.

Rózsa Péter, Corresponding Member of the Hungarian Academy of Sciences, retired professor of the Eötvös Loránd University of Budapest, died in February 1977. By now she has become a classic exponent of mathematical logic as one of the founders of recursion theory. Besides being a great scientist, she played a prominent role in the mathematical profession in Hungary.

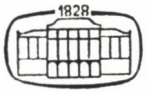
AKADÉMIAI KIADÓ, BUDAPEST

RECURSIVE FUNCTIONS IN COMPUTER SCIENCE

RECURSIVE FUNCTIONS IN COMPUTER SCIENCE

RÓZSA PÉTER

formerly
Professor of Mathematics
Eötvös Loránd University of Budapest



AKADÉMIAI KIADÓ, BUDAPEST 1981

This book is the English translation of the German edition
Rekursive Funktionen in der Komputer-Theorie
published by Akadémiai Kiadó, Budapest

Translated by
I. JUHÁSZ

Joint edition published by
Akadémiai Kiadó
Publishing House of the Hungarian Academy of Sciences
H-1054 Budapest, Alkotmány u. 21. Hungary
and
Ellis Horwood Limited
Market Cross House, Cooper Street, Chichester, West Sussex, England

ISBN 963 05 2257 8

Printed in Hungary

Table of Contents

Preface

In different aspects of computer programming one meets definitions that seem to be circular, in that the notion to be defined plays a role in the definition. A closer look, however, shows that in such cases we are always concerned with recursive definitions, and the aim of this book is to develop exact definitions of this background.

The action of a computer can always be thought of as a process such that in response to given input data, the machine produces certain outputs. Since both the input data and the sequential output of the results can be encoded into natural numbers, it follows that the functioning of a computer can always be considered as the computation of a value of a numeric function. With the idealization that the contents of the computer store are unlimited, it can be shown that the functions computable by a computer are identical with the class of functions known as the "**partial recursive** functions".

Therefore if we study how the computation of partial recursive number-theoretic functions can be programmed, essentially all questions concerning the problems solvable by a computer will be studied. The above idealization (which will be assumed throughout in what follows) always arises if a general mathematical theory is applied to practical problems. This is often expressed by saying "the infinite is a useful approximation to the large but finite".

The computer does not understand and manipulate the data (including the numbers) in substance, but only as sequences of symbols. Hence we shall also have to deal with the generalization of the theory of recursive functions to the case of sets of **numeric structure.**

The practical side of the subject does not fit into the framework of this book. I am in the convenient situation that I do not even have to cite any

literature dealing with this: it suffices to refer to the references quoted in Barron's book[1]. However, because my own publications are not quoted there and it is from these that almost the whole material of the present book is derived, I shall refer to these and to several papers by other authors throughout the book.

Almost no previous knowledge is necessary. The lengthy general proofs of the quoted works will not be given. The arguments will be mostly illustrated by examples.

Finally, may I express my gratitude to L. Kalmár, who persuaded me to work in this field and to write this book; to B. Dömölki, who carefully read the manuscript and helped me with valuable suggestions; and to G. Révész and J. Urbán, who also read the manuscript and made useful remarks.

Rózsa Péter

[1] D. W. Barron: *Recursive techniques in Programming*, Macdonald and Co, London (1968). References at the end of every chapter. See in particular J. McCarthy's works quoted there.

Foreword

Rózsa Péter, Corresponding Member of the Hungarian Academy of Sciences, retired professor of the Eötvös Loránd University of Budapest died in February 1977. Her scientific career started in the late 1920s. By now, she became a classic of mathematical logic as one of the founders of recursion theory. In her early works she has made important contributions to the development of the concept of recursive functions. In 1951 she was the first to publish a monograph on this subject. The present book, based mainly on her own research conducted in about the last twenty years of her life, gives an insight to possible applications of recursive functions and their generalizations in computer theory.

Besides being a great scientist, she has played a prominent role in the mathematical life of our country. Her kind forceful personality and her striving for justice made her a natural champion of all good causes we aimed at in organizing mathematics in Hungary.

Interviewed by the Hungarian Television in 1970, asked by the reporter whether her subject in mathematics had practical applications, she answered: "I must admit that I never thought of this while doing research work. The problems I dealt with arose as a consequence of inevitable inner developments in mathematics. This made them exciting for me and I would not even have dreamed that my results might have practical applications. It should be a warning example to all those who want to discourage research in pure mathematics that they are preventing the cause of the applications of mathematics as well."

This book was first published in German, in 1976. We should express our thanks to dr. István Juhász, who with indefatigable zeal worked to make it available in English.

András Hajnal

Chapter 1

Recursions in Binary Computer Arithmetic

1.1 Binary Representation

The fact that we can build a computer out of parts, each one of which is capable of having two states, is due to the recursive dependence of the basic operations between natural numbers on their binary digits [2].
We have to take care that the digits of the same place-value of the operands a and b should stand in the same order, and the appearing empty spaces be occupied by 0. The final (that is first from the right) of the binary digits of a and b will be denoted by a_0 and b_0, the next to the last ones by a_1 and b_1, and so on.

1.2 Digital Addition

When we add a and b let the corresponding digits of their sum be denoted by s_0, s_1, s_2, and so on, and those of the carry from the right-hand neighbouring place by 0, u_1, u_2, etc. Using the notation & for "and", and $\vee$ for "or" (the Latin "vel", permits the occurrence of both alternatives), and taking that the carry 1 occurs if at least two of the values a_n, b_n, u_n is equal to 1, we obtain the following relationship:

$$\begin{cases} u_0 = 0 \\ u_{n+1} = \begin{cases} 1, & \text{if} \quad a_n = b_n = 1 \vee a_n = u_n = 1 \vee b_n = u_n = 1 \\ 0 & \text{otherwise,} \end{cases} \\ s_n = \begin{cases} 1, & \text{if} \quad (u_n = 0 \,\&\, a_n \neq b_n) \vee (u_n = 1 \,\&\, a_n = b_n) \\ 0 & \text{otherwise.} \end{cases} \end{cases} \tag{1.2.1}$$

[2] The binary representation of a natural number is

$$2^k \cdot a_k + 2^{k-1} \cdot a_{k-1} + \ldots + 2 \cdot a_1 + a_0,$$

written in brief as

$$a_k a_{k-1} \ldots a_1 a_0,$$

where each one of the digits $a_0, a_1, \ldots, a_k$ is either 0 or 1. It can be required that $a_k = 1$ or an arbitrary number of zeroes may be added to the left.

For the computer it is still simpler, if two states are distinguished according to whether the carry is 0 or 1. In order to be able to indicate the change of states, we put, in general,

$$\bar{c} = \begin{cases} 1, & \text{if} \quad c = 0 \\ 0 & \text{otherwise,} \end{cases}$$

and the digits of the sum at different states will be distinguished by upper primes. Since for $u_n=0$ a change of state occurs if $a_n=b_n=1$, and for $u_n=1$ if $a_n=b_n=0$, the definition of the sum $a+b$ reads more precisely as follows:

$$\begin{cases} u_0 = 0 \\ u_{n+1} = \begin{cases} \bar{u}_n, & \text{if} \quad a_n = b_n \neq u_n \\ u_n & \text{otherwise,} \end{cases} \end{cases} \tag{1.2.2}$$

$$s'_n = \begin{cases} 1, & \text{if} \quad a_n \neq b_n \\ 0, & \text{if} \quad a_n = b_n, \end{cases}$$

$$s''_n = \begin{cases} 1, & \text{if} \quad a_n = b_n \\ 0, & \text{if} \quad a_n \neq b_n, \end{cases}$$

$$s_n = \begin{cases} s'_n, & \text{if} \quad u_n = 0 \\ s''_n, & \text{if} \quad u_n = 1. \end{cases}$$

1.3 Digital Subtraction

It is easy to see that for $a \geqq b$ the digits of the difference $a-b$ can be obtained in exactly the same way, if the definition of the carry, which we will denote by u_n^-, is modified like this:

$$\begin{cases} u_0^- = 0 \\ u_{n+1}^- = \begin{cases} \overline{u_n^-}, & \text{if} \quad a_n = u_n \neq b_n \\ u_n^- & \text{otherwise.} \end{cases} \end{cases} \tag{1.3.1}$$

1.4 Digital Multiplication

In multiplying a given multiplicand a with a given multiplier b we have to use the fact that the addition of two summands can already be done digitwise, hence so can be done the special case of the multiplication in which the multiplier is 2, since

$$2 \cdot c = c + c.$$

This can more easily be done directly, since the digits Z_n of the double of c are obtained by affixing 0 to the right-hand end of the binary form of c:

$$\begin{cases} z_0 = 0 \\ z_{n+1} = c_n. \end{cases}$$

In general the multiplication is carried out in such a way that a is multiplied, step by step, by

$$b_0, 2 \cdot b_1, 2^2 \cdot b_2, \ldots,$$

and then these products are summed up. This can also be accomplished so that first (that is at step zero) a is taken by itself, then it is doubled, then the result is again doubled, etc. and the value obtained in the nth step is taken as a summand if and only if $b_n = 1$. The addition of the subproducts can also be carried out step by step. Precisely,

$$\begin{cases} d(0) = a \\ d(n+1) = 2 \cdot d(n) \end{cases} \tag{1.4.1}$$

and

$$\begin{cases} s(0) = 0 \\ s(n+1) = \begin{cases} s(n), & \text{if } b_n = 0 \\ s(n) + d(n), & \text{if } b_n = 1. \end{cases} \end{cases} \tag{1.4.2}$$

If b_k is the last digit of b (from the right) that is not equal to 0, then

$$a \cdot b = s(k+1),$$

which can be computed digit by digit as above.

In the last two definitions, the functional notation $d(n)$ and $s(n)$ was used. The indexed letters could also be written in this form; e.g. the value of the digit a_n depends on n, hence it could be written as $a(n)$. There is no need to give other well-known examples of definitions in the basic binary arithmetic of the computer, for the ones given so far already show the basic problems of such definitions.

1.5 Circular Definitions

The definitions denoted by (1.2.1), (1.2.2), (1.3.1), (1.4.1) and (1.4.2) seem to be circular because in order to compute a value of the functions defined in them other values of the functions to be defined are needed. In full generality such definitions are indeed useless. If e.g. we omit the first line in definition (1.4.1) or we replace the second by

$$d(n+1) = d(2 \cdot n),$$

then in the first case none of the values $d(n)$ and in the second case none of those with $n>1$, could be computed.

Equation (1.4.1), however, is a particular example of primitive recursion, by means of which a numeric function (that is one defined for and taking its values from the natural numbers) is uniquely determined. Here this function is

$$d^*(n) = 2^n \cdot a,$$

since

$$d^*(0) = 2^0 \cdot a = a,$$

and

$$\mathbf{d^*(n+1) = 2^{n+1} \cdot a = 2 \cdot (2^n \cdot a) = 2 \cdot d^*(n)}$$

are satisfied.

Moreover this is the only function satisfying (1.4.1). Indeed, if the numeric function $d^{**}(n)$ satisfies it as well, then

$$d^{**}(0) = a = d^*(0),$$

and if for some n we have

$$d^{**}(n) = d^*(n),$$

then this equality is also valid for $n+1$, since then

$$d^{**}(n+1) = 2 \cdot d^{**}(n) = 2 \cdot d^*(n) = d^*(n+1).$$

Thus $d^{**}(n)$ is identical with $d^*(n)=2^n \cdot a$.

Thus (1.4.1) yields the special case of multiplication in which the multiplier is an arbitrary power of 2. The computer, however, knows nothing about $2^n \cdot a$. It can only recognize that in a storage location going from the right to the left, first 0 occurs n times and then the digits of a occur in order. If, purely formally, a is considered as the sequence of its binary digits, then (1.4.1) determines a new kind of primitive recursion, whereby a function which is not numeric but whose arguments and values are finite sequences of symbols. That is why in the following chapters we shall consider such generalizations of the notion of recursivity.

Chapter 2

General Recursive Functions

2.1 Primitive Recursion

First I restrict myself to the case of numeric functions.

Since the natural numbers can be obtained from 0 by means of the operation "counting 1 along", it is usual to prove a numeric statement by showing that it is satisfied for 0 and that its validity is "inherited" from any natural number to its successor (mathematical induction). Moreover we can define a numeric function by prescribing its value at 0 and providing a method for obtaining its value at $n+1$ from n and the value at n, for any given number n. Such a definition, by means of which the value of the function to be defined is computable in a finite number of steps for any given argument, is called a primitive recursion. It has the form

$$\begin{cases} \varphi(0) = a \\ \varphi(n+1) = \beta(n, \varphi(n)), \end{cases} \tag{2.1.1}$$

where a is a given number and $\beta(n, w)$ is an already known function of n and w. In addition to the "recursion variable" n other variables, known as parameters, can also occur in β.

There is always exactly one function $\varphi^*(n)$, which satisfies the defining system of equations (2.1.1). L. Kalmár [3] has shown this by using a sequence of partially defined functions (that is functions defined for a subset of the natural numbers), which he called "partial solutions" of (2.1.1). By this we mean that

(1) if such a function ψ is defined at 0, then

$$\psi(0) = a.$$

[3] L. Kalmár: *On the Possibility of Definition by Recursion,* Acta Sci. Math. **9** (1940) pp. 227–232.

(2) if $\psi(n+1)$ is defined for some n, then so is $\psi(n)$ and [4],

$$\psi(n+1) = \beta(n, \psi(n)).$$

If ψ_1 and ψ_2 are partial solutions of (2.1.1) and both are defined for n, then

$$\psi_1(n) = \psi_2(n),$$

since this is true for $n=0$ by (1) and is induced from n to $n+1$ by (2).
Now Kalmár defined a sequence of functions

$$\varphi_0, \varphi_1, \varphi_2, \ldots \tag{2.1.2}$$

for finite subsets of the natural numbers as follows: Let φ_0 be defined for 0 only by

$$\varphi_0(0) = a.$$

This is obviously a partial solution of (2.1.1). If a partial solution φ_n of (2.1.1) is now given, for which $\varphi_n(0)$ and $\varphi_n(n)$ are defined but $\varphi_n(n+1)$ is not (as is the case for $n=0$), then by [4] φ_n is only defined for arguments less than $n+1$. We define φ_{n+1} for an argument less than $n+1$ if and only if φ_n is defined there, with the same value, but we also define it for the argument $n+1$ as follows:

$$\varphi_{n+1}(n+1) = \beta(n, \varphi_n(n)).$$

Thus, this defined partial function φ_{n+1} inherits the above properties of φ_n. Firstly, if $\varphi_n(0)$ was defined then so is $\varphi_{n+1}(0)$, moreover

$$\varphi_{n+1}(0) = \varphi_n(0) = a.$$

Hence φ_{n+1} satisfies (1).
$\varphi_{n+1}(n+1)$ was defined, however $\varphi_{n+1}((n+1)+1)$ was not, since otherwise $\varphi_n((n+1)+1)$, and therefore by (2) $\varphi_n(n+1)$, would be defined.
Finally φ_{n+1} also satisfies (2), and therefore is a partial solution of (2.1.1). Indeed, assume that $\varphi_{n+1}(m+1)$ is defined for an m $(\neq(n+1)+1)$. If $m \neq n$, then $\varphi_n(m+1)$ and thus by (2) $\varphi_n(m)$, are also defined in such a way that

$$\varphi_{n+1}(m+1) = \varphi_n(m+1) = \beta(m, \varphi_n(m)) = \beta(m, \varphi_{n+1}(m)).$$

If $m=n$, then

$$\varphi_{n+1}(n+1) = \beta(n, \varphi_n(n)) = \beta(n, \varphi_{n+1}(n))$$

by the definition of φ_{n+1}. Hence φ_{n+1} satisfies condition (2).

[4] Note [2] implies that whenever ψ is defined for $n+1$ it is also defined for all the smaller numbers. However, in his proof, Kalmár avoided the use of the relation $m<n$, since this is defined in terms of addition like this: "there exists a number r different from 0 such that

$$n = m+r,"$$

while addition is defined through a primitive recursion.

The members of the above defined sequence (2.1.2) are partial solutions of (2.1.1), moreover for each n the function φ_n is defined for n.
It follows that if $\varphi^*(n)$ denotes the common value at n of the partial solutions of (D) defined for n, then $\varphi^*(n)$ *is defined for each n and is the unique (complete) solution of* (2.1.1).
The existence of a unique solution for the types of recursion to be mentioned below could be proved similarly, but we shall not discuss it here.

2.2 Dummy Variables

The functions defined in Ch. 1 were all primitive recursive, in a sense to be defined. To demonstrate this, we shall examine those definitions more closely.

2.2.1 The Functions 0 and $n+1$

The simplest of the definitions was used to define the "change of states" in the addition. Replacing the variable c used there by the more usual n, and taking into account that every number different from 0 can be written in the form $n+1$, this definition reads as follows:

$$\begin{cases} \bar{0} = 1 \\ \overline{n+1} = 0. \end{cases}$$

This is a primitive recursive definition of the function $\bar{n}$, where 1 stands for the constant a appearing in (2.1.1) and the function $\beta(n, w)$ is represented by the constant 0, which can also be considered as a function of n and w. *We shall always allow the use of dummy variables, on which a function does not really depend.* The constant 0 (including dummy variables) will be taken as an *initial function.*
The constant 1 also plays a role in the above definition but we do not have to take this as an initial function. Indeed, it is convenient to take the "successor function" $n+1$, which is more elementary than the sum [5] and therefore is often denoted by n', as an initial function. Clearly 1 is obtained by substituting 0 into it.

[5] It was observed that school children, who can immediately name the successor of an arbitary natural number, have difficulty when they have to write down in an equation the successor of x. It is not obvious for them that this is obtained as the sum $x+1$.

2.2.2 Primitive Recursive Functions

All the larger numbers can be obtained by substitution from the initial functions. We obtain 3, for example if first in $n+1$ we replace n by $n+1$, then in the resulting $(n+1)+1$ we do the same, finally in $((n+1)+1)+1$ we replace n by 0. Thus all the natural numbers are primitive recursive, for the general definition of this concept goes as follows:

A numeric function is called primitive recursive if it can be obtained from 0 *and* $n+1$ *by means of finitely many substitutions and primitive recursions.*

2.3 Recursive Operations

As a simple example of a primitive recursive function we can consider the identity function $\varphi(n)=n$, since it can be defined directly from the initial functions by the primitive recursion

$$\begin{cases} \varphi(0) = 0 \\ \varphi(n+1) = n+1. \end{cases}$$

In the primitive recursive definition of the sum

$$\varphi(n, a) = a+n,$$

in addition to the recursion variable n a parameter a also occurs. Thus the value of φ for $n=0$ is not a constant, but is already a given function of the parameter. Here this is the identity function $\varphi(n)=n$, corresponding to $\alpha(a)=a$.

$$\begin{cases} \varphi(0, a) = a \\ \varphi(n+1, a) = \varphi(n, a)+1. \end{cases}$$

The function corresponding to $\beta(n, a, w)$ of the general definition, which of course depends on the parameter a, is represented here by the initial function

$$\beta(n, a, w) = w+1,$$

with n and a as dummy variables.
Using the sum, the product

$$\varphi(n, a) = a \cdot n$$

is also obtained as a primitive recursive function, since $(n+1)$ times a can be obtained from n times a by adding a to it: –

$$\begin{cases} \varphi(0, a) = 0 \\ \varphi(n+1, a) = \varphi(n, a)+a. \end{cases}$$

Within the scope of the natural numbers we can only define the "arithmetical difference" (denoted by $a \dot{-} n$), which is the non-negative part of $a-n$; that is it is 0 if a is less than n. First we consider $\varphi(n)=n \dot{-} 1$, this is obtained by the primitive recursion

$$\begin{cases} \varphi(0) = 0 \\ \varphi(n+1) = n. \end{cases}$$

Moreover for $\varphi(n, a)=a \dot{-} n$ we have

$$\begin{cases} \varphi(0, a) = a \\ \varphi(n+1, a) = \varphi(n, a) \dot{-} 1. \end{cases}$$

One of the values $a \dot{-} b$ and $b \dot{-} a$ is always 0, and the other is the absolute value of $a-b$. Thus

$$|a-b| = (a \dot{-} b)+(b \dot{-} a)$$

is obtained by substitution from the sum and the arithmetical difference. Hence it also is primitive recursive.

2.3.1 Primitive Recursive Relations

In Ch. 1 we have seen definitions by cases according to whether two numbers were equal or not. Equality is said to be a primitive recursive relation, since $a=b$ if and only if the primitive recursive function $|a-b|$ vanishes. *In general a numeric relation* $B(a_1, \ldots, a_r)$ *(for* $r=1$ *a property) is called primitive recursive if it has a primitive recursive "characteristic" function* $\beta(a_1, \ldots, a_r)$*, which vanishes exactly for those arguments that satisfy B.*
Thus $a<b$ is also a primitive recursive relation, as it is satisfied exactly for those a and b for which

$$(a+1) \dot{-} b = 0.$$

2.4 Sign Functions

From the primitive recursivity of given relations we can deduce the primitive recursivity of certain others, which are built up from them.
For example, the negation of a relation B is primitive recursive if B is. Indeed, let β be a primitive recursive characteristic function of B. Of course, what is relevant about β is the places where it vanishes. In other words we are only interested in the "sign" of β, where we have in mind the function

$$\operatorname{sign}(a) = \begin{cases} 1, & \text{if} \quad a > 0 \\ 0, & \text{if} \quad a = 0 \\ -1, & \text{if} \quad a < 0, \end{cases}$$

defined for all integers. Since here negative numbers are not considered, a suitable sign function can be defined by means of the following primitive recursion: –

$$\begin{cases} \mathrm{sg}(0) = 0 \\ \mathrm{sg}(n+1) = 1. \end{cases}$$

For the negation $\bar{B}$ of B the exact opposite of this (denoted by $\overline{\mathrm{sg}}$) is used: –

$$\begin{cases} \overline{\mathrm{sg}}(0) = 1 \\ \overline{\mathrm{sg}}(n+1) = 0. \end{cases}$$

This was actually introduced earlier in Ch. 1 for another purpose, with the notation $\bar{c}$. Clearly $\overline{\mathrm{sg}}(\beta)$ is a primitive recursive characteristic function of B, since it vanishes if and only if B is not satisfied.

2.4.1 Closure of Recursive Relations

In definition (1.2.1) (and, implicitely, in (1.2.2) and (1.3.1) as well) combinations of relations by "and" (conjunction) and "or" (disjunction) occur. Together with B_1 and B_2 the relations

$$B_1 \,\&\, B_2 \quad \text{and} \quad B_1 \vee B_2$$

are also primitive recursive. Indeed, if β_1 and β_2 are primitive recursive characteristic functions of B_1 and B_2, respectively, then $\beta_1+\beta_2$ and $\beta_1 \cdot \beta_2$ are primitive recursive characteristic functions of $B_1 \& B_2$ and $B_1 \vee B_2$, respectively.

The implication $B_1 \to B_2$ (which means "if B_1 is valid, so is B_2") can also be written as

$$\bar{B}_1 \vee B_2;$$

it is also primitive recursive if B_1 and B_2 are.

For the negation $\overline{a=b}$ I will use the more usual notation $a \neq b$.

2.5 Definition by Cases

As was noted earlier, in Ch. 1 we used definitions by cases. In general we have: – If $\alpha_1, \alpha_2, \ldots, \alpha_k$ are primitive recursive functions and $B_1, B_2, \ldots, B_{k-1}$ are pairwise exclusive primitive recursive relations, then the function φ, defined as follows, is also primitive recursive:

$$\varphi = \begin{cases} \alpha_1, & \text{if } B_1 \text{ is true} \\ \ldots\ldots\ldots\ldots\ldots\ldots & \\ \alpha_{k-1}, & \text{if } B_{k-1} \text{ is true} \\ \alpha_k & \text{otherwise.} \end{cases}$$

First of all, the "otherwise" can be replaced here by

$$B_k = \bar{B}_1 \,\&\, \bar{B}_2 \,\&\, \ldots \,\&\, \bar{B}_{k-1}.$$

Secondly, assume that $\beta_1, \ldots, \beta_k$ are primitive recursive characteristic functions of $B_1, \ldots, B_k$, respectively. As these relations are pairwise exclusive, for each argument exactly one of the values $\overline{\mathrm{sg}}(\beta_1), \ldots, \overline{\mathrm{sg}}(\beta_k)$ is equal to 1 (namely $\overline{\mathrm{sg}}(\beta_i)=1$, if $\beta_i=0$, that is, if B_i is satisfied). Thus φ can be defined by

$$\varphi = \alpha_1 \cdot \overline{\mathrm{sg}}(\beta_1) + \alpha_2 \cdot \overline{\mathrm{sg}}(\beta_2) + \ldots + \alpha_k \cdot \overline{\mathrm{sg}}(\beta_k).$$

The built up function φ remains primitive recursive if in its definition by cases the right-hand side contains the value of φ taken at the immediately preceding value of the recursion variable. An example for this is given below as a modification of definition (1.2.2). The same is true for definitions by cases of other types of recursion that we shall treat later on.

A particular example of a built up primitive recursive function is $a(n)$, the nth digit from the right of a number a, given in binary form. If for example

$$a = 10111,$$

then

$$a(n) = \begin{cases} 1, & \text{if} \quad n = 0 \\ 1, & \text{if} \quad n = 1 \\ 1, & \text{if} \quad n = 2 \\ 0, & \text{if} \quad n = 3 \\ 1, & \text{if} \quad n = 4 \\ 0 & \text{otherwise.} \end{cases}$$

In this sense all the functions defined in Ch. 1 are primitive recursive. Consider, for example, (1.2.2) in which a and b are fixed numbers and instead of u_n, a_n, b_n the notation $u(n)$, $a(n)$, $b(n)$ is used. Then

$$\begin{cases} u(0) & = 0 \\ u(n+1) & = \begin{cases} \bar{u}(n), & \text{if} \quad a(n) = b(n) \neq u(n) \\ u(n) & \text{otherwise.} \end{cases} \end{cases}$$

As we have said already, here

$$\bar{u}(n) = \overline{\mathrm{sg}}\,(u(n)).$$

Since the function

$$\beta(n, w) = \begin{cases} \overline{\mathrm{sg}}(w), & \text{if} \quad a(n) = b(n) \,\&\, b(n) \neq w \\ w & \text{otherwise,} \end{cases}$$

is primitive recursive, $u(n)$ is thus defined by the primitive recursion: –

$$\begin{cases} u(0) = 0 \\ u(n+1) = \beta(n, u(n)). \end{cases}$$

2.6 Further Recursive Functions

After having given the above examples, in what follows I shall list other primitive recursive numeric functions, without however giving any actual proofs that they really are primitive recursive. For these I shall refer the reader to my book [6]. The functions used in elementary number theory are all primitive recursive, for example the nth prime number p_n as a function of n, the exponentiation a^n, the exponent $\exp_n(a)$ of the nth prime number p_n in the unique prime factor representation of a (we put $\exp_n(0)=0$), the "arithmetical quotient" $\left[\frac{a}{n}\right]$ (which is 0 if $n=0$ and the largest number contained in $\frac{a}{n}$ otherwise), or the remainder $\operatorname{res}(a, n)$, obtained when dividing a by n (this is understood to be a if $n=0$). This shows that the nth binary digit of the number a is a primitive recursive function of n and a, namely it is

$$\operatorname{res}\left(\left[\frac{a}{2^n}\right], 2\right).$$

Also $\left[\frac{a}{n}\right]$ for $n \neq 0$ was defined as the smallest number i up to (and including) a, for which $(i+1) \cdot n$ is already bigger than a. In general, for any primitive recursive relation B, the expression

$$\mu_i[i \leqq n \ \& \ B(i, a_1, \ldots, a_r)],$$

(which means the smallest number i up to (and including) n satisfying $B(i, a_1, \ldots, a_r)$, if there is such a number, and 0 otherwise), is a primitive recursive function of $a_1, \ldots, a_r$. Here the implicit bounded existential qualification, denoted by

$$(Ei)[i \leqq n \ \& \ B(i, a_1, \ldots, a_r)]$$

yields a primitive recursive relation of $n, a_1, \ldots, a_r$ for a primitive recursive B, similar to the expression

$$(i)[i \leqq n \rightarrow B(i, a_1, \ldots, a_r)],$$

meaning that "for each i up to n $B(i, a_1, \ldots, a_r)$ holds true".

The largest i for which the ith binary digit of $a \neq 0$ is 1 (where the first digit from the right is considered "0th") is then

$$\mu_i[i \leqq a \ \& \ 2^{i+1} > a],$$

[6] R. Péter: *Recursive Functions*, Budapest, New York, London (1967); earlier published in German, Russian and Chinese.

which is a primitive recursive function of a. It would be easy to provide here a smaller upper bound for i than a. However it is not worth-while to calculate the exact upper bound, this being irrelevant with respect to the smallest i of the given property. Denoting this by $k(a)$ and the nth binary digit of a by $z(n, a)$ we have

$$a = \sum_{n=0}^{k(a)} z(n, a) \cdot 2^n.$$

Here the values of a primitive recursive function of n are added up, from $n=0$ to a non-constant bound. In general, if $\varphi(n, a_1, \ldots, a_r)$ is primitive recursive, then

$$\sum_{n=a}^{b} \varphi(n, a_1, \ldots, a_r)$$

and

$$\prod_{n=a}^{b} \varphi(n, a_1, \ldots, a_r)$$

are primitive recursive functions of $a, b, a_1, \ldots, a_r$.

2.6.1 Sequential Calculation

According to the above, the binary digits of the value of any primitive recursive function φ can be obtained as primitive recursive functions of the binary digits of its arguments: first the arguments as primitive recursive functions of their digits, then φ of these arguments, finally the digits of the obtained φ-value.

These detours can actually be avoided. The digits of the values of primitive recursive functions can be computed successively from the right to the left, from the digits of the arguments.

For the initial functions 0 and $n+1$ this can be seen immediately. Every digit of 0 is 0 (independently of the variables), while $n+1$ is the particular case of the sum $a+b$, with $b=1$, having the digits

$$b_0 = 1,\ b_1 = 0,\ b_2 = 0, \ldots,$$

for which this has already been established. Of course, it could also be shown directly. Next, if the digits of the values of the functions $\alpha, \beta_1, \ldots$ $\ldots, \beta_s$ are obtainable as primitive recursive functions of the digits of their

arguments, then this holds also for the function

$$\alpha(\beta_1(a_1, \ldots, a_r), \ldots, \beta_s(a_1, \ldots, a_r)),$$

obtained from them by substitution. Finally, this property is carried over from the functions $\alpha(a_1, \ldots, a_r)$ and $\beta(n, a_1, \ldots, a_r, w)$ to the function φ determined by the primitive recursion

$$\begin{cases} \varphi(0, a_1, \ldots, a_r) = \alpha(a_1, \ldots, a_r) \\ \varphi(n+1, a_1, \ldots, a_r) = \beta(n, a_1, \ldots, a_r, \varphi(n, a_1, \ldots, a_r)). \end{cases}$$

Indeed, for $n=0$ this can be seen immediately, and it is transferred from n to $n+1$. If it is true for the value $\varphi(n, a_1, \ldots, a_r)$, then it is also true for $\varphi(n+1, a_1, \ldots, a_r)$ obtained from this value and β by substitution.
Thus the digits of the arithmetical quotient and of the remainder in division can be obtained consecutively from the digits of the dividend and divisor. Following through the construction of a primitive recursive function out of the initial ones, however, might bring with it unnecessary detours. In this construction of the sum $a+n$, for instance, the multiple application of the successor function is needed, whereas we also have a direct way of computing the digits of the sum from those of the summands. In practice one always strives for constructions with the fewest possible detours.

2.6.2. Restriction to Iterations

In the rest of this chapter, I will list several facts about number theoretic recursive functions, mainly without proofs. For these proofs I again refer to my book [6].
First I want to mention that in the construction of the primitive recursive functions, we can restrict ourselves to the following simplest particular case of primitive recursion:

$$\varphi(0) = 0$$

$$\varphi(n+1) = \beta(\varphi(n)),$$

provided that we also admit further initial functions; for example the sum $a+n$, the product $a \cdot n$, the arithmetical difference $a \dot{-} n$, and the "arithmetical square root" $[\sqrt{n}]$ (the largest integer not exceeding $\sqrt{n}$). In fact, the number of necessary initial functions can be reduced to three.

In the above special primitive recursion the values of φ are obtained as follows: –

$$\varphi(0) = 0.$$
$$\varphi(1) = \beta(0).$$
$$\varphi(2) = \beta(\beta(0)).$$
$$\varphi(3) = \beta(\beta(\beta(0))).$$
$$\ldots\ldots\ldots\ldots\ldots$$

This, therefore is simply the iteration of the function β at the argument 0, also denoted by $\beta^{(n)}(0)$.

2.6.3 Course-of-values Recursion

There are types of recursive definitions of numeric functions different from primitive recursion. Some of these can be reduced to primitive recursion, but not all of them.

In *course-of-values recursion* the value of the function at a given argument is expressed by means of values of the same function taken at arbitrary previous places (not only the immediately preceding one). The "course of values" of a function φ up to n can be encoded by the number

$$p_0^{\varphi(0)} \cdot p_1^{\varphi(1)} \cdot \ldots \cdot p_n^{\varphi(n)},$$

where p_n is the nth prime number in increasing order (2 being the "0th"). By the uniqueness of the prime factor representation of integers the value $\varphi(i)$ for $i \leqq n$ can be obtained from this number as

$$\exp_i\left(\prod_{j=0}^{n} p_j^{\varphi(j)}\right),$$

the exponent of the ith prime number. Thus, in general, the course-of-values recursion has the form

$$\begin{cases} \varphi(0) = a \\ \varphi(n+1) = \beta\left(n, \prod_{j=0}^{n} p_j^{\varphi(j)}\right), \end{cases}$$

where a (constant or depending on the parameters) and β are given primitive recursive functions.
Here the course-of-values function

$$\psi(n) = \prod_{j=0}^{n} p_j^{\varphi(j)}$$

belonging to φ can be defined by the following primitive recursion: –

$$\begin{cases} \psi(0) = p_0^{\varphi(0)} = 2^a \\ \psi(n+1) = \psi(n) \cdot p_{n+1}^{\varphi(n+1)} = \psi(n) \cdot p_{n+1}^{\beta(n, \psi(n))}, \end{cases}$$

and $\varphi(n)$ is obtained from $\psi(n)$ by the substitution

$$\varphi(n) = \exp_n(\psi(n)).$$

This shows that the course-of-value recursion remains within the class of the primitive recursive functions.

2.7 Simultaneous Recursion

The definition of two (and similarly more) functions by *simultaneous recursion* has the form

$$\begin{cases} \varphi_1(0) = a_1 \\ \varphi_1(n+1) = \beta_1(n, \varphi_1(n), \varphi_2(n)), \end{cases} \quad \begin{cases} \varphi_2(0) = a_2 \\ \varphi_2(n+1) = \beta_2(n, \varphi_1(n), \varphi_2(n)). \end{cases}$$

This can be reduced to the definition of the function

$$\varphi(n) = 2^{\varphi_1(n)} \cdot 3^{\varphi_2(n)},$$

from which the functions φ_1 and φ_2 arise by the following substitutions: –

$$\varphi_1(n) = \exp_0(\varphi(n)), \quad \varphi_2(n) = \exp_1(\varphi(n)).$$

Using the primitive recursive auxiliary function

$$\beta(n, u) = 2^{\beta_1(n, \exp_0(u), \exp_1(u))} \cdot 3^{\beta_2(n, \exp_0(u), \exp_1(u))},$$

φ is determined by the following primitive recursion: –

$$\begin{cases} \varphi(0) = 2^{a_1} \cdot 3^{a_2} \\ \varphi(n+1) = \beta(n, \varphi(n)). \end{cases}$$

Thus simultaneous recursion does not extend the class of the primitive recursive functions, either.

2.7.1 Nested Recursion

So far the parameters have played an incidental role, which is why sometimes they were not even indicated. There are, however, recursive definitions, in which the parameters do not remain unchanged. They might have to satisfy some conditions, even depending on previous values of the function to be

defined. This might also lead to nestings of the previous function values in the definition, as in the following example:

$$\begin{cases} \varphi(0, a) = \alpha(a) \\ \varphi(n+1, a) = \beta\big(n, a, \varphi(n, \gamma(n, a, \varphi(n, a)))\big). \end{cases}$$

If in such a definition the maximum number of nestings is fixed, then this does not extend the class of primitive recursive functions. However, the number of nestings can also be varied, and even be dependent on earlier values of the function to be defined. By means of such definitions it is possible to define functions that are not primitive recursive.

2.7.2 Multiple Recursion

There are also recursions on several variables simultaneously, as in the example below: –

$$\begin{cases} \varphi(0, n) = \alpha_1(n) \\ \varphi(m+1, 0) = \alpha_2(m) \\ \varphi(m+1, n+1) = \beta\big(m, n, \varphi(m, \gamma(m, n)), \varphi(m+1, n)\big), \end{cases}$$

where $\varphi\big(m, \gamma(m, n)\big)$ and $\varphi(m+1, n)$ can be considered as "previous" values of φ, in m and n respectively.

2.7.3 The Ackermann–Péter Function

If no nestings of the previous values occur, then these *multiple recursions* remain within the class of primitive recursive functions. However, consider the following double recursion with a single nesting

$$\begin{cases} \varphi(0, n) = n+1 \\ \varphi(m+1, 0) = \varphi(m, 1) \\ \varphi(m+1, n+1) = \varphi\big(m, \varphi(m+1, n)\big) \end{cases}$$

which defines φ. It is known as the Ackermann–Péter function, and is not primitive recursive.

2.8 General Recursive Functions

I will not list any further variations of recursive definitions. They all agree in that the whole construction of the function to be defined out of the initial functions is obtained via a defining system of equations of the form $r=s$, where both r and s are terms built out of natural numbers, number

variables, symbols for the function to be defined and some auxiliary functions [7]. The value of the function under definition, at any given argument, is obtained by applying the following simple steps a finite number of times.

A) the substitution of natural numbers for variables in an (original or derived) equation,

B) the replacement in an equation of a subterm, also occurring as the left-hand side of an equation, by the right-hand side of this second equation.

The above is the actual definition of *general recursion.* The functions computable from systems of equations of the above kind by finitely many applications of steps A) and B) are called *general recursive.*

Let us consider the primitive recursive function defined in (1.4.1) denoted by $\varphi_0(n)$ for the sake of homogeneity of notation. Then (1.4.1) reads as follows: –

$$\begin{cases} \varphi_0(0) = a \\ \varphi_0(n+1) = 2 \cdot \varphi_0(n). \end{cases}$$

The auxiliary function

$$\varphi_1(n) = 2 \cdot n (= n+n)$$

is used here. Taking into account the known definition of the sum

$$\varphi_2(n, m) = m+n$$

(cf. section 2.3), the complete definition of $\varphi_0(n)$ is as follows:

$$\varphi_0(0) = a \tag{2.8.1}$$

$$\varphi_0(n') = \varphi_1(\varphi_0(n)) \tag{2.8.2}$$

$$\varphi_1(n) = \varphi_2(n, n) \tag{2.8.3}$$

$$\varphi_2(0, m) = m \tag{2.8.4}$$

$$\varphi_2(n', m) = (\varphi_2(n, m))'. \tag{2.8.5}$$

Clearly, the sides of these equations are terms of the required form.

[7] More precisely, the successor function has to be denoted by n' $(=n+1)$ here. Hence the natural numbers

$$0, 1, 2, \ldots$$

are

$$0, 0', 0'', \ldots .$$

The number 0 and the numerical variables are terms. If a is a term, then so is a'. Thus all the natural numbers are terms. If $a_1, \ldots, a_r$ are terms and φ_i is a symbol for an r-place function, then $\varphi_i(a_1, \ldots, a_r)$ is also a term. All the terms are generated in this way.

Let us carry out, for this simple example the steps of the computation of the value of φ_0 at $n=1(=0')$, when $a=0''(=2)$. To the right of the newly derived equations we indicate whether they are obtained by the step A) or B), and the number of the equation(s) to which this step was applied. To start with, (2.8.1) has to be repeated with $a=0''$:

Process applied to	eq. no.		
		$\varphi_0(0) = 0''$	(2.8.6)
A	2	$\varphi_0(0') = \varphi_1(\varphi_0(0))$	(2.8.7)
B	7–6	$\varphi_0(0') = \varphi_1(0'')$	(2.8.8)
A	3	$\varphi_1(0'') = \varphi_2(0'', 0'')$	(2.8.9)
A	5	$\varphi_2(0'', 0'') = (\varphi_2(0', 0''))'$	(2.8.10)
A	5	$\varphi_2(0', 0'') = (\varphi_2(0, 0''))'$	(2.8.11)
A	4	$\varphi_2(0, 0'') = 0''$	(2.8.12)
B	11–12	$\varphi_2(0', 0'') = 0'''$	(2.8.13)
B	10–13	$\varphi_2(0'', 0'') = 0''''$	(2.8.14)
B	9–14	$\varphi_1(0'') = 0''''$	(2.8.15)
B	8–15	$\varphi_0(0') = 0''''$	(2.8.16)

Thus we obtained $\varphi_0(1)=4$, if $a=2$.

2.9 Partial Recursive Functions

The notion of *general recursive functions* includes more than the types of special recursive functions we have seen above. So far every concrete numeric function, whose values are effectively computable for all arguments, has proved to be a general recursive function.

It is an interesting fact that this extensive generality can already be achieved if, in the definition

$$\mu_i[i \leqq n \;\&\; B(i, a_1, \ldots, a_r)]$$

of section 2.6, the upper bound n for i is omitted. Indeed, Kleene has shown that every general recursive function can be constructed, starting from several primitive recursive functions, by finitely many applications of substitutions and *μ-operations*. For a relation $B(n, a_1, \ldots, a_r)$ such that

for every $a_1, \ldots, a_r$ there is an i satisfying $B(i, a_1, \ldots, a_r)$ the μ-operation

$$\mu_i[B(i, a_1, \ldots, a_r)]$$

means the smallest such i[8].

If we omit the above requirement that for every $a_1, \ldots, a_r$ there be an i with $B(i, a_1, \ldots, a_r)$, that is we allow that $\mu_i[B(i, a_1, \ldots, a_r)]$ be not defined everywhere, then the above procedure leads us to the *partial recursive functions*. These are also those partially defined numeric functions whose values for all arguments, where they are defined, can be computed from a system of equations with finitely many applications of steps A) and B), just as in the case of general recursive functions. Thus the general recursive functions are exactly those partial recursive ones that are everywhere defined. The identity of two partial recursive functions is denoted by

$$\varphi(a_1, \ldots, a_r) \simeq \psi(a_1, \ldots, a_r)$$

and is to be understood as follows: both are defined for the same values of $a_1, \ldots, a_r$, and wherever they are defined they take the same value.

2.9.1 The Kleene Form

Kleene has constructed a primitive recursive function $\psi(n)$ and for each r a primitive recursive function

$$\tau(i, n, a_1, \ldots, a_r)$$

such that

$$\xi(n, a_1, \ldots, a_r) = \psi(\mu_i[\tau(i, n, a_1, \ldots, a_r) = 0])$$

yields a *universal explicit form* of an r-place partial recursive functions, in the sense that to every system of equations defining a partial recursive function $\varphi(a_1, \ldots, a_r)$ one can determine a natural number n (called "Gödel number"), for which

$$\varphi(a_1, \ldots, a_r) \simeq \xi(n, a_1, \ldots, a_r).$$

In a system of equations defining a function, all the auxiliary functions occur as well as the ones connected with the function to be defined. Thus any one of them, if the admitted steps of computation at no place yield two different values, can also be taken as the function to be defined. Then the others are considered auxiliary. That is why what we said is also valid for the simultaneous partial recursive definition of several functions. Each of these can be brought into the Kleene explicit form.

[8] See S. C. Kleene: *General recursive functions of natural numbers*, Math. Annalen **112** (1936) pp. 727–742.

Chapter 3

Recursive Word Functions

3.1 Symbol Sequences

As was indicated at the end of Ch. 1, a computer does not understand our number theory, it can only notice that it received certain sequences of the symbols 0 and 1, and depending on these it can in turn emit such a sequence. The mathematician first has to consider very carefully how the binary digits of the result of an operation arise from the digits of the operands. He reasons that since

$$1 \cdot 2^n + 1 \cdot 2^n = 2 \cdot 2^n = 2^{n+1} = 1 \cdot 2^{n+1} + 0 \cdot 2^n,$$

a "carry" 1 results if we twice add the digit 1. Afterwards he observes what this implies for sequences of digits, and then this can be applied mechanically [9].

Because of the carry, even taking the successor needs some consideration for numbers in binary form. Let us denote the successor of x, given in binary form, by $s(x)$, the last (that is first from the right) binary digit of x by $\mathrm{lb}(x)$, the "initial part" of x remaining after the omission of $\mathrm{lb}(x)$ by $\mathrm{at}(x)$ and the empty sequence by Λ. Then the following rule can be observed for taking the binary form of the successor (where 1 is considered

[9] I can illustrate this with an example from my teaching experience. When solving equations the elimination of a subtrahend from one side was for a time always carried out with the explanation: "The equilibrium of a pair of scales is not disturbed if the same weight is placed into both scales". After a while, in order to speed things up, a student was asked to say without thinking what distinguished the two equations obtained in this way. She could say: "A subtrahend disappeared from the left-hand side of the first equation and it appeared as a summand on the right-hand side of the second". From here on it was done mechanically: "We transfer the subtrahend as a summand to the other side".

as the successor of the empty sequence):

$$s(x) = \begin{cases} 1, & \text{if } x = \wedge \\ \text{at}(x)1, & \text{if } \text{lb}(x) = 0 \\ s(\text{at}(x))0, & \text{if } \text{lb}(x) = 1. \end{cases}$$

In fact, for the binary forms

$$0, 1, 10, 11, 100, \ldots$$

of the natural numbers

$$0, 1, 2, 3, 4, \ldots$$

we obtain in order:

$$\begin{aligned} s(0) &= \text{at}(0)1 = \wedge 1 = 1. \\ s(1) &= s(\text{at}(1))0 = s(\wedge)0 = 10. \\ s(10) &= \text{at}(10)1 = 11. \\ s(11) &= s(\text{at}(11))0 = s(1)0 = 100. \\ s(100) &= \text{at}(100)1 = 101 \end{aligned}$$

and so on.

These are the successors

$$1, 10, 11, 100, 101, \ldots$$

of the natural numbers in binary form.

The definition of $s(x)$ is apparently some kind of recursion, but for finite sequences of symbols instead of natural numbers. The sequence $s(x)$ is determined in terms of $s(\text{at}(x))$, and the initial part $\text{at}(x)$ of the sequence x can be considered as "a place earlier than x". Taking all the time such earlier places we get back to the empty sequence $\wedge$, for which $s(\wedge)$ is defined as the single-termed sequence 1. The role of 0 in numeric recursion is taken over here by $\wedge$.

A conspicuous difference from the numeric case occurs here, in that x is not the only sequence immediately following $\text{at}(x)$. For example, both $x = 1011$ and $y = 1010$ immediately follow

$$\text{at}(x) = 101 (= \text{at}(y)).$$

3.2 Numeric Structures

In a lecture on September 3, 1959 at the International Symposium on the Foundations of Mathematics (Infinitistic Methods) in Warsaw [10], I outlined a far-reaching generalization of the theory of recursive functions for abstract sets, which, in a certain sense, have a numeric structure. Here a set of elements plays the role of 0, and a set of functions the role of the successor function.

3.2.1 Word Sets

As one of the most important particular cases I mentioned the set of "words over an alphabet A" (where A is a non-empty set, the members of which are called letters), that is the set of all finite sequences of elements of A. The role of 0 is played here by Λ, the empty sequence, while the attachments of a single letter to the end of a word play the role of the successor function. Thus, for each $a \in A$, here xa is a successor function.

3.2.2 Primitive Recursions in Word Sets

Temporarily I shall restrict myself to the particular case in which the predecessors of a word

$$x = a_1 a_2 \dots a_r$$

are its initial segments

$$a_1, a_1 a_2, \dots, a_1 a_2 \dots a_{r-1}, a_1 a_2 \dots a_r.$$

Here the last one is of course not a "proper predecessor", while the one before the last, that is $\mathrm{at}(x)$, is an immediate predecessor. $\mathrm{at}(\Lambda)$ is, by definition, Λ itself. The general form of a primitive recursion is, in this particular case (assuming, as we can, that we have a finite alphabet), as follows: –

$$f(x) = \begin{cases} g, & \text{if} \quad x = \Lambda \\ h(x, f(\mathrm{at}(x))) & \text{otherwise,} \end{cases} \tag{3.2.1}$$

[10] In the same month I submitted a long paper about this, which only appeared several years after the date of submission in two parts: R. Péter: *Über die Verallgemeinerung der Theorie der rekursiven Funktionen für abstrakte Mengen,* Acta Math. Acad. Sci. Hung. **12** (1961) pp. 271–314; second part: **13** (1962) pp. 1–24. Further references concerning this can also be found there. My later works contain new results and several corrections. The last one is R. Péter: *Die Pairschen freien Binoiden als Spezialfälle der angeordneten freien holomorphen Mengen,* Acta Maht. Acad. Sci. Hung. **21** (1970) pp. 297–313.

or with parameters: –

$$f(x, x_1, \ldots, x_n) = \begin{cases} g(x_1, \ldots, x_n), & \text{if } x = \wedge \\ h(x, x_1, \ldots, x_n, f(\mathrm{at}(x), x_1, \ldots, x_n)) & \text{otherwise,} \end{cases}$$

where g and h are given word functions. In the case with no parameters g is a constant word.

The primitive recursive word functions are generated from certain initial functions by means of finitely many applications of substitutions and primitive recursions.

3.3 Initial Functions

Clearly $s(x)$, as defined in section 3.1, is a word function in a word set M over an alphabet A containing 0 and 1. If $h(x, y)$ is defined by

$$h(x, y) = \begin{cases} \mathrm{at}(x)1, & \text{if } \mathrm{lb}(x) = 0 \\ y0, & \text{if } \mathrm{lb}(x) = 1 \\ \wedge & \text{otherwise,} \end{cases}$$

(where "otherwise" means $\mathrm{lb}(x) = \wedge$, that is $x = \wedge$, if A contains no letters different from 0 and 1), then $s(x)$ is determined by the following primitive recursion: –

$$s(x) = \begin{cases} 1, & \text{if } x = \wedge \\ h(x, s(\mathrm{at}(x))) & \text{otherwise.} \end{cases}$$

Thus if the built up function $h(x, y)$ is primitive recursive, so is $s(x)$.
Here $\wedge$ and the successor functions xa, for $a \in A$, are always taken as initial and the attaching of $\wedge$ to x, that is the identity function

$$f(x) = x\wedge = x,$$

can also be added to the list of the initial functions. We shall see later that $\mathrm{at}(x)$ is primitive recursive. Therefore so is $\mathrm{at}(x)1$, which is obtained from $x1$ by substituting $\mathrm{at}(x)$ into it. It will also be shown that a function built up from primitive recursive functions and relations, similar to the numeric case, is again primitive recursive. Therefore it remains only to examine the primitive recursivity of the function $\mathrm{lb}(x)$ and the relation of equality.

3.3.1 The Set of Natural Numbers

In the numeric case the primitive recursive function $|x-y|$ was a characteristic function of the equality $x=y$. It is instructive to keep in mind here the role of the natural numbers in counting objects. One can for example, indicate the occurring objects, each in turn, with a corresponding occurrence of the symbol 1. Thus the numbers 1, 2, 3, ... can be represented by

$$1, 11, 111, \ldots,$$

that is by finite sequences of 1, where 0 corresponds to the empty sequence. Of course 1 is not considered here as a numeral of the corresponding number in a number system. If the length of x is at least as big as the length of y, then of course $|x-y|$ is obtained if we omit those 1's from x that are in y. If nothing is left, then $|x-y|=0$. With the above notation the natural numbers yield a special word set, namely a word set over an alphabet consisting of a single letter, which is denoted by 1.

If we have two letters, the subtraction does not make sense anymore. If for example

$$a_1, a_2 \in A, \quad a_1 \neq a_2,$$

how could one "subtract" the letters of $y=a_2 \ldots a_2$ from $x=a_1 \ldots a_1$, or vice versa? It is therefore reasonable to add a characteristic function $\mathrm{eq}(x, y)$ to the initial functions, for example

$$\mathrm{eq}(x, y) = \begin{cases} \wedge, & \text{if} \quad x = y \\ a_0 & \text{otherwise,} \end{cases}$$

where a_0 is a fixed element of A.

The same is true for $\mathrm{lb}(x)$, namely that it has to be added to the initial functions. It was not a coincidence that $\mathrm{lb}(x)$ appeared in our first recursive definition of a word function. Now observe that (3.2.1) is not a perfect analogue of definition (2.1.1). Such an analogue would read

$$\begin{cases} f(\wedge) = g \\ f(xa) = h(x, f(x)), \quad \text{for} \quad a \in A. \end{cases}$$

Here $x=\mathrm{at}(xa)$, and thus the value of f anywhere would only depend on the initial part at that place, completely independently of its last letter. It is clearly not desirable to restrict the class of primitive recursive functions in this way. The possible dependence on $\mathrm{lb}(x)$ must somehow be ensured

in the defining system [11]. In (3.2.1) this is achieved by putting x, together with its last letter, as the first argument of h, instead of $\mathrm{at}(x)$. In the numeric case (in which the alphabet contains only 1) this is of course irrelevant. From $\mathrm{at}(x)$ we obtain x by simply attaching 1 to it.

3.3.2 The Idea of a Predecessor

It is actually quite arbitrary to consider only the initial segments of a word as its predecessor, for example

$$x = a_1 a_2 a_3 \quad (a_1, a_2, a_3 \in A),$$

$$\Lambda, a_1, a_1 a_2\ a_1 a_2 a_3.$$

At first I have considered all "connected pieces" of x as its predecessors. In the above example they are

$$\Lambda, a_1, a_2, a_1 a_2, a_3, a_2 a_3, a_1 a_2 a_3.$$

Since not every predecessor of x is also that of the initial part $\mathrm{at}(x) = a_1 a_2$, in addition to $\mathrm{at}(x)$, I have taken the final part (obtained by dropping the first letter), as an immediate predecessor of x. In our example this is $\mathrm{et}(x) = a_2 a_3$. Every proper predecessor of x is then a predecessor of at least one of its immediate predecessors. Accordingly, in a primitive recursion, the value of f at x is determined in terms of both $f(\mathrm{at}(x))$ and $f(\mathrm{et}(x))$.

In certain applications, however, even this turned out to be insufficient. In an application of recursive word functions to mathematical grammars [12], I found it necessary also to consider random pieces of x as its predecessors. In the above example $a_1 a_3$ is one. For the case of countable alphabets, these helped the representation (by means of a coding) only in the numeric case, where the notions of initial piece, connected piece, and random piece coincide. Indeed, in this case what matters is only the number of letters in a word and the value of f only depends on the digit of the letter in the word. In the numeric case

$$a_1 = a_2 = a_3 = 1,$$

[11] In my paper quoted in footnote [10], as well as in several further papers, this was ensured by taking the auxiliary function h to be dependent on a (the last letter of the argument). This requires a separate defining equation for every letter from the alphabet, and possibly an infinite number of equations. We could obtain constructive definitions from these by suitable restrictions, corresponding to the applications under consideration. Both $\mathrm{lb}(x)$ and $\mathrm{eq}(x, y)$ can be defined by such primitive recursions.

[12] R. Péter: *Zur Rekursivität der mathematischen Grammatiken,* Computational Linguistics Budapest **9** (1973) pp. 133–216, Submitted in December 1969.

and the variable here is $x=111$. The different parts of this in any case can only be Λ, 1, 11, or 111.
Also the random part is identical with 11.
Thus we see that, if the alphabet is countable, we can restrict ourselves to the case in which the predecessors of a word are its initial segments, and primitive recursions have the form (3.2.1) [13].

3.3.3 The Order of a Word

A word over a non-empty alphabet A can be obtained from Λ by means of as many applications of the successor functions as its *order*, i.e. the number of letters it contains. Thus

$$x = a_1 a_2 a_3 \quad (a_1, a_2, a_3 \in A)$$

is obtained from Λ and the successor functions

$$f_1(x) = xa_1, \quad f_2(x) = xa_2, \quad f_3(x) = xa_3$$

by the following substitutions: –

$$g_1(x) = f_3(f_2(x)) = xa_2a_3,$$

$$g_2(x) = g_1(f_1(x)) = xa_1a_2a_3,$$

$$g_2(\Lambda) = a_1a_2a_3.$$

3.4 Representing Natural Numbers

The order of a word, which will play an important role in what follows, is a natural number. This makes it desirable that the natural numbers be present in the word set. The steps of an enumeration can be indicated by a fixed element of the alphabet. In a word set M, which is concerned with the basic code of a computer, I shall always choose 1 as this element. Hence the natural numbers will be identified with the words consisting of 1's only. In particular, 0 will be identified with Λ. Thus there arises a double meaning of the words consisting solely of 1's. This will not lead to confusion, if the symbol of an operation, meant to be carried out digit-wise, is distinguished from the symbol of the corresponding absolute operation,

[13] See: *Lisp 1.5-Programmers Manual,* The Computation Center and Research Laboratory of Electronics. Massachusetts Inst. of Technology (1962).

which is independent of the number system, by writing 2 under the former (for example, $\overset{+}{}_2$).
Now, the order $o(x)$ of the word x can be defined by the following primitive recursion:

$$o(x) = \begin{cases} \wedge, & \text{if } x = \wedge \\ o(\mathrm{at}(x))1 & \text{otherwise} \end{cases}$$

where $o(\mathrm{at}(x))1$ is obtained from the successor function $x1$ by substitution. Clearly, x is a natural number if and only if

$$x = o(x).$$

It follows that the iteration of $o(x)$ does not change anything: –

$$o(o(x)) = o(x).$$

3.4.1 Number Functions and Word Set Relations

In this way, every numeric primitive recursive function can be represented by a primitive recursive function in the word set. In order to see this we shall prove the following: To every primitive recursive numeric function $\varphi(n_1, \ldots, n_r)$ there is a primitive recursive word function $f(x_1, \ldots, x_r)$ such that

$$o(f(x_1, \ldots, x_r)) = \varphi(o(x_1), \ldots, o(x_r)) \tag{3.4.1}$$

holds for all $x_1, \ldots, x_r$.

Proof. Firstly, this is true for the initial functions 0 and $n+1$; for $\varphi=0$ we can take $f=\wedge$, and for $\varphi(n)=n+1$ for example $f(x)=x1$.
Next, this property is preserved by substitution: If (3.4.1) holds for

$$\varphi_1(n_1, \ldots, n_r), \ldots, \varphi_k(n_1, \ldots, n_r), \psi(m_1, \ldots, m_k)$$

with

$$f_1(x_1, \ldots, x_r), \ldots, f_k(x_1, \ldots, x_r), g(y_1, \ldots, y_k)$$

respectively, then (3.4.1) also holds for $\psi(\varphi_1, \ldots, \varphi_k)$ with $g(f_1, \ldots, f_k)$, as the assumptions imply

$$\begin{aligned} &o(g(f_1(x_1, \ldots, x_r), \ldots, f_k(x_1, \ldots, x_r))) = \\ &= \psi(o(f_1(x_1, \ldots, x_r)), \ldots, o(f_k(x_1, \ldots, x_r))) = \\ &= \psi(\varphi_1(o(x_1), \ldots, o(x_r)), \ldots, \varphi_k(o(x_1), \ldots, o(x_r))). \end{aligned}$$

Finally, this property is also preserved under primitive recursion. Indeed, let φ be determined by the primitive recursion

$$\begin{cases}\varphi(0, n_1, \ldots, n_r) = \alpha(n_1, \ldots, n_r) \\ \varphi(n+1, n_1, \ldots, n_r) = \beta(n, n_1, \ldots, n_r, \varphi(n, n_1, \ldots, n_r))\end{cases}$$

where (3.4.1) holds for

$$\alpha(n_1, \ldots, n_r) \quad \text{and} \quad \beta(n, n_1, \ldots, n_r, m)$$

with

$$g(x_1, \ldots, x_r) \quad \text{and} \quad h(x, x_1, \ldots, x_r, y),$$

respectively. Then (3.4.1) also holds for φ with the word function f defined by the following primitive recursion:

$$f(x, x_1, \ldots, x_r) = \begin{cases} g(x_1, \ldots, x_r), & \text{if} \quad x = \wedge \\ h(\text{at}(x), x_1, \ldots, x_r, f(\text{at}(x), x_1, \ldots, x_r))) & \text{otherwise.}\end{cases}$$

Indeed, by our original assumption,

$$\begin{aligned} o(f(\wedge, x_1, \ldots, x_r)) = o(g(x_1, \ldots, x_r)) = \alpha(o(x_1), \ldots, o(x_r)) = \\ = \varphi(0, o(x_1), \ldots, o(x_r)) = \\ = \varphi(o(\wedge), o(x_1), \ldots, o(x_r)). \end{aligned}$$

Here we used $o(\wedge)=0$.

Now suppose that, for some natural number n, for every x with $o(x)=n$ we have

$$o(f(x, x_1, \ldots, x_r)) = \varphi(o(x), o(x_1), \ldots, o(x_r)).$$

Then the same is also valid for each x of order $o(x)=n+1$, since for such an x we have $o(\text{at}(x))=n$, and so by assumption

$$\begin{aligned} &o(f(x, x_1, \ldots, x_r)) = o(h(\text{at}(x), x_1, \ldots, x_r, f(\text{at}(x), x_1, \ldots, x_r))) = \\ &= \beta(o(\text{at}(x)), o(x_1), \ldots, o(x_r), o(f(\text{at}(x), x_1, \ldots, x_r))) = \\ &= \beta(o(\text{at}(x)), o(x_1), \ldots, o(x_r), \varphi(o(\text{at}(x)), o(x_1), \ldots, o(x_r))) = \\ &= \varphi(o(\text{at}(x))+1, o(x_1), \ldots, o(x_r)) = \\ &= \varphi(o(x), o(x_1), \ldots, o(x_r)). \end{aligned}$$

Replacing in (3.4.1) each $x_1, \ldots, x_r$ in turn with $o(x_1), \ldots, o(x_r)$ we obtain,

$$o(f(o(x_1), \ldots, o(x_r))) = \varphi(o(x_1), \ldots, o(x_r)).$$

On the left-hand side, there stands a primitive recursive word function, which at every place depends only on the order of its arguments, i.e. on natural numbers, and also takes natural numbers as values. Clearly, for natural numbers its value is equal to the value of φ. In particular, it vanishes

for the same natural numbers as φ. Thus it can be considered as a representative of φ in the word set. In what follows the representatives of numeric functions, as functions of the orders of their arguments, will be denoted in the same way as the corresponding numeric functions.
This also applies to the numeric relations; they can likewise be represented by primitive recursive relations in the word set, for the primitive recursivity of a relation means here also the existence of a primitive recursive characteristic function, which vanishes if the relation is satisfied, and otherwise can be defined to be 1. Since $\wedge$ is the empty sequence, "vanishes" can be taken here literally.

3.4.2 Examples

In our word set the counterparts of the numeric functions $\mathrm{sg}(x)$ and $\overline{\mathrm{sg}}(x)$ can be defined by the following primitive recursions:

$$\mathrm{sig}(x) = \begin{cases} \wedge, & \text{if } x = \wedge \\ 1 & \text{otherwise,} \end{cases}$$

$$\overline{\mathrm{sig}}(x) = \begin{cases} 1, & \text{if } x = \wedge \\ \wedge & \text{otherwise,} \end{cases}$$

and these are characteristic functions of the relations

$$x = \wedge \quad \text{and} \quad x \neq \wedge.$$

Exactly as in the numeric case, it follows also here that if the relation B is primitive recursive, then so is its negation $\overline{B}$. If b is a characteristic function of B, then $\overline{\mathrm{sig}}(b)$ is a characteristic function of $\overline{B}$.

However, not everything can be copied from number theory. We found there that the disjunction $B_1 \vee B_2$ of the primitive recursive relations B_1 and B_2 is also primitive recursive by multiplying their characteristic functions b_1 and b_2. In the word set we use the following trick: Let

$$d(x, y) = \begin{cases} \wedge, & \text{if } x = \wedge \\ \mathrm{sig}(y) & \text{otherwise.} \end{cases}$$

Then $d(x, y)$ vanishes if and only if at least one of x and y does, that is

$$d(b_1, b_2)$$

is a characteristic function of $B_1 \vee B_2$.

The primitive recursivity of the conjunction $B_1 \& B_2$ now follows from the above, for it is equivalent to

$$\overline{\bar{B}_1 \vee \bar{B}_2},$$

and thus

$$\overline{\operatorname{sig}}\,(d(\overline{\operatorname{sig}}\,(b_1), \overline{\operatorname{sig}}\,(b_2)))$$

is a characteristic function of $B_1 \& B_2$.
Similarly

$$B_1 \to B_2 = \bar{B}_1 \vee B_2$$

implies that, together with B_1 and B_2, $B_1 \to B_2$ is also primitive recursive.

3.5 Definition by Cases

Now we can turn to the definitions by cases (in analogy with section 2.5.1), which have the form

$$f = \begin{cases} g_1, & \text{if } B_1 \text{ is true} \\ \dots\dots\dots\dots\dots\dots \\ g_{k-1}, & \text{if } B_{k-1} \text{ is true} \\ g_k & \text{otherwise,} \end{cases}$$

where $g_1, \dots, g_k$ are primitive recursive word functions and $B_1, \dots, B_{k-1}$ are primitive recursive relations, and "otherwise" means that

$$B_k = \bar{B}_1 \,\&\, \dots \,\&\, \bar{B}_{k-1}$$

is satisfied.
In the numeric case the corresponding function was

$$\alpha_1 \cdot \overline{\operatorname{sg}}\,(\beta_1) + \dots + \alpha_k \cdot \overline{\operatorname{sg}}\,(\beta_k),$$

where

$$n \cdot \overline{\operatorname{sg}}\,(m) = \begin{cases} n, & \text{if } m = 0 \\ 0 & \text{otherwise.} \end{cases}$$

was the property of the product that we needed. The property of the sum we use is that it is equal to one of the summands if all the others vanish. In the present context we have to do something similar without multiplication and addition.
The first case is easy; we simply put

$$s_1(x, y) = \begin{cases} y, & \text{if } x = \wedge \\ \wedge & \text{otherwise,} \end{cases}$$

The following definitions lead us to the remaining cases

$$s_2(y_1, y_2) = \begin{cases} y_2, & \text{if } y_1 = \Lambda \\ y_1 & \text{otherwise,} \end{cases}$$

$$s_3(y_1, y_2, y_3) = s_2(s_2(y_1, y_2), y_3),$$

$$s_4(y_1, y_2, y_3, y_4) = s_2(s_3(y_1, y_2, y_3), y_4),$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

$$s_k(y_1, \ldots, y_k) = s_2(s_{k-1}(y_1, \ldots, y_{k-1}), y_k).$$

For $i=2, 3, \ldots, k$ and $j=1, 2, \ldots, i$ we have

$$s_i(\Lambda, \ldots, \Lambda, y_j, \Lambda, \ldots, \Lambda) = y_j,$$

hence for $y_j = \Lambda$

$$s_i(\Lambda, \ldots, \Lambda) = \Lambda.$$

Using the definition of s_2

$$s_2(\Lambda, y_2) = y_2,$$

since

$$s_2(y_1, \Lambda) = \begin{cases} \Lambda, & \text{if } y_1 = \Lambda \\ y_1 & \text{otherwise,} \end{cases}$$

we have in any case

$$s_2(y_1, \Lambda) = y_1.$$

It follows from

$$s_{i+1}(y_1, \ldots, y_{i+1}) = s_2(s_i(y_1, \ldots, y_i), y_{i+1})$$

that if the property under consideration holds for an s_i with $i<k$, then it also holds for s_{i+1}. Indeed, for $j \neq i+1$ we have

$$\begin{aligned} s_{i+1}(\Lambda, \ldots, \Lambda, y_j, \Lambda, \ldots, \Lambda) &= s_2(s_i(\Lambda, \ldots, \Lambda, y_j, \Lambda, \ldots, \Lambda), \Lambda) = \\ &= s_i(\Lambda, \ldots, \Lambda, y_j, \Lambda, \ldots, \Lambda) = \\ &= y_j, \end{aligned}$$

and for $j=i+1$

$$\begin{aligned} s_{i+1}(\Lambda, \ldots, \Lambda, y_{i+1}) &= s_2(s_i(\Lambda, \ldots, \Lambda), y_{i+1}) = \\ &= s_2(\Lambda, y_{i+1}) = \\ &= y_{i+1}. \end{aligned}$$

Thus s_k has the desired property of the sum. It follows that if $b_1, \ldots, b_k$ are characteristic functions of $B_1, \ldots, B_k$, respectively, then the built up function f is obtained as

$$f = s_k(s_1(b_1, g_1), s_1(b_2, g_2), \ldots, s_1(b_k, g_k)).$$

Consequently it is primitive recursive.

3.5.1 Initial Segments

In view of the above, it follows that

$$f(x, y, z) = \begin{cases} \wedge, & \text{if } o(x) < o(y) \\ x, & \text{if } o(x) = o(y) \\ z, & \text{if } o(x) > o(y) \end{cases}$$

is a primitive recursive word function. This can be used to give a primitive recursive definition of the initial segment of x consisting of $o(y)$ letters (denoted by $a(x, y)$), which is meant to be $\wedge$ if $o(x) < o(y)$, and, of course, x itself if $o(x) = o(y)$. Since for $o(x) > o(y)$, this is the same as the corresponding initial segment of $\mathrm{at}(x)$, using the above f it can be defined by

$$a(x, y) = \begin{cases} \wedge, & \text{if } x = \wedge \\ f(x, y, a(\mathrm{at}(x), y)) & \text{otherwise.} \end{cases}$$

In particular, we obtain that

$$\mathrm{at}(x) = a\big(x, o(x) \dot{-} o(1)\big)$$

is a primitive recursive word function. Moreover the $o(y)$th letter $b_l(x, y)$ from the left in x can be written as

$$b_l(x, y) = \mathrm{lb}\,(a(x, y)),$$

which implies that the $o(y)$th letter $b_r(x, y)$ from the right in x is

$$b_r(x, y) = b_l(x, o(x1) \dot{-} o(y)).$$

Thus we have solved our problem from the beginning of this chapter: the function $h(x, y)$ defined in section 33, hence also $s(x)$, the binary form of the successor of a natural number of the binary form x, (defined in section 3.1) are primitive recursive word functions over an alphabet containing 0 and 1, provided that, in addition to the successor functions, the identity function, a characteristic function of the equality, and $\mathrm{lb}(x)$ are taken to be initial functions.

3.6 Basic Operations in Binary Form

In such a word set M the binary forms of the results of the digitwise operations of Ch. 1 turn out to be primitive recursive. Indeed, the binary form of a natural number is a word, and its nth digit from the right (which is actually the $(n+1)$th as the first one has index 0), is the same as the nth letter, from the right, of the word. Every number n can be written in the form $o(y)$, and

$b_r(x, y)$, the $o(y)$th letter from the right in x is primitive recursive. It can be seen from its meaning, but also from the construction of the functions

$$f(x, y, z), \quad a(x, y), \quad b_l(x, y), \quad b_r(x, y),$$

that we have

$$b_r(x, y) = b_r(x, o(y)).$$

Let us consider for example the addition of two binary forms x and y. It is easy to check that if we denote the "carry" and the resulting digit at the $o(z)$th place from the right by $u(x, y, z)$ and $s(x, y, z)$, respectively, then definition (1.2.1) can be formulated in M as follows:
Putting first the variable w in place of the function u to be defined, we obtain for $n = o(z) \neq \Lambda$ (i.e. for $z \neq \Lambda$) the auxiliary function

$$h(x, y, z, w) = \begin{cases} 1, & \text{if} \quad b_r(x, z) = b_r(y, z) = 1 \vee b_r(x, z) \\ & \qquad = w = 1 \vee b_r(y, z) \\ & \qquad = w = 1 \\ 0 & \text{otherwise,} \end{cases}$$

which, by definition, is identical with $h(x, y, o(z), w)$.

3.6.1 Concatenation

A similar statement holds, consequently, for the following functions defined by means of h. We have

$$u(x, y, z) = \begin{cases} 0, & \text{if} \quad z = \Lambda \\ h(x, y, \text{at}(z), u(x, y, \text{at}(z))) & \text{otherwise,} \end{cases}$$

and

$$s(x, y, z) = \begin{cases} 1, & \text{if} \quad (u(x, y, z) = 0 \,\&\, b_r(x, z) \neq b_r(y, z)) \vee \\ & \qquad \vee (u(x, y, z) = 1 \,\&\, b_r(x, z) = b_r(y, z)) \\ 0 & \text{otherwise.} \end{cases}$$

It is irrelevant what this gives for words consisting not only of 0's and 1's. The elements of the binary form of $x + y$, which result from the digits obtained step by step as above, will be denoted by $t(x, y, o(z))$. This can be defined by

$$t(x, y, z) = \begin{cases} \Lambda, & \text{if} \quad z = \Lambda \\ s(x, y, \text{at}(z))t(x, y, \text{at}(z)) & \text{otherwise.} \end{cases}$$

Indeed, denoting the digit

$$s(x, y, z) = s(x, y, o(z))$$

simply by $s_o(z)$, we obtain

$$\begin{aligned} t(x, y, \wedge) &= \wedge \\ t(x, y, 1) &= s(x, y, \wedge)\wedge = s_\wedge \\ t(x, y, 11) &= s(x, y, 1)s_\wedge = s_1 s_\wedge \\ t(x, y, 111) &= s(x, y, 11)s_1 s_\wedge = s_{11} s_1 s_\wedge \end{aligned}$$

and so on, where $\wedge$, 1, 11, 111, ... represent in M the natural numbers 0, 1, 2, 3, ..., and are not considered as binary forms. With

$$\max(x, y) = \begin{cases} o(x), & \text{if} \quad o(x) \geqq o(y) \\ o(y), & \text{if} \quad o(x) \prec o(y) \end{cases}$$

the binary form of $x \overset{+}{_2} y$ is

$$t(x, y, \max(x, y)1),$$

with at most one unnecessary 0 at the left end of the word, which could easily be eliminated.

In the definition of t, however, we applied the *concatenation* of two words. This is such an important operation in word sets that in general it has to be added to the initial functions. But if the alphabet is finite, and we can restrict ourselves to this case because a computer can recognize only finitely many symbols, it can be shown easily that

$$f(x, y) = xy$$

is primitive recursive.

3.6.2 More Primitive Recursive Word Functions

In this section I will list a few more primitive recursive word functions and relations, without giving proofs.

The "final segments" of order $o(y)$ of x, denoted by $e(x, y)$ and in particular the "final part" $\mathrm{et}(x)$ of x, obtained by omitting its first letter, are also primitive recursive. The first letter $\mathrm{eb}(x)$ of x, being equal to $a(x, 1)$, has already been shown to be primitive recursive.

The relation "y is a predecessor (i.e. initial segment) of x", denoted shortly by $y \trianglelefteq x$, is primitive recursive. This occurs in the relations

$$(\mathrm{E}y)[y \trianglelefteq x \,\&\, B(y, x_1, \ldots, x_r)],$$

$$(x)[y \trianglelefteq x \rightarrow B(y, x_1, \ldots, x_r)],$$

and in the function

$$\mu_y[y \trianglelefteq x \,\&\, B(y, x_1, \ldots, x_r)]$$

(the "bounded μ-operation"), which respectively have the following meanings: –
"There is a predecessor y of x such that $B(y, x_1, \ldots, x_r)$ holds";
"For every predecessor y of x $B(y, x_1, \ldots, x_r)$ holds"; and
"A fixed predecessor y of x, for which $B(y, x_1, \ldots, x_r)$ holds, if there is such; $\&\wedge$ otherwise".
If B is primitive recursive, each one of these is also. In fact so is every set having a numeric structure.

3.7 List Processing

As an application, we consider the basic notions of "list processing", which has been used frequently as a kind of model of the complex relations between the different kinds of information stored in a computer.
What we have to deal with here are finite linear arrays, called "lists", which are constructed from certain elements. The empty array NIL is the only object which can be considered both as an element and a list. A list l has the form

$$l = (x_1, \ldots, x_n),$$

where every x_i is either an element or a list. According to the above we have

$$\text{NIL} = (\text{NIL}),$$

but if x is different from NIL, then x and (x) are distinct. We also make the convention that

$$(x_1, \ldots, x_n) = (x_1, \ldots, x_n, \text{NIL})$$

holds.
In the above list l the entry x_1 is called the *head* and the list

$$(x_2, \ldots, x_n)$$

remaining after x_1 is omitted, is called the *tail* of l. In notation

$$x_1 = \text{hd}\,[l], \quad (x_2, \ldots, x_n) = \text{tl}\,[l].$$

(In order to avoid misunderstandings, the arguments of list functions will be put in square brackets.)
By means of the function cons, l can be recovered from its head and tail: If x is an element or a list and y is a list, then cons $[x, y]$ is the list with x as head and y as tail. Thus for the above l: –

$$\text{cons}\,[x_1, (x_2, \ldots, x_n)] = l.$$

3.8 Coding Sequences of Words

In a formal sense the lists are words of a word set $M^{(l)}$ over an alphabet $A^{(l)}$ containing the elements, the parantheses, and the comma. (The parentheses and the comma will be printed boldface, when considered as letters.)
The functions hd[x], tl[x] and cons [x, y] are primitive recursive in $M^{(l)}$.
Let us first observe the heads of several lists:

$$\mathrm{hd}\,[(x_1, x_2, x_3)] = x_1$$
$$\mathrm{hd}\,[((x_1, x_2), x_3)] = (x_1, x_2)$$
$$\mathrm{hd}\,[((x_1, (x_2, x_3)), (x_4, x_5), x_6)] = (x_1, (x_2, x_3))$$
$$\mathrm{hd}\,[(((x)), y)] = ((x))$$
$$\mathrm{hd}\,[((x))] = \mathrm{hd}\,[((x), \mathrm{NIL})] = (x)$$

and so on.

As can be seen, the head of a list x is obtained by removing its opening parenthesis, that is taking its final part et(x), and then the initial segment of smallest order of et(x) for which the following relation B holds: the number of its left parentheses coincides with the number of its right parentheses.
In order to check, which is the initial segment of smallest order of a word x satisfying B, one can examine the letters of x one by one, starting from the left. At a letter different from the parentheses we do nothing; at a left parenthesis we write down an a_0 (where a_0 can be e.g. a fixed element), at a right parenthesis we erase one of the a_0s already written down. The first time that $\wedge$ is obtained in this way is when we have the shortest initial segment with property B. The function kl(x, y), defined by primitive recursion in $M^{(l)}$, does exactly this for $o(y)=1, 2, \ldots$:

$$\mathrm{kl}\,(x, y) = \begin{cases} \wedge, & \text{if} \quad y = \wedge \\ \mathrm{kl}\,(x, \mathrm{at}\,(t)), & \text{if} \quad b_l(x, y) \neq (\,\&\, b_l(x, y) \neq) \\ \mathrm{kl}\,(x, \mathrm{at}\,(y))\,a_0, & \text{if} \quad b_l(x, y) = (\\ \mathrm{at}\,(\mathrm{kl}\,(x, \mathrm{at}\,(y))), & \text{if} \quad b_l(x, y) =). \end{cases}$$

Here we used the function $b_l(x, y)$ (the $o(y)$th letter of x from the left), which was introduced in section (3.5.1).
If this is applied to et(x), where x is a list, then reaching the smallest $o(y) \neq \wedge$ such that kl[at(x), y]$=\wedge$ means that we have reached the last letter of hd[x]. Since the initial segment of x of order $o(y)$ is denoted by $a(x, y)$, we have then

$$\mathrm{hd}\,[x] = a\big(\mathrm{et}\,(x), \mu_y\,[y \preceq o\big(\mathrm{et}\,(x)\big)\,\&\, y \neq \wedge\,\&\,\mathrm{kl}\,\big(\mathrm{et}\,(x), y\big) = \wedge]\big),$$

if x is a list. For words that are not lists the value of hd is irrelevant. The same applies to the following: –
If x is a list, then we can obtain its tail tl[x] by first removing its initial segment

$$(\text{hd}\,[x],$$

and then attaching an opening parenthesis to the front of the remaining word of order

$$o(x) \dot{-} o((\text{hd}\,[x],).$$

Since the final segment of x of order $o(y)$ is denoted by $e(x, y)$, we have

$$\text{tl}\,[x] = (e(x, o(x) \dot{-} o((\text{hd}\,[x],)).$$

Finally if x is an element or a list and y is a list, cons [x, y] is built by removing the opening parenthesis of y, whereby we obtain et(y). Then we put

$$\text{cons}\,[x, y] = (x, \text{et}\,(y)).$$

All the further notions of list processing can be shown to be primitive recursive in $M^{(l)}$ in a similar way.
From this basis of list processing is constructed the programming language LISP 1.5. For more details about this see Chapter 11.

3.8.1 General and Partial Word Functions

Generalizations of primitive recursion similar to the numeric case can also be introduced into word sets. These can or cannot be reduced to primitive recursion, just like their counterparts in the theory of numbers.
The methods, however, cannot be copied. We do not have here, for example, any unique prime factor representation. In the numeric case this was essential in order to reduce the course-of-values recursion to a primitive recursion, by establishing a coding of finite sequences of integers by single numbers.
What we need here is a correspondence between a finite sequence of words and a single word, from which the terms of the sequence can be recovered. The simple concatenation of the members of the sequence clearly will not do, unless certain "separating symbols" are used between the words. Clearly this could be achieved by taking a new letter as a separating symbol. However, it is still possible without extending the alphabet to produce separating symbols out of two fixed letters. These in what follows, will be denoted by 0 and 1. If the alphabet consists of a single letter, as in the numeric case, this method is not applicable.

For a finite sequence of words

$$x_0, x_1, \ldots, x_n$$

in M, we can determine a suitable separating symbol as follows. Let

$$\underbrace{11 \ldots 1}_{i}$$

denote the word consisting of i ones. Let i be the largest number, for which such a word occurs as a connected piece in at least one of the words $x_0, x_1, \ldots, x_n$. Then

$$\underbrace{011 \ldots 10}_{i+1}$$

is a suitable separating symbol for our sequence. This can be made to correspond to the word

$$c_n(x_0, x_1, \ldots, x_n) = x_0 \underbrace{011 \ldots 10}_{i+1} x_1 \underbrace{011 \ldots 10}_{i+1} \ldots x_n \underbrace{011 \ldots 10}_{i+1}$$

which depends primitive recursively on the members of the sequence.

The relation "x is a word that corresponds to a finite sequence of words" is primitive recursive. So are the number of terms denoted by long (x), and the $o(y)$th term $k_{0(y)}(x)$, for $o(y)=0, 1, 2, \ldots,$ long (x), of the sequence corresponding to x.

The notions of general and partial recursive functions can be transferred to word sets in the same way. If the alphabet is countable, they can be obtained from the primitive recursive functions by substitutions and the applications of unbounded μ-operations, though the meaning of this last concept has still to be clarified. In the theory of numbers, this meant the smallest number with a given property, but what do we mean by "the smallest word with a given property"? What we can have is a word of smallest index in a given infinite sequence of words. Such a sequence can be considered as a word function $f(o(x))$, which depends only on the order of x and possibly on the order of other variables. $f(o(x))$ is the $o(x)$th term of the sequence. If $f(o(x))$ is primitive, general or partial recursive, we say that the sequence is primitive, general, or partial recursive, respectively. According to this the unbounded μ-operation

$$\mu_{f(o(y))}[B(f(o(y)), x_1, \ldots, x_r)]$$

means the value $f(o(y))$ of the smallest index for which $B(f(o(y)), x_1, \ldots, x_r)$ holds, provided that to the arguments $x_1, \ldots, x_r$ under consideration there is a y with $B(f(o(y)), x_1, \ldots, x_r)$. Otherwise the result of the operation is undefined for these arguments.

3.9 McCarthy's Conditions

I would like to remark that in the considerations of section 3.5 on built up definitions, we have hardly used the fact that we are dealing with word sets. With slight changes these arguments can be carried out in general for sets of numeric structure. The conditional expression

$$f = [B_0 \rightarrow e_1, \ldots, B_n \rightarrow e_n],$$

introduced by McCarthy in connection with partial computations by the computer (see the book quoted in footnote[1]), does not extend outside the class of partial recursive functions over a set of numeric structure either. Here $e_0, \ldots, e_n$ are symbolic expressions and $B_0, \ldots, B_n$ are relations between such expressions, while all of these can be undefined. The order of the expressions is essential, as follows from the following definition of the meaning of the conditional expression f: f is undefined, if either none of the B_is is true, or if to the left of the first true B_i there is an undefined B_j; otherwise the value of f is that e_i towards which the arrow of the first true B_i points.

Let us suppose that in a non-trivial set of expressions there are two different elements, say e' and e''. Using these, we have corresponding to each relation B the characteristic function

$$b = \begin{cases} e', & \text{if } B \text{ is true} \\ e'', & \text{if } \bar{B} \text{ is true,} \end{cases}$$

which is defined if and only if B is defined. For an undefined B we have neither B, nor $\bar{B}$, i.e. neither $b=e'$, nor $b=e''$). Thus McCarthy's definition is equivalent to the following built up definition: –

$$f = \begin{cases} e_0, & \text{if } b_0 = e' \\ e_1, & \text{if } b_0 = e'' \,\&\, b_1 = e' \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ e_n, & \text{if } b_0 = e'' \,\&\ldots\&\, b_{n-1} = e'' \,\&\, b_n = e'. \end{cases}$$

It can be shown that if $e_0, \ldots, e_n$ and $b_0, \ldots, b_n$ are partial recursive in a set H of numeric structrure (that is in a holomorphic set), then so is the function f defined above.

Chapter 4

The Recursivity of Everything Computable

4.1 Assembly Language

The words of the binary language of a computer, consisting solely of the letters 0 and 1, are difficult for people to understand. In an assembly language, these are replaced by sentences of symbols that reflect their meaning, yet they can still be translated easily back to the language of computer and the addresses within the computer can be denoted by numbers in their ordinary decimal form. The programs in this chapter will be written in such an assembly language.

For a computer with a very simple system of statements it can be shown that, if no bound is put on the size of its memory, for every partial recursive function there is a program such that computation with this program yields the value of the function, if it is defined, and goes on forever, without calculating anything if it is not [14].

I will restrict myself to two registers: the statement counter W and a result register E. We also have the following one-address statements, where addresses are always positive numbers; (x) denotes the contents of the address or register x, $x \Rightarrow (y)$ denotes putting x into y with the erasure of the earlier contents of y, and finally $0 \Rightarrow (x)$ means the deletion of x: –

La (load statement): $(a) \Rightarrow (E)$

SPa (store statement): $(E) \Rightarrow (a)$

SP^0a (store and delete statement): $(E) \Rightarrow (a);\ 0 \Rightarrow (E)$

Aa (addition statement): $(E)+(a) \Rightarrow (E)$

Sa (subtraction statement): $(E)-(a) \Rightarrow (E)$

[14] See J. C. Shepherdson and H. E. Sturgis: *Computability of recursive functions,* Journ. of the ACM **10** (1963) pp. 217–255, and R. Péter: *Programmierung und partiell-rekursive Functionen,* Acta Math. Acad. Sci. Hung. **14** (1963) pp. 373–401.

Ma (multiplication statement): $(E)\times(a)\Rightarrow(E)$

Ga (go to statement): $a\Rightarrow(W)$

$\left.\begin{matrix}G^{=}a\\G^{>}a\\G^{\geqq}a\end{matrix}\right\}$ (conditional go to statement): $a\Rightarrow(W)$ if $\begin{cases}(E)=0\\(E)>0\\(E)\geqq 0\end{cases}$

ST (stop statement): Stop.

In computing the values of a numeric function, which I will simply call computing the function, we need the arithmetical difference $a\dot{-}n$ instead of $a-n$. This can be computed by means of the following program:

Initially, let the contents of the addresses 1 and 2 be the given arguments a and n. Also let the contents of the following addresses be the following statements: –

address	statement	result
3.	$L1$	$(E)=a$
4.	$S2$	$(E)=a-n$
5.	$G^{\geqq}7$	$\begin{cases}\text{If } a\geqq n \text{ go to address 7,}\\\text{i.e. } (W)=7\end{cases}$
6.	$SP^{0}8$	$(E)=0$
7.	ST	Stop.

If "Stop" is reached, then $(E)=a\dot{-}n$.

4.2 Computing $[\sqrt{n}]$

As a somewhat more complicated example, we consider a program to compute $[\sqrt{n}]$. This is the smallest number i, for which $(i+1)^2>n$, therefore it is certainly not greater than n. Hence

$$[\sqrt{n}]=\mu_i[i\leqq n\,\&\,(i+1)^2>n]$$

is a primitive recursive function.

To start with, let $1, n, i=0, 0$ be the contents of the addresses 1, 2, 3, 4, respectively. Address 4 serves as a working space. Let the contents of the following addresses be: –

address	statement	result
5.	$L3$	$(E) = i$
6.	$A1$	$(E) = i+1$
7.	$SP4$	$(4) = i+1$
8.	$M4$	$(E) = (i+1)^2$
9.	$S2$	$(E) = (i+1)^2 - n$
10.	$G>14$	If $(i+1)^2 > n$, go to address 14 i.e. $(W) = 14$
11.	$L4$	$(E) = i+1$
12.	$SP3$	$(3) = i+1$
13.	$G6$	$(W) = 6$; here everything starts anew, with $i+1$ as the new i
14.	$L3$	(E) = the required value of i
15.	ST	Stop.

If "Stop" is reached, then $(E) = [\sqrt{n}]$.
The program looks circular, since from the statement in address 13 everything starts anew. However, with $i+1$ instead of i, it only goes on until i, increased by 1 for each new start, does not satisfy $(i+1)^2 > n$. Then the statement at address 10 orders to jump out of the "circle". The statements under the addresses 6—13 form a cycle, which is not a closed circle, but rather an ever progressing spiral.

4.3 Computing Recursive Number Functions

It was mentioned in section 2.6.2 that, starting from the functions $a+n$, $a \cdot n$, $a \dot{-} n$ and $[\sqrt{n}]$ (which, according to the above, can be computed by our computer, that is they are machine computable), every primitive recursive numeric function can be obtained with the application of finitely many substitutions and iterations of the form

$$\begin{cases} \varphi(0) = 0 \\ \varphi(n+1) = \beta(\varphi(n)). \end{cases}$$

If the functions

$$\alpha(m_1, \ldots, m_r),\ \beta_1(n_1, \ldots, n_s), \ldots, \beta_r(n_1, \ldots, n_s)$$

are machine computable, then so is

$$\varphi(n_1, \ldots, n_s) = \alpha(\beta_1(n_1, \ldots, n_s), \ldots, \beta_r(n_1, \ldots, n_s))$$

which can be obtained from them by substitution. Indeed, if for any given $n_1, \ldots, n_s$ the values of $\beta_1, \ldots, \beta_r$ can be computed and stored, then the value of α can be computed for these arguments.

Now, if the function $\beta(n)$ is machine computable, then so is the function

$$\varphi(n) = \beta^{(n)}(0)$$

obtained from β by the above iteration.
Since $\varphi(0)=0$ is known, it suffices to do the computation for $n>0$.
By our assumption, we can use a "subroutine" for the calculation of $\beta(n)$. This will obtain its argument from a fixed address (here this will be address 4), and put the computed value into E. Let us add to our system of statements a subroutine calling statement.
The contents of the addresses 1, 2, 3, 4 are initially $1, n, i=1, a=0$, respectively. Here a denotes the subresult successively taking the values

$$\beta(0), \beta(\beta(0)), \beta(\beta(\beta(0))), \ldots .$$

This changes after each call of the subroutine. The contents of the following addresses are: –

address	statement	result
5.	Call of the subroutine for β	$(E) = \beta(a)$
6.	$SP4$	$(4) = \beta(a)$
7.	$L3$	$(E) = i$
8.	$S2$	$(E) = i-n$
9.	$G^{=}14$	If $i = n$ go to address 14 i.e. $(W) = 14$
10.	$L3$	$(E) = i$
11.	$A1$	$(E) = i+1$
12.	$SP^{0}3$	$(E) = 0, (3) = i+1$
13.	$G5$	$(W) = 5$, here everything starts anew with $i+1$ instead of i and $\beta(a)$ instead of a
14.	$L4$	(E) = the latest value of a
15.	ST	Stop.

As i increases gradually to n, one will get out of the cycle with

$$(E) = \beta^{(n)}(0).$$

It follows, from the above, that every primitive recursive numeric function is machine computable. This could have been obtained without computing $[\sqrt{n}]$, provided that we had not restricted ourselves to iteration, which is a particular case of primitive recursion. In the computation of $[\sqrt{n}]$, however, I wanted to give a non-trivial concrete example, to which I shall want to return later.

4.4 Computing General Recursive Functions

Machine computability is also preserved in the application of a μ-operation, not only in the bounded case, as we could see in the example of $[\sqrt{n}]$, but in the unbounded case as well.

For a relation B, whose characteristic function β is machine computable, put

$$\varphi(n) = \mu_i[B(i, n)],$$

that is

$$\varphi(n) = \mu_i[\beta(i, n) = 0].$$

For functions of several variables we can proceed similarly. By the definition in section 2.9, for an n such that there is at least one i with $\beta(i, n)=0$, $\varphi(n)$ is the smallest such i.

Assume therefore that we have a subroutine for the computation of $\beta(m, n)$ and to start with, let the contents of addresses 1, 2, 3 be 1, n, $i=0$, respectively. The contents of the following addresses are: –

address	statement	result
4.	Call of the subroutine for (3), (2)	$(E) = \beta(i, n)$
5.	$G^{=}10$	If $\beta(i, n) = 0$ go to address 10 i.e. $(W) = 10$
6.	$L3$	$(E) = i$
7.	$A1$	$(E) = i+1$
8.	$SP^{0}3$	$(E) = 0, (3) = i+1$
9.	$G4$	$(W) = 4$; here everything starts anew with $i+1$
10.	$L3$	(E) = the required i
11.	ST	Stop.

If there is an i and an n with $\beta(i, n)=0$, then we get out of the cycle with the smallest such i as (E). Otherwise the cycle never ends. The computer computes nothing, in accordance with the fact that $\varphi(n)$ is undefined.
In view of the Kleene explicit form of partial recursive functions, given in section 2.9.1, it follows from the results of this chapter that every numeric partial recursive function is machine computable. Thus, in particular, every general recursive function is machine computable.

4.5 A Universal Program

The universal Kleene explicit form of r-place partial recursive numeric functions can be made even more universal, namely independent of the number of variables. Indeed, the arguments $a_1, \ldots, a_r$ of an r-place function can be represented for example by the number

$$a = p_1^{a_1} \cdot \ldots \cdot p_r^{a_r},$$

(where, as above, p_i denotes the ith prime number). Thus we obtain the universal explicit form of partial recursive numeric functions of arbitrarily many variables (since the exponent of p_i in the prime factor representation of a was denoted by $\exp_i(a)$) as the two-place partial recursive function

$$\lambda(n, a) = \psi(\mu_i[\tau(i, n, \exp_1(a), \ldots, \exp_r(a)) = 0]),$$

where ψ and τ are fixed primitive recursive functions. If n is the Gödel number of a system of equations defining the r-place partial recursive function φ, then we have

$$\varphi(a_1, \ldots, a_r) \simeq \lambda(n, p_1^{a_1} \cdot \ldots \cdot p_r^{a_r}).$$

According to the above, the function λ, which is constructed from primitive recursive functions with the help of a single μ-operation, is machine computable. A program computing its values can be considered as a universal program. If it is stored in the memory of the computer, then the computer yields automatically the values of every partial recursive numeric function, at any point where it is defined, provided that as further information the Gödel number of a system of equations defining our function, and the arguments in question, are also stored in the machine.

The construction of a program for λ would still not be simple. That is why I have constructed [15] a universal program for the computation of all partial recursive functions, without having to resort to the Kleene explicit form. In this program, the defining systems of equations are considered as sequences of symbols, that is words over a finite alphabet, and the admissible steps of computation from them are considered as passing from one sequence of symbols to another, that is as word functions.

[15] R. Péter: *Automatische Programmierung zur Berechnung der partielle-rekursiven Funktionen,* Studia Sci. Math. Hung. **5** (1969) pp. 447–463.

4.6 Coding

It is not only true that the partial recursive functions are all machine computable, the converse also holds: – Whatever is machine computable, is also partial recursive. We shall consider this in detail using as an example the computation of $[\sqrt{n}]$ (see section 4.2).

The codes

$$ST, L, SP, A, S, M, G, G^{>}$$

for the types of statements which occur are not directly understood by the computer anyway. We could as well use the natural numbers

$$0, 1, 2, 3, 4, 5, 6, 7$$

for the codes of these statements. The codes of the addresses are already natural numbers, in our example from 1 to 15. Instead of W and E, as the codes of the statement counter and result register, we shall use 0 and 16, respectively. In what follows they will be dealt with just like the store addresses.

If for some positive $j \leqq 7$ the statement of type j refers to the address a, then

$$2^j \cdot 3^a$$

is the code number of the resulting statement. The only 0-address statement, type ST, is in itself a statement, and has already the code number 0.

It will not cause any confusion that several of the code numbers of statements also occur as address code numbers. They will always be used in distinct roles.

The program for the computation of $[\sqrt{n}]$, using all these code numbers, can be represented by the following primitive recursive numeric function $p(m, n)$, which determines the contents of each address m at the beginning of the computation:

$$p(m, n) = \begin{cases} 5, & \text{if } m = 0 \\ 1, & \text{if } m = 1 \\ n, & \text{if } m = 2 \\ 2^1 \cdot 3^3 = 54, & \text{if } m = 5 \\ 2^3 \cdot 3^1 = 24, & \text{if } m = 6 \\ 2^2 \cdot 3^4 = 324, & \text{if } m = 7 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \\ 0, & \text{if } m = 15 \\ 0 & \text{otherwise.} \end{cases}$$

The lines indicated by dotting could be read off the program, equally easily.

4.7 Recursion in Program Control

The program is executed step by step. At each step one statement is executed. This is the next statement if the previous one was not a go to statement. First we need a small modification. We also have a subtraction statement. This should be replaced by the arithmetic difference, which here yields the same result. We have already given a subroutine for the arithmetical difference $a \dot{-} n$.

Alongside the program, we have indicated the results of the single steps. Checking these we can see that, in executing any statement of type $j \leqq 7$ referring to the address a, only the contents of a, 0 (that is W) and 16 (that is E) might change. Moreover they only depend on j, a and the present contents (a), (0), (16), which are now considered as variables. Let us denote the resulting new contents of a, 0, and 16 by

$$g(j, a, (a), (0), (16)), \quad g_w(j, a, (a), (0), (16))$$

and

$$g_E(j, a, (a), (0), (16)).$$

These can be obtained by means of the following definitions by cases (as can easily be seen from the corresponding statements): –

$$g(j, a, (a), (0), (16)) = \begin{cases} (a), & \text{if } j = 0, 1, 3, 4, 5, 6, 7 \\ (16), & \text{if } j = 2 \\ 0 & \text{otherwise,} \end{cases}$$

$$g_w(j, a, (a), (0), (16)) = \begin{cases} (0), & \text{if } j = 0 \\ (0)+1, & \text{if } j = 1, 2, 3, 4, 5 \\ & \quad \text{or} \\ & \quad j = 7 \quad \text{and} \quad (16) \leqq 0 \\ a, & \text{if } j = 6 \\ & \quad \text{or} \\ & \quad j = 7 \quad \text{and} \quad (16) > 0 \\ 0 & \text{otherwise,} \end{cases}$$

$$g_E(j, a, (a), (0), (16)) = \begin{cases} (16), & \text{if } j = 0, 2, 6, 7 \\ (a), & \text{if } j = 1 \\ (16)+(a), & \text{if } j = 3 \\ (16) \dot{-} (a), & \text{if } j = 4 \\ (16) \times (a), & \text{if } j = 5 \\ 0 & \text{otherwise.} \end{cases}$$

Here all these functions are primitive recursive, and so is the following built up function

$$f(j, a, (a), (0), (16), (m), m) = \begin{cases} g(j, a, (a), (0), (16)), & \text{if } m = a \\ g_w(j, a, (a), (0), (16)), & \text{if } m = 0 \\ g_E(f, a, (a), (0), (16)), & \text{if } m = 16 \\ (m) & \text{otherwise,} \end{cases}$$

where (m) denotes the present contents of the address m. This gives, for each address m, the contents of m after the execution of the statement in question.

4.7.1 An Example

Let us denote by $\varphi(i, m, n)$ the contents of address m after the execution of the ith step. Step 0 is the input of the program into the computer. Hence

$$\varphi(0, m, n) = p(m, n).$$

Now assume that for some i the values $\varphi(i, m, n)$ are already given for each m. If the statement to be executed is of type j and refers to the address a, then according to the above

$$\varphi(i+1, m, n) = f(j, a, (a), (16), (m), m).$$

The address of the statement to be executed is, however, the present contents of the statement counter, which is

$$\varphi(i, 0, n).$$

The contents of this address is

$$\varphi(i, \varphi(i, 0, n), n),$$

and it has to be a statement, i.e. its prime factor representation is of the form

$$2^j \cdot 3^a.$$

We obtain j and a from here as the exponents of the 0th and 1st prime numbers, respectively: –

$$j = \exp_0(\varphi(i, \varphi(i, 0, n), n)), \quad a = \exp_1(\varphi(i, \varphi(i, 0, n), n)),$$

showing that the present contents of the address a is

$$(a) = \varphi(i, \exp_1(\varphi(i, \varphi(i, 0, n), n)), n).$$

In addition to the expression

$$(0) = \varphi(i, 0, n)$$

which we have already used, we also have

$$(16) = \varphi(i, 16, n),$$

and

$$(m) = \varphi(i, m, n).$$

Putting all this into the expression obtained earlier for $\varphi(i+1, m, n)$, and using the known expression for $\varphi(0, m, n)$, we obtain a definition of φ by the following nested recursion: –

$$\begin{cases} \varphi(0, m, n) = p(m, n) \\ \varphi(i+1, m, n) = f(\exp_0(\varphi(i, \varphi(i, 0, n), n)), \exp_1(\varphi(i, \varphi(i, 0, n), n)). \\ \varphi(i, \exp_1(\varphi(i, \varphi(i, 0, n), n)), \varphi(i, 0, n), \varphi(i, 16, n). \\ \varphi(i, m, n), m). \end{cases}$$

Such a recursion, however, as was noted in section 2.7.1, still remains within the class of primitive recursive functions.

Therefore the function $\varphi(i, m, n)$ representing the execution of the program is primitive recursive. This is the consequence of the fact that the occuring cycle is not a circle.

4.7.2 Computable Functions

But what about the result? We saw that after the ith step the statement to be executed has the form

$$j = \exp_0(\varphi(i, \varphi(i, 0, n), n)).$$

If this is 0, which codes the stop statement, then the result is the contents of the result register, that is $\varphi(i, 16, n)$. Hence, to obtain the result, we have to search for the smallest i with the indicated property, that is for

$$\mu_i[\exp_0(\varphi(i, \varphi(i, 0, n), n)) = 0],$$

and then substitute this for i in $\varphi(i, 16, n)$.

In our example, it is easy to give an upper bound for this i. The program has a cycle of 8 terms, moreover 3 statements. The cycle is repeated as many times as there are positive numbers, the squares of which do not exceed n. Clearly, there are at most n such numbers. Therefore the number of steps in executing the program is at most $8n+3$.

Consequently, the result of the computation with our program is

$$\varphi(\mu_i[i \leqslant 8n+3 \;\&\; \exp_0(\varphi(i, \varphi(i, 0, n), n)) = 0], 16, n),$$

which is a primitive recursive function.

4.8 Partial Recursion in Binary Computer Arithmetic

We have known, of course, already that this program computes the primitive recursive function $[\sqrt{n}]$. But our reasoning can be generalized to apply to a computer with an arbitrary system of statements (for which the result appears not necessarily as (E), but can also be a sequence). In my paper quoted in footnote [14] I have shown in this way that the result of an arbitrary – suitably coded and stored – program with the input parameters

$$n_1, n_2, \ldots, n_r$$

can be obtained from primitive recursive functions by means of a single μ_i-operation. If a primitive recursive upper bound can be found for i, then the result is a primitive recursive function. If this is not the case but it can be proved that for every choice of the input parameters, there is such an i, then the result of the program is still general recursive. It is, however, always partial recursive.
Consequently, we can indeed say that *whatever is machine computable, is also partial recursive.*

Thus if we study the programming problems of the computation of partial recursive functions, this means, in principle, the study of programming of all the machine solvable problems.
The coding by natural numbers is, however, something extraneous to the computer. It understands only whatever can be coded by finite sequences of the symbols 0 and 1. Its mother tongue is the binary language, that is the word set M with an alphabet consisting solely of 0 and 1. The whole of the above reasoning can, however, be carried out in this word set. Instead of the code numbers of the addresses we can consider their binary forms as code words in M, and instead of the arithmetical operations between numbers we have the digital operations between their binary forms, which by section 3.6 are primitive recursive functions in M. The codes of statement types can also be expressed as binary forms of numbers. Here, of course, to code a statement, whose type is coded by j, and which is referred to by the address coded as a, we cannot make use of the unique prime factor representation of natural numbers. Instead, the same end is served in M by the primitive recursive function $c(j, a)$ introduced in section 3.8. This makes a word w correspond to the two-term sequence j, a, from which the respective terms can be recovered by means of the functions

$$k_0(w), \quad k_1(w),$$

which are primitive recursive in M. Let us consider e.g. the statement $SP4$, which was earlier coded by the number

$$2^2 \cdot 3^4.$$

As the binary forms 10 of 2 and 100 of 4 contain only one occurrence of 1, the appropriate "separating symbol" here is 0110. Hence the code word of the two-term sequence, representing the statement, is

$$c(10, 100) = 1001101000110.$$

The sequence of symbols 0110 at the end of this word shows that (after having checked that the word has no connected part of the form 111 or 0110110) this sequence plays the role of a separating symbol. Hence the terms of the sequence can be uniquely recovered from this word as

$$k_0(1001101000110) = 10, \quad k_1(1001101000110) = 100.$$

In this manner we have arrived at the following result: *everything obtainable by a computer is partial recursive in the binary language of the computers.*

Chapter 5

Sequential Program Translation

5.1 The Bracketless Form

Programs are not formulated in the language of the computer. They must first be translated into that language. This can happen in several stages.

Let us consider, as a simple example the statement requiring the computation of the expression

$$(b+b\times c)\times a+c,$$

which is composed of several arithmetical operations. Let the first stage of the translation be the transformation of this expression into a bracketless form.

Several such forms are known. The first of these is due to Lukasiewicz [16]. Here we shall use the so called "reversed-Polish" form

$$bbc\times+a\times c+,$$

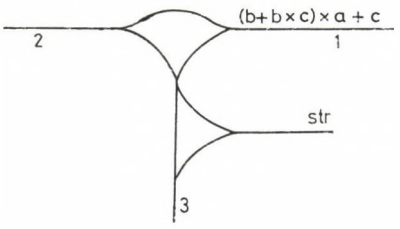
in which the operation symbol is placed after the two operands. Its meaning can be read off by checking the symbols one by one, going from right to left, as follows: –

"A sum, whose second term is c and first term is a product, whose second factor is a and the first factor is a sum, for which the second term is a product with c as second and b as first factor, and b is the first term." Accordingly, the first operation symbol from the left is the innermost.

[16] Concerning the uniqueness of this form see for example, L. Kalmár: *Another proof of the Markov–Post theorem,* Acta Math. Acad. Sci. Hung. **3** (1952) pp. 1–27. The uniqueness of the "reversed form" can be proved in a similar way.

The translation to this form can be carried out by an algorithm due to E. W. Dijkstra [17], which he has illustrated by "railway marshalling" statements.

For this, the different symbols occurring in the given expression (variable symbols, operation symbols, and left and right parentheses) will represent railway cars of different types. Here however every car has an engine. Appropriate cars form our expression into a train, and the task is to send this train, with its cars re-arranged in the reversed-Polish form, from track 1 to track 2. In the course of this, track 3 with a sidetrack (denoted as str) is at our disposal. See the diagram below.



In our expression $b \times c$ is not put in parentheses, since, by convention, multiplication takes precedence over addition. We shall express this by saying that each operation has associated with it a priority, namely addition has priority 1, multiplication has priority 2, the priority of subtraction is also 1, while exponentiation has priority 3.

Now the marshalling instructions are as follows: Separate the cars. The opening parenthesis has to go to track 3. The next variable always has to go to track 2. The next operation symbol goes to track 3 temporarily, but unless it meets there an operation symbol of lower priority, it has to give way; that is it has to pull onto the side track, while this other operation symbol of higher or equal priority goes to track 2. Only afterwards can it go back to track 3. If the next symbol to leave track 1 is a closing parenthesis, put it on the side track. Then let the operation symbols from track 3 go to track 2 one by one, until we reach an opening parenthesis. Add this to the closing parenthesis waiting on the side track and discard this pair of used up parentheses, in other words, send them to the depot. Finally, if nothing is left on track 1, let the symbols still waiting on track 3 go one by one to track 2.

[17] E. W. Dijkstra: *Making a translator for Algol 60,* A. P. I. C. Bull. 7 (1961) pp. 3–11. See also B. Randell and L. J. Russel: *Algol 60 implementation* (1964) London, New York.

In the following table, the successive execution of the marshalling instructions is indicated in detail in the case of our expression: –

track 1	track 2	track 3	str
$(b+b\times c)\times a+c$	–	–	–
$b+b\times c)\times a+c$	–	(	–
$+b\times c)\times a+c$	b	(	–
$b\times c)\times a+c$	b	$(+$	–
$\times c)\times a+c$	bb	$(+$	–
$c)\times a+c$	bb	$(+\times$	–
$)\times a+c$	bbc	$(+\times$	–
$\times a+c$	bbc	$(+\times$	)
$\times a+c$	$bbc\times$	$(+$	)
$\times a+c$	$bbc\times+$	(	)
$\times a+c$	$bbc\times+$	–	–
$a+c$	$bbc\times+$	$\times$	–
$+c$	$bbc\times+a$	$\times$	–
c	$bbc\times+a$	$\times$	+
c	$bbc\times+a\times$	+	–
–	$bbc\times+a\times c$	+	–
–	$bbc\times+a\times c+$	–	–

At the end the reversed-Polish form of our expression has appeared on track 2.

5.1.1 The Three-address Code

Looking at the meaning of this form it is easy to deduce from it an algorithm for its decomposition into three-address computer statements of the form θuvw, where θ is an operation symbol. The statement requires the execution of the corresponding operation for the contents of u and v and placing the result in w. Proceeding from the left to the right, one always has to look for the first operation-symbol "car" (corresponding to the "innermost" operation), attach the two immediately preceding cars and a car of a new type (brought from the depot) after it, and then let the train put together in this way go to a new track. A second copy of the above car of new type is also to be brought from the depot, in order to fill up the resulting gap in the old train. This contains the result of the operation just executed, which, from here on, is treated as an operand. Then everything starts again from the beginning. One has to look for the first operation symbol from the left in the modified train, and so on.

By means of suitable switching devices and marshalling instructions, the separate small trains (each consisting of 4 cars) can be collected on a track sequentially (i.e. the cars will occur in their original order), as happened in putting the train on track 2. I will not go into the details of this here. However, the sequentiality of the procedure is certainly disturbed by the fact that, in the train on track 2, one has to look for the first operation symbol, and then eventually one has to return to earlier symbols. This can be avoided if the new procedure is carried out simultaneously with the old one. We start the collection of the train on track 2 according to the original algorithm, until an operation symbol appears on track 2. With this we proceed as has just been described. Only afterwards is the original algorithm continued, until the next operation symbol appears. This is illustrated in the following table, in which the symbol ":" indicates that the small trains on track 4 are not attached to each other.

track 1	track 2	track 3	str	track 4
$(b+b\times c)\times a+c$	–	–	–	–
$b+b\times c)\times a+c$	–	(	–	–
$+b\times c)\times a+c$	b	(	–	–
$b\times c)\times a+c$	b	(+	–	–
$\times c)\times a+c$	bb	(+	–	–
$c)\times a+c$	bb	(+×	–	–
$)\times a+c$	bbc	(+×	–	–
$\times a+c$	bbc	(+×	)	–
$\times a+c$	$bbc\times$	(+	)	–
$\times a+c$	bv_1	(+	)	$\times bcv_1$:
$\times a+c$	bv_1+	(	)	$\times bcv_1$:
$\times a+c$	v_2	(	)	$\times bcv_1$:$+bv_1v_2$:
$\times a+c$	v_2	–	–	$\times bcv_1$:$+bv_1v_2$:
$a+c$	v_2	×	–	$\times bcv_1$:$+bv_1v_2$:
$+c$	v_2a	×	–	$\times bcv_1$:$+bv_1v_2$:
c	v_2a	×	+	$\times bcv_1$:$+bv_1v_2$:
c	$v_2a\times$	+	–	$\times bcv_1$:$+bv_1v_2$:
c	v_3	+	–	$\times bcv_1$:$+bv_1v_2$:$\times v_2av_3$:
–	v_3c	+	–	$\times bcv_1$:$+bv_1v_2$:$\times v_2av_3$:
–	v_3c+	–	–	$\times bcv_1$:$+bv_1v_2$:$\times v_2av_3$:
–	v_4	–	–	$\times bcv_1$:$+bv_1v_2$:$\times v_2a_3$:$+v_3cv_4$

Finally, only v_4 remains on track 2, which, however, contains the value of our expression, for according to the above, the respective statements on track 4 have the following meanings: –

$$b\times c \Rightarrow v_1;\quad b+b\times c \Rightarrow v_2;\quad (b+b\times c)\times a \Rightarrow v_3;\quad (b+b\times c)\times a+c \Rightarrow v_4.$$

5.1.2 Reduction to One-address Code

A three-address statement, of course, can be decomposed very simply into three one-address statements of the sort we introduced in section 4.1. For example

$$\times bcv_1 \quad \text{into} \quad Lb;\ Mc;\ SPv_1$$

and

$$+v_3cv_4 \quad \text{into} \quad Lv_3;\ Ac;\ SPv_4.$$

5.1.3 Translation into Word Functions

In translating programming languages, certain sequences of symbols are replaced by others. The sequences of symbols can be considered as words over an alphabet containing all the necessary symbols. Hence here we are dealing with word functions. Using a suitable notation, we can always restrict ourselves to a finite alphabet; e.g. from the two symbols x and | one can build the following infinite sequence of variables:

$$x|,\ x||,\ x|||,\ \ldots\ .$$

For the sake of clarity, however, I shall adopt the more usual notation with lower indices.

5.2 Push-down Stores

In determining (that is computing) the successive symbols of a function value, one examines the separate symbols of the arguments. It is convenient to move up and down among these symbols. Hence it is always reasonable to look for a sequential computation procedure, in which the symbols of the arguments are taken into consideration successively, in their correct order. The application of push-down stores will help us to achieve this goal. In the above examples the role of the push-down stores was played by the tracks.

A push-down store is a symbolic store, in which letters of the alphabet and perhaps also some auxiliary letters can be placed. These can be taken out of the store, either in such a way that the corresponding letter is erased there, or the letter which is taken out also stays in the store, that is only a copy of it is taken out. In the railway analogy, this corresponds to bringing a car of the same type from the depot. Such a store satisfies the following conditions: – Whenever a letter is placed in the store, it pushes down all the letters already there one place deeper (that is the train backs up). Whenever a letter is taken out, this must be the top letter, that is the most recently

added letter is removed first. If a letter is taken out with erasure, then the other letters in the store automatically pop up one place (that is the train pulls forward). Whatever has to be done at a given step of the computation of the function value depends on the current symbol at the top of the push-down store, which we call the top symbol.

The push-down stores make it possible to move up and down among the letters of the arguments, without disturbing the sequential character of the computation. Indeed, all the letters of the arguments can be poured, one by one, into a push-down store, where they are kept until the end. Then if one of these letters is needed, the letters placed on top of it can be poured into another push-down store, and after the work is done they can be poured back again. The same can be done with the intermediate results of the computation. In a recursive procedure, this must be done many times. I have given [18] a general method for the sequential computation of every recursive partial word function over a finite alphabet, in which the number of push-down stores is independent of the arguments. J. Urbán [19] has later shown that the use of three pushdown stores always suffices. Conversely, I have also shown that every word function sequentially computable with the help of push-down stores is partial recursive in the word set, extended with some auxiliary letters.

Comparing this with the final conclusion of Ch. 4, we can say the following: *Everything which can be calculated by a computer, can be calculated sequentially, with the use of three push-down stores.*

5.2.1 Some Conventions

In the proofs mentioned above, the following notation was used: –

We had a word set M over a finite alphabet A, the letters of which I will denote here by a perhaps with some indices

For the sake of brevity, we assume that on the bottom of every push-down store is the symbol L (denoting empty); hence the top symbol of an empty push-down store is L. Further, if a symbol is erased, then we say that it is replaced by l, and the top symbol of an arbitrary push-down store is the first symbol, counted from the top, which is different from l. Of course, neither L nor l can occur among the letters of A. These auxiliary symbols

[18] R. Péter: *Über die sequenzielle Berechnenbankeit von rekursiven Wortfunktionen durch Kellerspeicher,* Acta Math. Acad. Sci. Hung. **16** (1965) pp. 231–253.

[19] J. Urbán: *Die Minimalisierung der zur sequenziellen Berechnung der partiell-rekursiven Wortfunktionen notwendigen Kellerspeicher,* Acta Math. Acad. Sci. Hung. **17** (1966) pp. 335–358.

are distinct from the symbol introduced earlier for the empty word. This symbol has to be written out and dealt with in the same way as the other symbols of the alphabet, although in the result it is to be considered as something non-existent. The reason for this is that, in the course of the computation, an intermediate value might turn out to be Λ, with which one has to deal just like with any other intermediate value.

Instead of listing from the bottom to the top, the symbols obtained in a push-down store will be listed from the left to the right, as it happened in the railway analogy.

The push-down stores will be denoted by capital letters, among them one called I for input and another called O for output. Their current top symbols will be denoted by the corresponding small letters.

The computation of a word function $f(x_1, \ldots, x_n)$ for the arguments

$$a_{1,1} \ldots a_{1,r_1}, \ldots, a_{n,1} \ldots a_{n,r_n}$$

always starts with pouring the symbols of this chain (which I will denote by s), one by one, proceeding from the right to the left, into I (including the comma, which is not a letter). Initially therefore the contents of I (after L) is the chain of symbols

$$a_{n,r_n} \ldots a_{n,1}, \ldots, a_{1,r_1} \ldots a_{1,1}$$

(denoted by $\bar{s}$) and the contents of every other store is L. At the end of the computation we find in O after L the required function value as a chain of letters and Λ-symbols. This chain will be denoted symbolically as $f(s)$, while in all the other stores we find L.

The computation procedure is in separate stages, which will be denoted by q (with indices). It starts with q_1 (after placing $\bar{s}$ in I), and ends with a stage calling for stop.

What has to be done at a given moment depends only on the current stage and the current top symbols of the stores. Depending on these the new top symbols are obtained and a new stage follows. These are not necessarily different from the earlier ones. In the short description of a computational step, only the top symbols which are affected, will be indicated. For example, suppose that in a stage q_t the top symbol of I is to be removed and placed in the store K unless the top symbol of K is L, and then the stage q_u is to follow. This will be denoted by

$$q_t(k \neq L)|(i \to K)\, q_u,$$

while the same operation when i is kept in I, will be denoted by

$$q_t(k \neq L)|(i \to \mathrm{I})(i \to K)\, q_u.$$

If i has to be erased only, and not placed anywhere, we denote this by

$$l \to i.$$

5.2.2 Computation of Initial Functions

We consider below as an example the computation of the initial functions of M.

To compute

$$f(x_1, \ldots, x_n) = \wedge$$

one has to empty I and place $\wedge$ in O;

$$q_1(i \neq L)\,|\,(l \rightarrow i)\,q_1$$
$$q_1(i = L)\,|\,(\wedge \rightarrow \mathrm{O})\,q_2$$
$$q_2\,|\,\mathrm{Stop}.$$

To compute

$$xa \quad (a \in A)$$

one has to pour $\bar{s}$ out of I into O, by which the original order of the letters, that is s, is recovered. Then a has to be added to the end: –

$$q_1(i \neq L)\,|\,(i \rightarrow \mathrm{O})\,q_1$$
$$q_1(i = L)\,|\,(a \rightarrow \mathrm{O})\,q_2$$
$$q_2\,|\,\mathrm{Stop}.$$

To compute

$$f(x_1, \ldots, x_n) = x_j (1 \leqq j \leqq n)$$

one has to remove from I everything that comes after x_j. Then the letters of the argument x_j are to be poured into O, where they regain their original order. Finally everything that still remained in I must be erased. To begin with, for $j > 1$, we have the stages

$$q_t \quad (t = 1, 2, \ldots, j-1)$$

with the effect

$$q_t(i \neq ,)\,|\,(l \rightarrow i)\,q_t$$
$$q_t(i = ,)\,|\,(l \rightarrow i)\,q_{t+1}.$$

Then follows

$$q_j(i \neq ,)(i \neq L)\,|\,(i \rightarrow \mathrm{O})\,q_j$$
$$q_j(i = ,)\,|\,(l \rightarrow i)\,q_{j+1}$$
$$q_j(i = L)\,|\,q_{j+2}$$
$$q_{j+1}(i \neq L)\,|\,(l \rightarrow i)\,q_{j+1}$$
$$q_{j+1},(i = L)\,|\,q_{j+2}$$
$$q_{j+2}\,|\,\mathrm{Stop!}$$

For the computation of

$$\mathrm{lb}\,(x),$$

the last letter of x, it is actually counter-productive to pour the letters of the argument into I, since in this way the last letter gets to the bottom. However, this is done for the sake of uniformity. One has to remove the top symbol of I and put it in O. If I is now empty, this was the last letter of the argument. If not, this letter has to be erased from O and the procedure has to be repeated with the new top symbol of I, thus: –

$$q_1|(i \to \mathrm{O})q_2$$

$$q_2(i \neq L)|(l \to o)q_1$$

$$q_2(i = L)|q_3$$

$$q_3|\mathrm{Stop}.$$

To compute

$$\mathrm{eq}\,(x, y) = \begin{cases} \wedge, & \text{if} \quad x = y \\ a_0 & \text{otherwise,} \end{cases}$$

where a_0 is a fixed element of A, the two arguments x, y are to be poured into two different push-down stores. O can be one of these temporarily. The other is denoted by $\bar{\mathrm{I}}$. Then we can compare and erase their letters one by one and if they are all found identical, we put a_0 into O, which has been emptied by that time. Otherwise we put $\wedge$ into O, thus: –

$$q_1(i \neq ,)|(i \to \bar{\mathrm{I}})q_1$$

$$q_1(i = ,)|(l \to i)q_2$$

$$q_2(i \neq L)|(i \to \mathrm{O})q_2$$

$$q_2(i = L)|q_3$$

$$q_3(\bar{i} \neq L)(\bar{i} = o)|(l \to \bar{i})(l \to o)q_3$$

$$q_3(\bar{i} = L)(o = L)|(\wedge \to \mathrm{O})q_6$$

$$q_3(\bar{i} = L)(o \neq L)|q_4$$

$$q_4(o \neq L)|(l \to o)q_4$$

$$q_4(o = L)|(a_0 \to \mathrm{O})q_6$$

$$q_3(\bar{i} \neq L)(\bar{i} \neq o)|q_5$$

$$q_5(\bar{i} \neq L)|(l \to \bar{i})q_5$$

$$q_5(\bar{i} = L)|q_4$$

$$q_6|\mathrm{Stop}.$$

5.2.3 Computing Partial Recursive Functions

As a further example, assume that the word functions

$$f(x, y), \quad g(x), \quad h(x)$$

are sequentially computable by means of three push-down store systems, which start with the input stores

$$\mathrm{I}_f, \mathrm{I}_g, \mathrm{I}_h,$$

and end with the output stores

$$\mathrm{O}_f, \mathrm{O}_g, \mathrm{O}_h,$$

respectively. Using these, we are able to produce a push-down store system for the computation of

$$f(g(x), h(x)).$$

Here it is possible to identify O with O_f.
Initially I contains $\bar{s}$, and by simply pouring this into another store we get s, whereas our task is to place $\bar{s}$ in both I_g and I_h. Therefore we have to take an additional push-down store $\bar{\mathrm{I}}$. The first step is then to pour the contents of I into $\bar{\mathrm{I}}$, and then to pour them out of $\bar{\mathrm{I}}$ into both I_g and I_h. In this way the order of the letters will be that required. We can then follow the steps of the (already known) computation of the value of g, which together will be denoted by the symbol Q_g. This ends with $g(s)$ in O_g (after L), and with L in all the other stores except I_h. The steps of the computation of the value of h (denoted by Q_h) should now follow, where at the end $h(s)$ appears in O_h (after L). Finally, the contents of O_h and then of O_g have to be poured into I_f, separated by a comma. After this can follow the steps of the computation of f for these arguments, denoted together by Q_f. The stage symbols q_i belonging to

$$Q_g, Q_h, Q_f$$

respectively, should be distinguished from each other and from the stage symbols with no index, by corresponding indices g, h, f. We now have to add the initial stage of Q_h to this at the end of Q_g, instead of "Stop", and to put the stage q_4 at the end of Q_h instead of "Stop". This procedure can be described as follows:

$$q_1(i \neq L) \mid (i \to \bar{\mathrm{I}})\, q_1$$

$$q_1(i = L) \mid q_2$$

$$q_2(\bar{i} \neq L) \mid (\bar{i} \to \bar{\mathrm{I}})(\bar{i} \to I_g)\, q_3$$

$$q_3 \mid (\bar{i} \to \mathrm{I}_h)\, q_2$$

$$q_2(\bar{i} = L) \mid Q_g$$

$$Q_h$$
$$q_4(o_h \neq L)\,|\,(o_h \to \mathrm{I}_f)\,q_4$$
$$q_4(o_h = L)\,|\,(, \to \mathrm{I}_f)\,q_5$$
$$q_5(o_g \neq L)\,|\,(o_g \to \mathrm{I}_f)\,q_5$$
$$q_5(o_g = L)\,|\,Q_f.$$

It was shown in a similar way in the paper quoted in footnote [18] that computability by means of push-down stores is preserved under arbitrary substitutions, and (which is somewhat more tedious) under primitive recursions, and also under μ-operations. This yields the methods of sequential computation of all the recursive functions by means of push-down stores.

5.2.4 Restriction to Three Push-down Stores

The result of the paper quoted in footnote [19], namely that three push-down stores always suffice, is achieved by a suitable blocking of the separate stores. For this, L is used in a new role, and a new auxiliary symbol $\odot$ is also needed, which makes it possible to store several different sequences of symbols in the same store.

We shall illustrate this with the example of the previous section, where of course the additional assumption is made that the functions

$$f(x, y), \quad g(x), \quad h(x)$$

can be computed with the help of 3 push-down stores. For the initial functions this is true. To compute the function

$$f(g(x), h(x))$$

we have then the following procedure, using only the three stores I, $\bar{\mathrm{I}}$, O: – The values $g(s)$ and $h(s)$, in this order, separated by a symbol L, will be placed in O. In order that the place where this begins can be found again, first the symbol $\odot$ is put in O, and then on top of it another L: –

$$q_1\,|\,(\odot \to \mathrm{O})\,q_2$$
$$q_2\,|\,(L \to \mathrm{O})\,q_3.$$

We could now carry out the computation Q_g of $g(s)$ using the three stores I, $\bar{\mathrm{I}}$, O. However s must also be preserved for the computation of $h(s)$. Therefore, we first put $\bar{s}$ simultaneously into $\bar{\mathrm{I}}$ and O, then from O we pour

it back into I, while in $\bar{\mathrm{I}}$ we block it by an L. Then we can start Q_g: –

$$q_3(i \neq L)|(i \to \mathrm{I})(i \to \bar{\mathrm{I}})\, q_4$$
$$q_4|(i \to \mathrm{O})\, q_3$$
$$q_3(i = L)|\, q_5$$
$$q_5(o \neq L)|(o \to \mathrm{I})\, q_5$$
$$q_5(o = L)|(L \to \bar{\mathrm{I}})Q_g.$$

Q_g ends with $g(s)$ on top of the highest L in O, and with L as the top symbol, in the other stores. The L from $\bar{\mathrm{I}}$ has to be taken out and used to block O. Thus s will be opened up again in $\bar{\mathrm{I}}$. So we can pour it into I again. Now our stores are ready for the computation Q_h of $h(s)$: –

$$q_6|(\bar{\imath} \to \mathrm{O})\, q_7$$
$$q_7(\bar{\imath} \neq L)|(\bar{\imath} \to \mathrm{I})\, q_7$$
$$q_7(\bar{\imath} = L)|Q_h.$$

After this the contents of O are

$$L \odot Lg(s)Lh(s),$$

and what we have to do is to compute f for the chain of arguments

$$g(s), \quad h(s).$$

This can be poured (in the required reversed order) from O into I, meanwhile erasing and replacing by a comma the first L from the right. We still have to erase the auxiliary symbols $\odot L$ from O, so that only the original L will remain. After this the computation Q_f of f for the desired arguments can be executed:

$$q_8\ (o \neq L)|(o \to \mathrm{I})\, q_8$$
$$q_8\ (o = L)|(l \to o)\, q_9$$
$$q_9\ |(, \to \mathrm{I})\, q_{10}$$
$$q_{10}(o \neq L)|(o \to \mathrm{I})\, q_{10}$$
$$q_{10}(o = L)|(l \to o)\, q_{11}$$
$$q_{11}|(l \to o)Q_f.$$

Of course, in this, Stop at the end of Q_g has to be replaced by q_6, and Stop at the end of Q_h by q_8.

It happened here that, at the beginning of a computation of a function value the contents of the stores $\bar{\mathrm{I}}$ or O were not just L. In more complicated cases, it can also happen that I also contains a chain of symbols below $L\bar{s}$. The

chain of symbols up to (and including) the highest L which can be found in a store at the beginning of a computation is called a kernel chain. This remains unchanged during the computation, at the end of which in I and $\bar{\text{I}}$ we will find their kernel chains, while in O is left its kernel chain with the result of the computation on top of it. This result is completely independent of the kernel chains.

In a similar way one can show that sequential computability with 3 push-down stores is also preserved for arbitrary substitutions, primitive recursions, and μ-operations.

5.3 Partial Recursivity in Push-down Stores

The converse of this is also true. Whatever can be computed by means of push-down stores in a word set M over a finite alphabet A, is partial recursive in the word set M' over the alphabet A', which, in addition to the letters of A, also contains the auxiliary symbols

$$,;\ \wedge;\ L;\ l$$

and in the case of the method applied for the minimization of the number of push-down stores, also $\odot$. The proof of this will be illustrated using as an example the computation of $\mathrm{lb}(x)$, executed in section 5.2.2. The successive instructions of the computation were as follows: –

$$q_1 \mid (i \to \mathrm{O}) q_2$$
$$q_2 (i \neq L) \mid (l \to \mathrm{O})\, q_1$$
$$q_2 (i = L) \mid q_3$$
$$q_3 \mid \mathrm{Stop}.$$

In every computation there are only a finite number of instructions (here 4), in which a finite number of stages occur (here 3). In the 0th moment, that is before the computations starts, the contents of the stores I, $\bar{\text{I}}$, O are the kernel chains

$$w_1 L,\ w_2 L,\ w_3 L,$$

respectively. Moreover, in I, on top of this, is the reversed argument chain which in our example is the single argument x. The reverse $\bar{x}$ of a word x is primitive recursive in every word set, since it can be defined by the following primitive recursion: –

$$\bar{x} = \begin{cases} \wedge, & \text{if} \quad x = \wedge \\ \mathrm{lb}(x)\, \overline{\mathrm{at}(x)} & \text{otherwise.} \end{cases}$$

Let us denote the contents of I, $\bar{\text{I}}$, O and the index of the current stage in the $o(z)$th moment of the computation by

$$\iota\ (w_1, w_2, w_3, x, z)$$
$$\bar{\iota}\ (w_1, w_2, w_3, x, z)$$
$$\omega(w_1, w_2, w_3, x, z)$$
$$\sigma\ (w_1, w_2, w_3, x, z)$$

respectively. As the parameters

$$w_1, w_2, w_3, x$$

do not change in the course of the computation, I shall denote them by

$$\iota(z), \bar{\iota}(z), \omega(z), \sigma(z),$$

respectively.

In our example, in which $\bar{\text{I}}$ does not appear, these functions can be defined as follows (where a top symbol means the last letter of a word, the erasure of which yields the initial part of the word): –

$$\iota(z) = \begin{cases} w_1 L\bar{x}, & \text{if} \quad z = \Lambda \\ \text{at}\,(i(\text{at}\,(z))), & \text{if} \quad z \neq \Lambda \,\&\, \sigma(\text{at}\,(z)) = 1 \\ \iota(\text{at}\,(z)) & \text{otherwise,} \end{cases}$$

$$\omega(z) = \begin{cases} w_3 L, & \text{if} \quad z = \Lambda \\ \omega\,(\text{at}\,(z))\,\text{lb}\,(i(\text{at}\,(z))), & \text{if} \quad z \neq \Lambda \,\&\, \sigma\,(\text{at}\,(z)) = 1 \\ \text{at}\,(\omega(\text{at}(z))), & \text{if} \quad z \neq \Lambda \,\&\, \sigma(\text{at}\,(z)) = 2 \,\&\, \text{lb}\,(\iota\,(\text{at}\,(z))) \neq L \\ \omega\,(\text{at}\,(z)) & \text{otherwise,} \end{cases}$$

$$\sigma(z) = \begin{cases} 1, & \text{if} \quad z = \Lambda \\ 1, & \text{if} \quad z \neq \Lambda \,\&\, \sigma(\text{at}\,(z)) = 2 \,\&\, \text{lb}\,(\iota(\text{at}\,(z))) \neq L \\ 2, & \text{if} \quad z \neq \Lambda \,\&\, \sigma(\text{at}\,(z)) = 1 \\ 3 & \text{otherwise.} \end{cases}$$

Here 1, 2, ... denote those words which represent the corresponding natural numbers in M'.

It should be noted that on the right-hand side of these definitions, unless $z = \Lambda$, z appears only as an argument of at, ι, ω and σ. Therefore, as was explained in section 3.8, these simultaneous recursions can be reduced to primitive recursions of the following form: –

$$\iota(z) = \begin{cases} w_1 L\bar{x}, & \text{if} \quad z = \Lambda \\ h_1(\iota(\text{at}\,(z))) & \text{otherwise,} \end{cases}$$

$$\omega(z) = \begin{cases} w_3 L, & \text{if} \quad z = \Lambda \\ h_2(\omega(\text{at}\,(z))) & \text{otherwise,} \end{cases}$$

$$\sigma(z) = \begin{cases} 1, & \text{if} \quad z = \Lambda \\ h_3(\sigma(\text{at}\,(z))) & \text{otherwise.} \end{cases}$$

These definitions, however, are pure iterations. From the first, for example, we obtain

$$\iota(z) = \begin{cases} w_1 L\bar{x}, & \text{if } z = \wedge, \\ h_1(w_1 L\bar{x}), & \text{if } o(z) = 1, \\ h_1(h_1(w_1 L\bar{x})), & \text{if } o(z) = 2, \\ h_1(h_1(h_1(w_1 L\bar{x}))), & \text{if } o(z) = 3. \end{cases}$$

Hence $\iota(z)$ is the $o(z)$th iteration of h_1 at the place $w_1 L\bar{x}$, which will be denoted by

$$\iota(z) = h_1^{(o(z))}(w_1 L\bar{x}).$$

Similarly we obtain

$$\omega(z) = h_2^{(o(z))}(w_3 L),$$

$$\sigma(z) = h_3^{(o(z))}(1).$$

Clearly, these functions, which do not depend on z but only on $o(z)$ (and parameters w_1, w_3, x which are not shown), are primitive recursive sequences.

The result of the computation is obtained in the first such moment m, in which the stop stage

$$\sigma(m) = 3$$

is reached. Hence, by the definition of the μ-operation from section 3.8,

$$m = \mu_{o(z)}[\sigma(o(z)) = 3].$$

The result, then, is the word obtained from $\omega(m)$ by erasing the kernel chain $w_3 L$ from its beginning, or in other words, the final segment (see section 3.6.2) of order

$$o(\omega(m)) \dot{-} o(w_3 L)$$

which is independent of the kernel chain. Hence the value of our function lb for the argument x is

$$\mathrm{lb}\,(x) = e(\omega(x, m),\ o(\omega(x, m)) \dot{-} o(w_3 L))$$

with the above m.

It is easy to find an upper bound for this m on the basis of the computing instructions above.

One has to move the characters of $\bar{x}$ (if $x = \wedge$ their number is 1, otherwise it is $o(x)$) one by one from I into O. Here all but the last one of them have to be erased, and then comes the transition to the stop stage. For $x = \wedge$ this means 2 steps, otherwise $2 \cdot o(x)$ steps. Hence in any case there are at most $2 \cdot o(x) + 2$ steps.

Consequently, m is definable by means of the bounded μ-operation

$$m = \mu_z[z \leqq 2 \cdot o(x) + 2 \ \&\ \sigma(z) = 3]$$

z, as a predecessor of a natural number, must itself be a natural number. Hence $z=o(z)$. Thus m is primitive recursive and consequently the result of our computation is primitive recursive.

Of course, we already knew this for $\mathrm{lb}(x)$, but our reasoning can be generalized to show that the result is always obtainable from primitive recursive word functions by means of a single μ-operation (to which, in general, no primitive recursive bound is available). Hence every word function computable by means of push-down stores is (after a suitable extension of the alphabet by auxiliary symbols), partial recursive.

5.4 Illustration on Railway Marshalling

Now these simple computation steps have a very natural translation into a "railway marshalling language", in which every letter or auxiliary symbol has associated with it a car type (denoted by it), the push-down stores correspond to tracks, and the stages to marshalling yards.

In this way, the word functions can be defined by railway marshalling graphs and traffic regulations corresponding to these [20]. A railway marshalling graph means a finite, connected, directed graph containing only triple edges. These correspond to the tracks. The edges connecting the same vertices (which will be called parallel) are directed in the same way, and will be denoted by I, $\bar{\mathrm{I}}$ and O. The vertices correspond to yards with suitable switching devices. So now we are able to execute the following simple instructions concerning the last cars of the trains standing simultaneously on the three parallel tracks:

1) Disconnect the last car from the train standing on one of the tracks and send it to the depot. (This corresponds to the erasure of a given top symbol.)

2) Bring a car of a given type from the depot and join it to the last car of a train standing on one of the tracks. (This corresponds to putting a certain symbol into a given push-down store.)

3) Disconnect the last car of a train on one of the tracks and join it to the end of another. (This corresponds to moving a top symbol from one store another.)

4) Bring out of the depot a car of the same type as the last car of a fixed train and join this to the end of one of the trains. (This corresponds to

[20] R. Péter: *Veranschaulichung eines sequenziellen Berechnung der rekursiven Funktionen durch „eisenbahnrangierende Graphen"*, Periodica Math. Hung. **3** (1973) pp. 183–187.

the transfer of a certain to symbol into another fixed push-down store, while it also stays in the original store.)

The graph has two important vertices: an initial station, where three edges run in from the outside world (these will also be denoted by I, Ī, O), and a final station (the stop station), from which no edges run out. Every other vertex is at the intersection of both incoming and outgoing edges. After this we can start the computation of the value at the argument chain s of the word function f, defined by such a "railway marshalling graph". The first step is to have three trains standing on the three tracks coming in from the outside word to the initial station. These will be called the "kernel trains I, Ī, O" respectively, and they all end with a railcar denoted by the symbol L. To the end of the kernel train I are also joined, one by one, the cars denoted by the symbols of $\bar{s}$. Then the three trains are started in such a way that they arrive at the initial station simultaneously. Here, as at every other station except the final station, the traffic regulations of the graph, depending on the last cars of the three trains standing on the parallel tracks, determine which station they have to proceed to (without changing their track symbols), and whether they remain unchanged in the course of this or undergo one of the modifications 1), 2), 3), 4). They again have to arrive at the next station simultaneously. The "next" station can actually coincide with this one, but in this case at least one of the modifications must be carried out. Cars from the kernel trains are never disconnected.

Following the traffic regulations, the trains might have to return often to the same station, and this can be repeated without any limitation. If the trains never arrive at the final station, then f is not defined for the arguments under consideration.

Furthermore, the traffic instructions are chosen in such a way that if eventually the trains arrive at the final station, then the kernel trains I and Ī will stand on the tracks I and Ī while on track O will stand the kernel train O together with a chain W of cars joined to its end. Here the symbols denoting the cars in W are either letters of the alphabet or $\wedge$. The word composed of these symbols is the value of f for the given arguments. If W contains only $\wedge$, then this word is also $\wedge$. Otherwise, of course, the symbols $\wedge$ have to be omitted. This value is independent of the kernel trains.

Thus, we have sketched a new translation of the "push-down store method". In accordance with section 5.1.3 we have, then: *Everything obtainable by a computer can also be obtained by means of a railway marshalling graph with very simple regulations.* In short, the regulations might, depending on the last cars of the three trains at a station require the disconnection of the last car of a train or the joining of a certain car onto the end of a train.

5.5 Sequential Procedures

I would like to mention here briefly that the significance of sequential procedures, and in particular of the push-down store method, goes far beyond program translations. Of the numerous applications I would like to emphasize its connections with the constructibility of formula controlled computers.

5.5.1 Kalmár's Formula Controlled Computer

As is proved by several patents, F. L. Bauer and K. Samelson have actually considered the application of the method for this purpose. The fundamental ideas of a different solution (compatible with the notation system using parentheses) for formula controlled automata were sketched by L. Kalmár as early as September 1959, at the Warsaw Symposium. He had in mind a computer which can be programmed in a mathematical formula language, and which executes the symbols of a program, written in such a language, one after the other as statements. After this L. Kalmár worked out a version of such computers feasible in practice [21]. According to an oral communication by Z. L. Rabinovič (Cybernetical Institute of the Ukrainian Academy of Sciences), the first universal formula controlled computer was built in this Institute (1963–1966), on the basis of Kalmár's ideas.

[21] See also L. Kalmár: *Über einen Rechenautomaten, der eine mathematische Sprache versteht,* Zeitschrift für Angew. Math. und Mech. **40** (1960) pp. 64–65; and L. Kalmár: *On a digital computer which can be programmed in a mathematical formula language,* Second Hungarian Math. Congress Budapest 24–31 August (1960) Abstract of lectures 2, pp. 3–16.

Chapter 6

Recursivity of Flow Charts

6.1 Graphical Representations

In assembly languages as well as in higher level programming languages (which are closer to the language of mathematics, but farther from the language of a computer) it is usual to accompany the trains of thought that is the logical structure of the program by diagrams called flow charts. These make translation into computer language easier. I shall deal with this notion, which in practice is used without sufficient precision, in the exact form due to Kalužnin, and I shall use his terminology for it: graph scheme.

6.2 Flow Charts in Algol 60

Let us consider the following procedure to define the values of a numeric function $f(a, b)$, given in the programming language Algol 60:

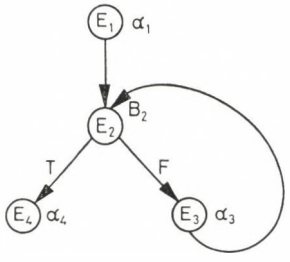
integer procedure $f(a, b)$; **value** a, b; **integer** a, b; **begin integer** i, w; $i:=0$; $w:=1$; c: **if** $i=b$ **then go to** e **else begin** $w:=w\times a$; $i:=i+1$; **go to** c **end;** $e: f:=w$ **end**;

Even for somebody not familiar with the language Algol 60 it is easy to understand what this means; a procedure to compute the integer value of $f(a, b)$, provided that integer values are given to a and b. The procedure consists of statements, while **begin** and **end** play the roles of an opening and closing parenthesis, respectively. First it is stated that the auxiliary variables i, w are also given integer values. Then 0 is set as the initial value of i and 1 as the initial value of w. Now follows a statement marked by the symbol c. If $i=b$, then go (immediately) to the statement marked by e. Otherwise the statements follow in their normal order within the parentheses **begin** and **end**. Replace the actual value of w by $w \cdot a$ and the actual value

of i by $i+1$. Then go back to the statement marked by c: (for cycle). Finally, the statement marked by e: (for end) says that f takes the last value of w. This is the value of the function f for the pair of arguments (a, b).

This value w can be considered as a one-term sequence. The arguments (a, b) form a two-term sequence. The intermediate values also depend on the auxiliary variables i and w, that is on four-term sequences. In the course of the computation sequences of natural numbers are transformed into other similar sequences. This transformation can be represented by a finite, connected, directed graph, the vertices of which are associated with certain variables, where these variables run through sequences of natural numbers with a given number of terms. The values of these functions are also such sequences, possibly with a different number of terms. A vertex associated with a relation is called a logical vertex. Two edges issue from such a vertex, one denoted by T (for true), and one by F (for false). The other vertices are called mathematical vertices. A single edge issues from each such vertex with the exception of one particular vertex, the output vertex, from which no edge issues. Every vertex has an edge which ends there, with the exception of another particular vertex, called the input vertex.

The above computation of $f(a, b)$ is represented by the following graph scheme G: –



where E_1 is the input and E_4 is the output. Moreover

$$\alpha_1(a, b) = (a, b, 0, 1),$$

$$B_2(a, b, i, w) \equiv i = b,$$

$$\alpha_3(a, b, i, w) = (a, b, i+1, w \times a),$$

$$\alpha_4(a, b, i, w) = w.$$

Here $\alpha_1(a, b)$, $\alpha_3(a, b, i, w)$ and $\alpha_4(a, b, i, w)$ are functions, the variables of which are two-term and four-term sequences of natural numbers, respectively. The same applies to the relation

$$B_2(a, b, i, w).$$

As the argument of the function associated with the input E_1 is (a, b), and the value of the function associated with the output is a number, we say that the graph scheme G determines (or computes) a numeric function $G(a, b)$ of two variables. The computation consists of the following steps. To begin with, the argument (a, b) is given to E_1. Then α_1 is computed at this place and its value $(a, b, 0, 1)$ is sent as an argument along the unique edge leading from E_1 to E_2. From this logical vertex, it is sent further as an unchanged argument along either edge T or edge F to the vertex E_4 or E_3, according to whether B_2 is true or false that is $0=b$ or $0\neq b$. In the second case the function α_3 associated with E_3 is computed for this argument, and its value $(a, b, 1, a)$ is sent back to E_2 along the edge leading from it. Here everything starts all over again. Whenever a certain (a, b, i, w) is taken to E_3, the value $(a, b, i+1, w \cdot a)$ is sent back from here to E_2, to see whether the third term has become equal to b. If the answer is affirmative, then we proceed along the edge T to the output E_4, and the value of α_4 obtained here is the value of the function $G(a, b)$ computed by G. As can be seen from the above description of the computation steps, this coincides with $f(a, b)$ computed by our Algol procedure.

Actually, $G(a, b)$ is a well-known function. Indeed, the above procedure yields us the following (denoting the transition from certain number sequences to others by $\rightarrow$ and $\frac{|}{|\ |}$):

$$(a, b) \longrightarrow (a, b, 0, 1)$$

$$\frac{|}{|\quad|}$$

$$1, \text{ if } b=0 \quad (a, b, 1, a) \text{ otherwise}$$

$$\frac{|}{|\quad|}$$

$$a, \text{ if } b=1 \quad (a, b, 2, a \cdot a) \text{ otherwise}$$

$$\frac{|}{|\quad|}$$

$$a \cdot a, \text{ if } b=2 \quad (a, b, 3, a \cdot a \cdot a) \text{ otherwise}$$

$$\frac{|}{|\quad|}$$

$$a \cdot a \cdot a, \text{ if } b=3 \quad \ldots$$

Thus we see that G computes the bth power of a, that is

$$f(a, b) = G(a, b) = a^b.$$

6.3 Flow Charts of Word Functions

It is possible to read off the determining graph scheme directly from the definition of a function.

Let us consider e.g. the following definition of a function $f(x, y)$ in a word set M over a finite alphabet A: –

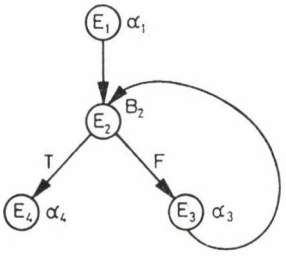
$$f(x, y) = g(x, \wedge, y),$$

where g is defined by

$$g(x, u, v) = \begin{cases} v, & \text{if} \quad o(x) = o(u) \\ g(x, ua_0, \text{at}(v)) & \text{otherwise.} \end{cases}$$

Here a_0 is an element of A, such that the words which are built out of it represent in M the natural numbers and consequently also the orders of the words.

The definition shows that the first step is to move from (x, y) to $(x, \wedge, y)$. Then begins the computation of g by cases, according to whether or not the orders of the first and second terms coincide. In the first case one has to take the third term of the three-term word sequence. In the second case, however, one has to pass to a new three-term sequence, with ua_0 instead of the previous u as the second term, and with $\text{at}(v)$ instead of the previous v as the third term. Then one has to return with this new argument to the point where the computation of g began. This can be represented by a graph of the same structure as above: –



However with

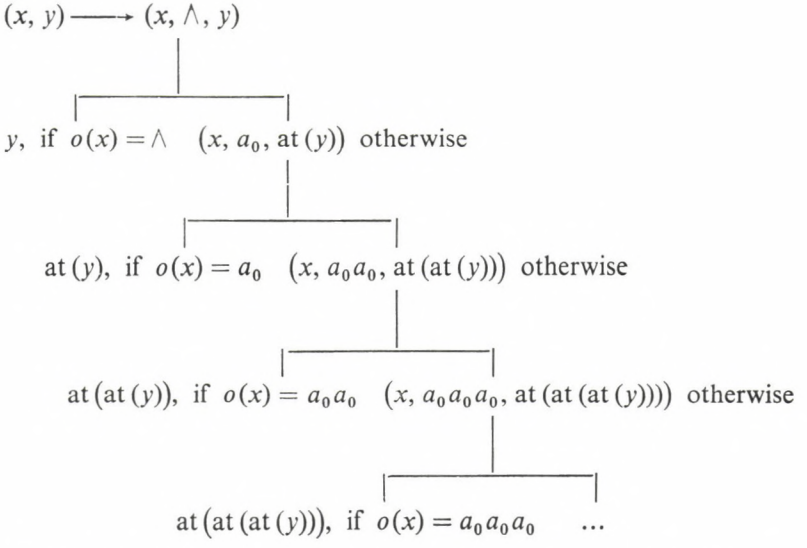
$$\alpha_1(x, y) = (x, \wedge, y),$$

$$B_2(x, u, v) \equiv o(x) = o(u),$$

$$\alpha_3(x, u, v) = (x, ua_0, \text{at}(v)),$$

$$\alpha_4(x, u, v) = v.$$

The function determined by this graph scheme is computed through the following steps; –



Clearly, this is the $o(x)$th iteration of the function at for the argument y: –

$$f(x, y) = \mathrm{at}^{(o(x))}(y).$$

6.4 Partial Recursivity of Flow Charts

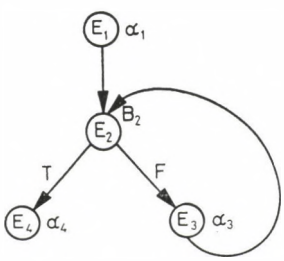
In general, with every graph scheme G, there is associated a set M_G in such a way that the domains of the functions, the ranges of the former, and the relations associated with the vertices are subsets of M_G. The same is then true for the function determined by G. This is defined for those elements of M_G, for which as input arguments, the computational procedure described in the examples never gets stuck before reaching the output. This can happen as early as at the input, if the function or relation associated with it is not defined for the input argument. Also the procedure must not contain an infinite cycle, and the function associated with the (always mathematical) output vertex must always be defined for the incoming argument.

In what follows, I will restrict myself to graph schemes defining numeric functions of an arbitrary number of variables. Then M_G can be chosen as the set of all finite sequences of natural numbers.

One has to take certain initial functions and relations that can be associated with the vertices. Then the functions defined by such graph schemes can be

associated with the mathematical vertices of new graph schemes, and so on. It is easy to see, however, that if a mathematical vertex E of a graph scheme G is replaced by the graph scheme determining the function associated with E, then the modified graph scheme computes the same function as G. Therefore we can restrict ourselves to graph schemes, the mathematical vertices of which only have the initial functions associated with them. With a suitable choice of the initial functions and relations, I have proved [22] that *every numeric function computable by a graph scheme is partial recursive.* The proof will be illustrated for the example of section 6.2 without reducing the functions occurring there to the initial functions.

Let us consider again this graph scheme and the functions and relations associated with its vertices: –



$$\alpha_1(a, b) = (a, b, 0, 1)$$

$$B_2(a, b, i, w) \equiv i = b$$

$$\alpha_3(a, b, i, w) = (a, b, i+1, w \cdot a)$$

$$\alpha_4(a, b, i, w) = w.$$

We can code finite sequences of numbers by natural numbers, for example the sequence $(n_1, n_2, \ldots, n_l)$ by the number $n = 2^{l+1} \cdot p_1^{n_1} \cdot p_2^{n_2} \cdot \ldots \cdot p_l^{n_l}$. Then the sequence coded by n is simply denoted by a_n. Here, as before, p_i is the ith prime number, with 2 considered as the 0th. The exponent of p_i in the prime factor representation of n will be denoted by $\exp_i(n)$. Taking into account that the nth prime number is certainly bigger than n, we see that the number n codes an l-term sequence if the following primitive recursive relation holds: –

$$Z_l(n) \equiv \exp_0(n)+1 = l \,\&\, (i)\big[i \leqq n \to (i > l \to \exp_i(n) = 0)\big].$$

[22] R. Péter: *Über die Partiell-Rekursivität der durch Graphschemata definierten zahlentheoretischen Funktionen,* Ann. Univ. Sci. Budapest **2** (1959) pp. 41–48.

The sequences occurring in the definition of α_1, α_3 and α_4 can be coded as follows:

$$(a, b) = a_{n_1} \quad \text{with} \quad n_1 = 2^1 \cdot 3^a \cdot 5^b$$

and

$$(a, b, 0, 1) = a_{m_1} \quad \text{with} \quad m_1 = 2^3 \cdot 3^a \cdot 5^b \cdot 7^0 \cdot 11^1 = 44 \cdot n_1.$$

Also

$$(a, b, i, w) = a_{n_2} \quad \text{with} \quad n_2 = 2^3 \cdot 3^a \cdot 5^b \cdot 7^i \cdot 11^w$$

and

$$(a, b, i+1, w \cdot a) = a_{m_2} \quad \text{with} \quad m_2 = 2^3 \cdot 3^a \cdot 5^b \cdot 7^{i+1} \cdot 11^{w \cdot a} =$$

$$= 8 \cdot 3^{\exp_1(n_2)} \cdot 5^{\exp_2(n_2)} \cdot 7^{\exp_3(n_2)+1} \cdot 11^{\exp_4(n_2) \cdot \exp_1(n_2)},$$

$$w = a_{m_3} \quad \text{with} \quad m_3 = 2^0 \cdot 3^w = 3^{\exp_4(n_2)}.$$

Let us put for $j = 1, 3, 4$

$$\beta(j, n) = \begin{cases} m, & \text{if} \quad \alpha_j(a_n) = a_m \\ 0, & \text{if} \quad \alpha_j(a_n) \text{ is undefined.} \end{cases}$$

Then, since α_1 is defined for two-term sequences and α_3 and α_4 for four-term sequences, $\beta(j, n)$ is determined by the following definition by cases as a primitive recursive numeric function: –

$$\beta(j, n) = \begin{cases} 44n, & \text{if} \quad j = 1 \,\&\, Z_2(n) \\ 8 \cdot 3^{\exp_1(n)} \cdot 5^{\exp_2(n)} \cdot 7^{\exp_3(n)+1} \cdot 11^{\exp_4(n) \cdot \exp_1(n)}, & \text{if} \quad j = 3 \,\&\, Z_4(n) \\ 3^{\exp_4(n)}, & \text{if} \quad j = 4 \,\&\, Z_4(n) \\ 0 & \text{otherwise.} \end{cases}$$

For $n = 0$ a_n was not defined. Therefore $\beta(j, n)$ vanishes if and only if $\alpha_j(a_n)$ is not defined.

The primitive recursive function $\gamma(n)$, belonging to the logical vertex, and defined by

$$\gamma(n) = \begin{cases} n, & \text{if} \quad Z_4(n) \\ 0 & \text{otherwise} \end{cases}$$

(in accordance with the fact that the values coming in to logical vertices are sent onwards unchanged) vanishes if and only if $B_2(a_n)$ is not defined, since B_2 is defined for four-term sequences. Moreover if

$$n = 2^3 \cdot 3^a \cdot 5^b \cdot 7^i \cdot 11^w, \quad \text{that is} \quad a_n = (a, b, i, w),$$

then $\gamma(n) \neq 0$ and

$$B_2(a_n) \equiv i = b \equiv \exp_3(n) = \exp_2(n).$$

6.4.1 Recursivity of Graphical Structure

So far we have defined primitive recursive counterparts of the functions and relations associated with the vertices. Now the structure of the graph G has to be described in a primitive recursive way.
For this purpose we define the following functions:

$$\nu(j) = \begin{cases} 0, & \text{if } E_j \text{ is a mathematical vertex,} \\ 1, & \text{if } E_j \text{ is a logical vertex,} \\ 2 & \text{otherwise;} \end{cases}$$

$$\tau(j) = \begin{cases} j', & \text{if the edge starting out of the} \\ & \text{mathematical vertex } E_j \text{ leads to } E_{j'} \\ 0 & \text{otherwise;} \end{cases}$$

$$\varkappa(j) = \begin{cases} j', & \text{if the edge T starting at the logical} \\ & \text{vertex } E_j \text{ leads to } E_{j'} \\ 0 & \text{otherwise;} \end{cases}$$

$$\lambda(j) = \begin{cases} j', & \text{if the edge F starting at the logical} \\ & \text{vertex } E_j \text{ leads to } E_{j'} \\ 0 & \text{otherwise.} \end{cases}$$

These can be defined as primitive recursive functions by the following definitions by cases: –

$$\nu(j) = \begin{cases} 0, & \text{if } j = 1 \vee j = 3 \vee j = 4 \\ 1, & \text{if } j = 2 \\ 2 & \text{otherwise,} \end{cases}$$

$$\tau(j) = \begin{cases} 2, & \text{if } j = 1 \vee j = 3 \\ 0 & \text{otherwise,} \end{cases}$$

$$\varkappa(j) = \begin{cases} 4, & \text{if } j = 2 \\ 0 & \text{otherwise,} \end{cases}$$

$$\lambda(j) = \begin{cases} 3, & \text{if } j = 2 \\ 0 & \text{otherwise.} \end{cases}$$

The computation of $\mathsf{G}(a, b)$ is carried out at successive moments, where in "moment 0" the argument (a, b) is to be taken, and in moment 1 the function or relation associated with E_1 is to be dealt with.
Let

$$\iota(r, a, b) = \begin{cases} j, & \text{if in moment } r \text{ the computation has} \\ & \text{to deal with } E_j, \\ 0 & \text{otherwise.} \end{cases}$$

Furthermore

$$\omega(r, a, b) = \begin{cases} \text{the number corresponding to the argument } (a, b) \text{ if } r=0; \\ \text{otherwise: the value of the function } \beta \text{ or } \gamma \text{ belonging to} \\ \text{the vertex of index } \iota(r, a, b), \text{ computed at moment } r, \\ \text{according to whether } E_{\iota(r,a,b)} \text{ is a mathematical or} \\ \text{logical vertex.} \end{cases}$$

Hence these functions can be defined by the following simultaneous recursion: –

$$\iota(0, a, b) = 0,$$

$$\iota(1, a, b) = 1,$$

and for $r \geqq 1$

$$\iota(r+1, a, b) = \begin{cases} \tau(\iota(r, a, b)), & \text{if} \quad v(\iota(r, a, b)) = 0 \\ (\iota(r, a, b)), & \text{if} \quad v(\iota(r, a, b)) = 1 \;\&\; \exp_3(\omega(r, a, b)) = \\ \quad = \exp_2(\omega(r, a, b)) & \\ \lambda(\iota(r, a, b)), & \text{if} \quad v(\iota(r, a, b)) = 1 \;\&\; \exp_3(\omega(r, a, b)) \neq \\ \quad \neq \exp_2(\omega(r, a, b)) & \\ 0 & \text{otherwise,} \end{cases}$$

$$\omega(0, a, b) = 2 \cdot 3^a \cdot 5^b,$$

$$\omega(r+1, a, b) = \begin{cases} \beta(\iota(r+1, a, b), \omega(r, a, b)), & \text{if} \quad v(\iota(r+1, a, b)) = 0 \\ \gamma(\omega(r, a, b)), & \text{if} \quad v(\iota(r+1, a, b)) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

The $\iota(r+1, a, b)$ occurring in the definition of $\omega(r+1, a, b)$ can be replaced by the right-hand side of the definition of $\iota(r+1, a, b)$. Thus not only $\iota(r+1, a, b)$, but also $\omega(r+1, a, b)$ can be defined as primitive recursive functions of $\iota(r, a, b)$ and $\omega(r, a, b)$. Such a simultaneous recursion, however, can be always reduced to primitive recursive definitions of

$$\iota(r, a, b) \quad \text{and} \quad \omega(r, a, b).$$

The value of $G(a, b)$ is obtained at the first moment when the output vertex E_4 is reached, and with a value, for which α_4 is defined, such that the corresponding value of β, and consequently of ω, is not 0; in other words, at the first moment r, in which both

$$\iota(r, a, b) = 4 \quad \text{and} \quad \omega(r, a, b) \neq 0$$

hold. The value of α_4 at moment r is the value of w at this moment: a single number, that is a one-term sequence a_n, whose code is

$$n = 2^0 \cdot 3^w = 3^w.$$

Thus 3^w is the value of β belonging to this vertex in this moment, that is $\omega(r, a, b)$; now w can be obtained from this as the exponent of 3 in its prime factor representation, that is as

$$\exp_1(\omega(r, a, b)).$$

Thus we have

$$G(a, b) = \exp_1(\omega(\mu_r[\iota(r, a, b) = 4 \ \& \ \omega(r, a, b) \neq 0], a, b)).$$

6.5 The Computability of Flow Charts

It is easy to give an upper bound for the μ-operation applied here: Whenever a vertex is reached, one stays there for a moment. E_1 and E_4 are reached only once, E_2 and E_3 as many times as there are values of i to increase to b from 0, that is, both are reached b times. This yields the upper bound $2b+2$ for r. Hence the function

$$G(a, b) = \exp_1(\omega(\mu_r[r \leqq 2+2b \,\&\, \iota(r, a, b) = 4 \,\&\, \omega(r, a, b) \neq 0], a, b)),$$

determined by G, is primitive recursive.

Of course we knew this already, since $G(a, b)$ is equal to the power a^b. But a similar way of reasoning applies to the general case, showing that all the numeric functions computable by graph schemes (with suitably chosen initial functions) are obtained from primitive recursive functions through substitutions and a single μ-operation, and consequently they are partial recursive. According to Chapter 4, however, every partial recursive function is computable by a computer. Hence we can conclude: –

Whatever can be computed by a graph scheme, is also computable by a computer.

Chapter 7

Recursive Procedures and Algol 60

7.1 The Converse Results

What happens to the converse of the final conclusion in the previous chapter? Everything obtainable by a computer is partial recursive. Can all the partial recursive functions be computed by graph schemes?
In sections 6.2 and 6.3, this was shown for two particular cases, for the primitive recursive numeric function a^b, and for the primitive recursive word function

$$f(x, y) = \mathrm{at}^{(o(x))}(y).$$

In the latter case, it was somewhat obscured by not taking the natural primitive recursive definition of $f(x, y)$: –

$$f(x, y) = \begin{cases} y, & \text{if} \quad x = \wedge \\ \mathrm{at}\,(f(\mathrm{at}\,(x), y)) & \text{otherwise.} \end{cases}$$

Is it not possible to obtain from this a graph scheme determining $f(x, y)$?
This definition starts with the decision whether $x = \wedge$ or not. This gives a logical input vertex E_1 with the associated relation

$$B_1(x, y) \equiv x = \wedge.$$

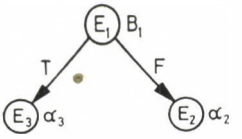
The edge F starting out of this vertex has to lead to a mathematical vertex E_2, with which is associated the function

$$\alpha_1(x, y) = (\mathrm{at}\,(x), y).$$

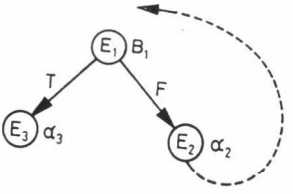
Similarly the edge T has to lead to the output vertex, say E_3, with which is associated the function

$$\alpha_3(x, y) = y.$$

This yields the final result. So far everything is all right: –



But how do we proceed after having reached E_2? Our process fails here. We would have to apply f to the argument $(\mathrm{at}(x), y)$. Hence the edge originating at E_2 would have to lead back to E_1, where the computation of f started: –



However, this would start a rotation between E_1 and E_2, with pairs whose second term is invariably y, and the order of whose first term is 1 less after each turn. The "at" applied to f in the definition could not even be mentioned. After $o(x)$ turns, one would end up in E_1 with the pair $(\wedge, y)$, for which B_1 is true. Then one has to proceed to E_3, where y is obtained as the result. Hence this graph scheme would not compute

$$\mathrm{at}^{(o(x))}(y),$$

but

$$f(x, y) = y.$$

Thus it was with good reason that we used a different definition of

$$\mathrm{at}^{(o(x))}(y).$$

7.2 Recursion in Algol 60

The numeric function

$$f(a, b) = a^b$$

was not given by a definition, but by an Algol procedure. The natural definition would be: –

$$f(a, b) = \begin{cases} 1, & \text{if } b = 0 \\ f(a, b \dot{-} 1) \cdot a & \text{otherwise.} \end{cases}$$

It would not be correct in this case either to deduce from this definition a graph scheme determining a^b. In the same way we would have to take a logical input vertex E_1 and the relation

$$B_1(a, b) \equiv b = 0$$

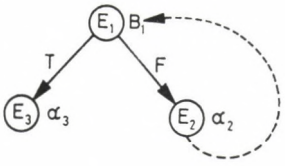
associated with it. From E_1 the edge F must lead to a mathematical vertex E_2 with

$$\alpha_2(a, b) = (a, b \dot{-} 1),$$

and the edge T must lead to the output vertex E_3 with

$$\alpha_3(a, b) = 1.$$

Thus we would have



Here f should be applied again to the pair $(a, b \dot{-} 1)$ obtained in E_2. Therefore the edge starting out of E_2 would have to lead back to the start of the computation of f, that is to E_1. This cycle would have to continue with the second term decreased by 1 for each repeat (without even mentioning the multiplication by a), until finally the second argument is 0. Hence B_1 holds, in which case one has to proceed to the output E_3. Here the constant 1 is obtained as the result of the computation. Consequently this graph scheme does not compute a^b.

From the above natural definition of a^b we should obtain the following Algol procedure for its computation: –

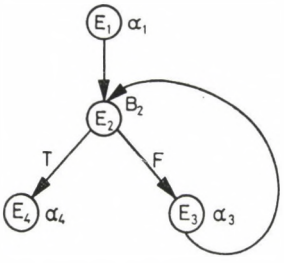
integer procedure $f(a, b)$; **value** a, b; **integer** a, b;
f:= **if** $b=0$ **then** 1 **else** $f(a, b-1)\times a$;

This is essentially different from the procedure given in section 6.2; namely it calls itself for the computation of f at another place. Such sections of program are called recursive procedures. Among these are counted the simultaneous procedures which call one another mutually.

7.3 Non-recursive Algol Procedures

If we could have deduced from the primitive recursive definition of $f(a, b) = a^b$ a graph scheme determining it, then this would have yielded us a non-recursive procedure. Indeed, in general [23], *if a numeric function is computable by a graph scheme, then a non-recursive Algol procedure can be given for its computation.*

The general reasoning will again be illustrated with the example of a graph scheme G, which was used earlier as an example of a general kind in section 6.4: –



Now, however, the variables of the different functions and relations have to be denoted in a different way, and in the values of the functions (which are finite sequences) the dependences of the separate terms on the corresponding variables have to be indicated. (For the sake of clarity, I shall use lower indices and also Greek letters. It is easy to replace these by expressions admissible in Algol.) Thus we have

$$\alpha_1(v_{1,1}, v_{1,2}) = (\alpha_{1,1}(v_{1,1}, v_{1,2}), \alpha_{1,2}(v_{1,1}, v_{1,2}), \alpha_{1,3}(v_{1,1}, v_{1,2}), \alpha_{1,4}(v_{1,1}, v_{1,2})),$$

where

$$\alpha_{1,1}(v_{1,1}, v_{1,2}) = v_{1,1};\ \alpha_{1,2}(v_{1,1}, v_{1,2}) = v_{1,2};\ \alpha_{1,3}(v_{1,1}, v_{1,2}) = 0;$$

$$\alpha_{1,4}(v_{1,1}, v_{1,2}) = 1;$$

$$B_2(v_{2,1}, v_{2,2}, v_{2,3}, v_{2,4}) \equiv v_{2,3} = v_{2,2};$$

$$\alpha_3(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4}) = (\alpha_{3,1}(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4}), \alpha_{3,2}(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4}),$$
$$\alpha_{3,3}(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4}), \alpha_{3,4}(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4})),$$

where

$$\alpha_{3,1}(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4}) = v_{3,1};\ \alpha_{3,2}(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4}) = v_{3,2};$$

$$\alpha_{3,3}(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4}) = v_{3,3}+1;\ \alpha_{3,4}(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4}) = v_{3,4}\times v_{3,1};$$

$$\alpha_4(v_{4,1}, v_{4,2}, v_{4,3}, v_{4,4}) = \alpha_{4,1}(v_{4,1}, v_{4,2}, v_{4,3}, v_{4,4}),$$

[23] R. Péter: *Die prinzipielle Ausschaltbarkeit des rekursiven Prozeduren aus der Programmierungssprache Algol 60,* Acta Cybernetica **1** (1972) pp. 219—231.

whence

$$\alpha_{4,1}(v_{4,1}, v_{4,2}, v_{4,3}, v_{4,4}) = v_{4,4}.$$

The heading of the Algol procedure deduced from G reads as follows: –

integer procedure $f(a, b)$; **value** a, b; **integer** a, b;

In the procedure body first the necessary variables are declared by

begin integer $v_{1,1}, v_{1,2}, v_{2,1}, v_{2,2}, v_{2,3}, v_{2,4}, v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4}, v_{4,1}, v_{4,2},$

$v_{4,3}, v_{4,4}$.

Then, starting at E_1, the statements belonging to the vertices follow, which if necessary can be marked by the symbols of the corresponding vertices. Firstly by the general method we have

$v_{1,1}:=a$; $v_{1,2}:=b$; $v_{2,1}:=\alpha_{1,1}(v_{1,1}, v_{1,2})$; $v_{2,2}:=\alpha_{1,2}(v_{1,1}, v_{1,2})$; $v_{2,3}:=\alpha_{1,3}$ $(v_{1,1}, v_{1,2})$; $v_{2,4}:=\alpha_{1,4}(v_{1,1}, v_{1,2})$; E_2: **if** $v_{2,3}=v_{2,2}$ **then begin** $v_{4,1}:=v_{2,1}$; $v_{4,2}:=v_{2,2}$; $v_{4,3}:=v_{2,3}$; $v_{4,4}:=v_{2,4}$; **go to** E_4 **end else begin** $v_{3,1}:=v_{2,1}$; $v_{3,2}:=v_{2,2}$; $v_{3,3}:=v_{2,3}$; $v_{3,4}:=v_{2,4}$; $v_{2,1}:=\alpha_{3,1}(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4})$; $v_{2,2}:=\alpha_{3,2}(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4})$; $v_{2,3}:=\alpha_{3,3}(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4})$; $v_{2,4}:=\alpha_{3,4}$ $(v_{3,1}, v_{3,2}, v_{3,3}, v_{3,4})$; **go to** E_2; **end**; E_4: $f:=\alpha_{4,1}(v_{4,1}, v_{4,2}, v_{4,3}, v_{4,4})$ **end**;

In our example, replacing each $\alpha_{i,j}$ by its value, we obtain the following statements: –

$v_{1,1}:=a$; $v_{1,2}:=b$; $v_{2,1}:=v_{1,1}$; $v_{2,2}:=v_{1,2}$; $v_{2,3}:=0$; $v_{2,4}:=1$; E_2: **if** $v_{2,3}=v_{2,2}$ **then begin** $v_{4,1}:=v_{2,1}$; $v_{4,2}:=v_{2,2}$; $v_{4,3}:=v_{2,3}$; $v_{4,4}:=v_{2,4}$; **go to** E_4 **end else begin** $v_{3,1}:=v_{2,1}$; $v_{3,2}:=v_{2,2}$; $v_{3,3}:=v_{2,3}$; $v_{3,4}:=v_{2,4}$; $v_{2,1}:=v_{3,1}$; $v_{2,2}:=v_{3,2}$; $v_{2,3}:=v_{3,3}+1$; $v_{2,4}:=v_{3,4}\times v_{3,1}$; **go to** E_2 **end**; E_4: $f:=v_{4,4}$ **end**;

In this particular case several further simplifications are possible. Firstly, one sees that during the procedure $v_{1,1}$, $v_{2,1}$, $v_{3,1}$ and $v_{4,1}$ take only the value a: further $v_{1,2}$, $v_{2,2}$, $v_{3,2}$ and $v_{4,2}$ take only the value b. Therefore these variables are superfluous, wherever they occur they can be replaced by a and b respectively. Moreover, it can be seen that $v_{3,3}:=v_{2,3}$, and then $v_{2,3}:=v_{3,3}+1$ can be replaced by the statement $v_{2,3}:=v_{2,3}+1$. Similarly $v_{3,4}:=v_{2,4}$ and then $v_{2,4}:=v_{3,4}\times v_{3,1}$. That is, $v_{2,4}:=v_{3,4}\times a$ can be replaced by the statement $v_{2,4}:=v_{2,4}\times a$, finally $v_{4,4}:=v_{2,4}$ and then $f:=v_{4,4}$ can be replaced by $f:=v_{2,4}$. Consequently the variables $v_{3,3}$, $v_{3,4}$ and $v_{4,4}$ are also superfluous. So is $v_{4,3}$, which is not used at all. After all these simplifications we obtain the following Algol procedure: –

integer procedure $f(a, b)$; **value** a, b; **integer** a, b; **begin integer** $v_{2,3}, v_{2,4}$; $v_{2,3}:=0$; $v_{2,4}:=1$; E_2: **if** $v_{2,3}=b$ **then go to** E_4 **else begin** $v_{2,3}:=v_{2,3}+1$; $v_{2,4}:=v_{2,4}\times a$; **go to** E_2 **end** E_4: $f:=v_{2,4}$ **end**;

This, with the notation

$$v_{2,3},\ v_{2,4},\ E_2,\ E_4$$

instead of

$$i,\quad w,\quad c,\quad e,$$

coincides with the non-recursive procedure from which the graph scheme G determining f was deduced in section 6.2.

7.4 Unfolding a Primitive Recursion

According to the above, it might seem that the functions defined by primitive recursion are in general not computable by graph schemes. However in my paper mentioned in footnote [23], I have proved that this is not the case. Primitive recursions can always be replaced by other definitions suitable for the purpose.

Let us consider the general case of the definition by primitive recursion of a two-place numeric function $f(a, b)$. The order of the variables is irrelevant so

$$f(a, b) = \begin{cases} g_0(a), & \text{if} \quad h = 0 \\ g\big(a, b \dot{-} 1, f(a, b \dot{-} 1)\big) & \text{otherwise.} \end{cases}$$

We assume that we already have Algol procedures for the computation of the functions g_0 and g. This suggests the following Algol procedure:

integer procedure $f(a, b)$; **value** a, b; **integer** a, b; $f:=$**if** $b=0$ **then** $g_0(a)$ **else** $g\big(a, b-1, f(a, b-1)\big)$;

In the "else" case this procedure calls itself to compute the value of f for the arguments $a, b \dot{-} 1$. An ordinary computer program cannot do anything with such a situation, unless the procedure is suitably expanded. From the definition of f we obtain gradually the following: –

If $b=0$, then

$$f(a, b) = g_0(a);$$

otherwise

$$f(a, b) = g\big(a, b \dot{-} 1, f(a, b \dot{-} 1)\big).$$

If here $b=1$, so that $b \dot{-} 1=0$, then $f(a, b \dot{-} 1)=g_0(a)$ and

$$f(a, b) = g\big(a, b \dot{-} 1, g_0(a)\big) = g\big(a, 0, g_0(a)\big).$$

Otherwise, given that $(b \dot{-} 1) \dot{-} 1 = b \dot{-} 2$, then

$$f(a, b \dot{-} 1) = g(a, b \dot{-} 2, f(a, b \dot{-} 2)).$$

Hence we have to compute

$$f(a, b) = g(a, b \dot{-} 1,\ g(a, b \dot{-} 2, f(a, b \dot{-} 2))).$$

Now, if $b=2$, then

$$f(a, b \dot{-} 2) = g_0(a).$$

Hence

$$f(a, b) = g(a, b \dot{-} 1,\ g(a, b \dot{-} 2,\ g_0(a))) = g(a, 1, g(a, 0, g_0(a)))$$

and so on.

It follows that for every $b>0$

$$f(a, b) = g(a, b \dot{-} 1,\ g(a, b \dot{-} 2, \ldots, g(a, 1, g(a, 0, g_0(a)))\ldots))$$

holds.

Only after this expansion can the machine computation be carried out, step by step. To begin with (in "step 0") $g_0(a)$ is computed. Then with the value w we obtained here $g(a, 0, w)$ is computed; with the new w value $g(a, 1, w)$ is computed, and so on. If, in general, in step i the value w is obtained, then in step $i+1$ the computation of $g(a, i, w)$ follows. Finally, $f(a, b)$ is that value w which is obtained in step b.

This is reflected by the following definition of an auxiliary function h, which for $i<b$ gives the transition from step i to $i+1$, and from the actual value w to $g(a, i, w)$, while for $i=b$ yields the actual value of w:

$$h(a, b, i, w) = \begin{cases} w, & \text{if} \quad i = b \\ h(a, b, i+1, g(a, i, w)) & \text{otherwise.} \end{cases}$$

Since in step 0 we have $w=g_0(a)$, it remains to be proved that

$$h(a, b, 0, g_0(a)) = f(a, b). \tag{7.4.1}$$

Clearly, it suffices to prove the following proposition: –

For $i \leqq b$ we have

$$h(a, b, 0, f(a, 0)) = h(a, b, i, f(a, i)). \tag{7.4.2}$$

Indeed, for $i=b$, using $f(a, 0)=g_0(a)$, we obtain from this exactly (7.4.1). Now (7.4.2) is clearly satisfied if $b=0$, since then i ($\leqq b$) is also 0. Hence both sides are identical.

For $b \neq 0$ (7.4.2) is proved by induction on i. For $i=0$ both sides are identical. Assuming that (7.4.2) is valid for $i<b$, we shall show that it is also valid for $i+1$. Indeed, by the definitions of h and f we have

$$h(a, b, 0, f(a, 0)) = h(a, b, i, f(a, i)) = h(a, b, i+1, g(a, i, f(a, i))) =$$
$$= h(a, b, i+1, f(a, i+1)).$$

If, in particular

$$g_0(a) = 1 \quad \text{and} \quad g(a, b, w) = w \cdot a,$$

then we have the primitive recursion defining

$$f(a, b) = a^b$$

as in section 4.2. This function can be defined also as

$$f(a, b) = h(a, b, 0, 1),$$

with

$$h(a, b, i, w) = \begin{cases} w, & \text{if} \quad i = b \\ h(a, b, i+1, w \cdot a) & \text{otherwise.} \end{cases} \tag{7.4.3}$$

An added parameter or the omission of one or a series of arguments does not change the above proof.

The numeric function determined by the general primitive recursion

$$\begin{cases} f(0, a_1, \ldots, a_r) = g_0(a_1, \ldots, a_r) \\ f(n+1, a_1, \ldots, a_r) = g(n, a_1, \ldots, a_r, f(n, a_1, \ldots, a_r)) \end{cases}$$

is also definable as

$$f(n, a_1, \ldots, a_r) = h(n, a_1, \ldots, a_r, 0, g_0(a_1, \ldots, a_r)),$$

with

$$h(n, a_1, \ldots, a_r, i, w) = \begin{cases} w, & \text{if} \quad i = n \\ h(n, a_1, \ldots, a_r, i+1, g(i, a_1, \ldots, a_r, w)) & \text{otherwise.} \end{cases}$$

7.4.1 The Resulting Flow Chart

The situation is similar for primitive recursions in a word set. Indeed, the application of similar considerations for the primitive recursion from section 7.1, by which the word function

$$\mathrm{at}^{(o(x))}(y)$$

was defined, leads to the definition of this function given in section 6.3, which is analogous to the above. In other words, it leads to the definition, from which the graph scheme determining

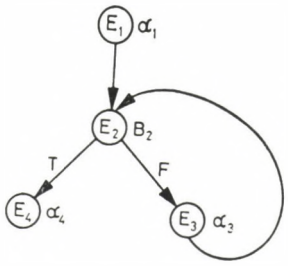
$$\mathrm{at}^{(o(x))}(y)\text{-}$$

was deduced.

Similarly, it is possible to deduce, from definition (7.4.3) of the numeric function $f(a, b) = a^b$, a graph scheme determining it, which coincides with the graph scheme deduced from the Algol procedure in section 6.3.

In (7.4.3), one first has to proceed from (a, b) to $(a, b, 0, 1)$. Then begins

the computation of h by cases, according to whether the second and third terms of a four-term sequence coincide or not. In the first case, one has to take the fourth term of the sequence as the result ("output"). In the second, a new four-term sequence is to be taken, in which the first and second terms remain unchanged, but instead of i we have $i+1$, and instead of w we have $w \cdot a$, as the third and fourth terms respectively. Then, with this new four-term sequence, one has to go back to the computation of h. This is represented by the graph scheme: –



with

$$\alpha_1(a, b) = (a, b, 0, 1)$$

$$B_2(a, b, i, w) \equiv i = b,$$

$$\alpha_3(a, b, i, w) = (a, b, i+1, w \cdot a),$$

$$\alpha_4(a, b, i, w) = w.$$

This is indeed the same as that from which, in section 7.3, the non-recursive Algol procedure determining a^b was deduced.

7.5 Normal Flow Charts

In order to be able to do the same in the general case, finally we have to make precise the initial functions and relations to be associated with the vertices of graph schemes.

We take as initial functions those functions α, which make k-term sequences into l-term sequences:

$$\alpha(n_1, \ldots, n_k) = (m_1, \ldots, m_l)$$

in such a way that every m_i $(i=1, 2, \ldots, l)$ is either n_j or n_j+1 for some $j=1, 2, \ldots, k$, or is equal to 0.

As initial relations we take those of the form

$$B(n_1, \ldots, n_k) \equiv m_1 = m_2,$$

where both m_1 and m_2 are one of the n_i $(i=1, 2, \ldots, k)$.

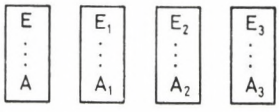
A graph scheme with every vertex associated with an initial function or relation is called a normal scheme. The empty scheme is also assumed to be normal.
I claim now that *every primitive recursive function can be determined by a normal scheme* [24].

7.5.1 Determining Recursive Functions by Flow Charts

The initial functions of normal schemes (which, as special cases, contain the initial functions 0 and $n+1$ of the primitive recursive functions) can immediately be defined by normal schemes. The corresponding scheme consists of a single vertex, which is both the input and the output. The function to be defined is associated with this vertex of course. It now remains to be shown that computability by a normal scheme is preserved under substitutions and primitive recursions.
Assume, for instance, that the functions

$$f(a, b, c), \quad g_1(a, b), \quad g_2(a, b), \quad g_3(a, b)$$

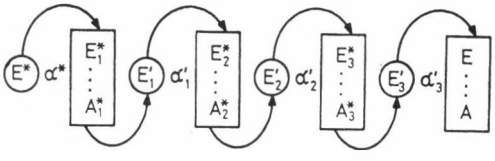
are determined by normal schemata, represented by the following blocks in which only the input and output vertices are explicitly indicated: –



Then the function

$$f(g_1(a, b),\ g_2(a, b),\ g_3(a, b)),$$

obtained by substitution, is determined by the following normal scheme: –



[24] See R. Péter: *Graphschemata und rekursive Funktionen,* Dialectica **12** (1958) pp. 373–393. Concerning these arguments, see also my paper quoted in footnote [12]. Kalužnin's definition of the graph schemes became known to me through an indirect oral communication. Later it appeared in L. A. Kalužnin: *Ob algoritmizacii mathematiceskich zadač,* Problemi Kibernetiki **2** Moscow (1959) pp. 51–67.

where the following modifications of the above blocks (indicated by asterisks) have to be executed: –

After receiving the values for a and b first $g_1(a, b)$ has to be computed. However the argument (a, b) must be preserved since it is also used for the computation of $g_2(a, b)$. Therefore the input vertex E^* is taken with

$$\alpha^*(a, b) = (a, b, a, b),$$

and the block computing $g_1(a, b)$ has to be modified so that every sequence $(r_1, r_2, \ldots, r_s)$ occurring as an argument or value at one of its vertices has to be replaced by

$$(a, b, r_1, \ldots, r_s).$$

In particular, at the output vertex A_1^* we obtain, instead of w_1 (the value of g_1 for the arguments a, b), the sequence

$$(a, b, w_1).$$

This sequence again has to be preserved, a and b for the computation of $g_3(a, b)$, and w_1 as the first argument to be put into f. That is why E_1' is added with

$$\alpha_1'(a, b, w_1) = (a, b, w_1, a, b).$$

Now the block computing $g_2(a, b)$ has to be modified so that every sequence $(r_1, \ldots, r_s)$ occurring as an argument or value at one of its vertices is replaced by

$$(a, b, w_1, r_1, \ldots, r_s).$$

Thus, at A_2^* the sequence (a, b, w_1, w_2) is obtained instead of w_2. Since the terms of this sequence will be used later, we add E_2' with

$$\alpha_2'(a, b, w_1, w_2) = (a, b, w_1, w_2, a, b),$$

and the block computing $g_3(a, b)$ is modified in a way similar to the above. Thus at A_3^* the sequence

$$(a, b, w_1, w_2, w_3)$$

will be obtained instead of w_3.

Now a and b are not needed anymore. Therefore we add E_3' with

$$\alpha_3'(a, b, w_1, w_2, w_3) = (w_1, w_2, w_3)$$

and (w_1, w_2, w_3) is found as the argument for the block which computes f. Thus we shall obtain at A the value $f(w_1, w_2, w_3)$, of f for the arguments w_1, w_2, w_3, where $w_i = g_i(a, b)$ (for $i = 1, 2, 3$) is the value of g_i for the arguments a, b.

Clearly, an initial function or relation is associated with every vertex.

The proof that computability by a normal scheme is preserved by arbitrary substitutions can be carried out in a similar way.
Now assume that the functions

$$g_0(a) \quad \text{and} \quad g(n, a, w)$$

are computable by the normal schemes represented by the blocks

$$\boxed{\begin{array}{|c|}\hline E_0 \\ \vdots \\ A_0 \\ \hline\end{array}} \quad \text{and} \quad \boxed{\begin{array}{|c|}\hline E_1 \\ \vdots \\ A_1 \\ \hline\end{array}}.$$

We show that the function $f(n, a)$, defined by the primitive recursion

$$\begin{cases} f(0, a) = g_0(a) \\ f(n+1, a) = g(n, a, f(n, a)) \end{cases}$$

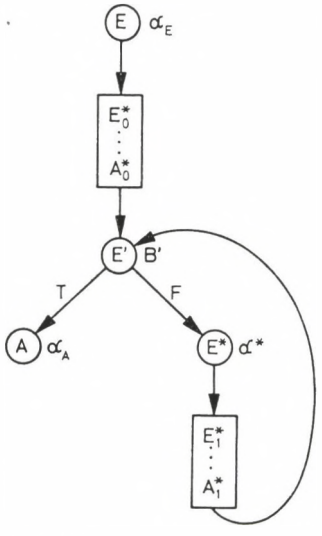
is also definable by a normal scheme.
Let us first pass to the alternative definition, which was given in section 7.5 for the general case: –

$$f(n, a) = h(n, a, 0, g_0(a))$$

with

$$h(n, a, i, w) = \begin{cases} w, & \text{if } i = n \\ h(n, a, i+1, g(i, a, w)) & \text{otherwise.} \end{cases}$$

From this we obtain the following normal scheme, which determines $f(n, a)$: –



This is interpreted as follows: -

The arguments for f are n and a, but we actually want to compute the function h for the arguments

$$n, a, 0, g_0(a).$$

The fourth of these, the value of g_0 at a, has to be computed, while the other three must be preserved. That is why the input vertex E is taken with

$$\alpha_E(n, a) = (n, a, 0, a),$$

and the block computing $g_0(a)$ is modified in that every sequence $(r_1, \ldots, r_s)$ occurring in it as an argument or value has to be replaced by

$$(n, a, 0, r_1, \ldots, r_s).$$

Thus at A_0^*, instead of $w_0 (= g_0(a))$ the sequence

$$(n, a, 0, w_0)$$

is obtained.

Now, the computation of h begins for a four-term sequence (n, a, i, w). We decide whether $i = n$ or not. That is why the relation

$$B'(n, a, i, w) \equiv i = n$$

is associated with the logical vertex E′. If this test fails, the argument (n, a, i, w) is sent along the edge F to the next vertex. Then according to the definition of h we have to compute the value of h for the arguments

$$n, a, i+1, g(i, a, w).$$

This requires the computation of $g(i, a, w)$, while the first three arguments have to be preserved. That is why E* is added with

$$\alpha^*(n, a, i, w) = (n, a, i+1, i, a, w).$$

Moreover the block computing g is modified in that every sequence $(r_1, \ldots, r_s)$ occurring in it has to be replaced by

$$(n, a, i+1, r_1, \ldots, r_s).$$

Therefore, at A_1^* we shall have the sequence

$$(n, a, i+1, w_1)$$

instead of w_1, which is the value of g for the arguments i, a, w.

With this sequence one returns to E′ to check whether or not the new value of its third term has reached the value n. This is repeated until i does increase to the value n, whereupon one passes from E′ along the edge T to the output vertex A, with this vertex, by the definition of h,

$$\alpha_A(n, a, i, w) = w$$

is associated. Clearly, this w is equal to $h(n, a, 0, g_0(a))$, that is to the required function value $f(n, a)$.
This reasoning works the same way for an arbitrary number of parameters. Thus "definability by a normal scheme" is preserved under primitive recursions.
Consequently, *every primitive recursive function can be defined by a normal scheme.*

7.5.2 Reasons behind this Process

This result was made possible because the primitive recursion

$$f(n, a_1, \ldots, a_r) = \begin{cases} g_0(a_1, \ldots, a_r), & \text{if } n = 0 \\ g(n \dot{-} 1, a_1, \ldots, a_r, f(n \dot{-} 1, a_1, \ldots, a_r)) & \text{otherwise,} \end{cases} \tag{7.5.1}$$

from which no graph scheme determining f could be deduced, was replaced by the definition

$$\begin{cases} f(n, a_1, \ldots, a_r) = h(n, a_1, \ldots, a_r, 0, g_0(a_1. \ldots, a_r)) \\ h(n, a_1, \ldots, a_r, i, w) = \begin{cases} w, & \text{if } i = n \\ h(n, a_1, \ldots, a_r, i+1, g(i, a_1, \ldots, a_r, w)) & \text{otherwise.} \end{cases} \end{cases} \tag{7.5.2}$$

From the latter, it is possible to deduce a normal scheme for computing f, and from this (by the result of section 7.3) a non-recursive Algol procedure for the computation of f. On the other hand, from (7.5.1) only a recursive Algol procedure is deducible, which is not immediately understood by a computer.
Thus, from the point of view of programming, definition (7.5.2) is much simpler than (7.5.1). What is the explanation for this?
In any case, the definition of f by (7.5.2) is a particular case of partial recursion, for it can be brought into the form of a defining system of equations by the use of $\mathrm{sg}(n) = \begin{cases} 0 \text{ if } n=0 \\ 1 \text{ otherwise} \end{cases}$ and $\overline{\mathrm{sg}}(n) = \begin{cases} 1 \text{ if } n=0 \\ 0 \text{ otherwise} \end{cases}$ as follows: –

$$\begin{cases} f(n, a_1, \ldots, a_r) = h(n, a_1, \ldots, a_r, 0, g_0(a_1, \ldots, a_r)) \\ h(n, a_1, \ldots, a_r, i, w) = \overline{\mathrm{sg}}(|n-i| \cdot w + \\ \qquad + \mathrm{sg}(|n-i|) \cdot h(n, a_1, \ldots, a_r, i+1, g(i, a_1, \ldots, a_r, w)). \end{cases}$$

For $i=n$, the second term and for $i \neq n$ the first term on the right-hand side of the second expression vanishes, and at the same time the first factor reduces to 1. Clearly this definition yields a more complicated case of gene-

ral recursions than primitive recursion. It is in fact general recursive, although for $i>n$, (7.5.2) does not determine h, since these values of h are not needed for the computation of f. Indeed, f is equal to an h-value with $i=0$. This in turn is equal to an h-value with $i=1$, and so on, until an h-value with $i=n$ is reached. This is obtainable without the application of any further h-value as its last argument. We could have put

$$h(n, a_1, \ldots, a_r, i, w) = w, \quad \text{if} \quad i \geqq n.$$

Then h would be general recursive. In (7.5.2) the value of h is determined at a point

$$(n, a_1, \ldots, a_r, i, w) \quad (\text{for } i < n)$$

by taking an h-value at such a point, where none of the arguments can be considered preceding. Indeed $n, a_1, \ldots, a_r$ remain unchanged, and i is increased by 1. Finally, in place of w we have $g(i, a_1, \ldots, a_r, w)$, which in general is not smaller than w. From the point of view of programming, this mixed definition must still be called primitive.

L. Kalmár conjectured as soon as computers appeared that they might bring changes in our conception of what is "simple" in mathematics. He even thought it possible that in the lowest forms of the future school the teaching of mathematics will start not with the four fundamental arithmetic operations rather with the operations made possible by the computer. By now this conjecture is actually realized in the field of recursive functions.

However independent of programming considerations, we have nothing to show us why the definition scheme (7.5.2) is simpler than (7.5.1). In both, the value of the defined function (f or h) at a given place is obtained with the help of its value at another place. These latter values will be called shortly applied f-value and applied h-value, respectively. The difference is that in (7.5.2) the applied h-value does not occur as the argument of a function, while in (7.5.1) the applied f-value is an argument of g. *This* is the decisive factor. In section 7.1 of this chapter the main point was that g could not even be mentioned. The decision factor is not the way the arguments of the applied f- and h-values were chosen. This contrasts with our earlier notion of primitivity of a recursion, in which the applied f-value had to be taken at the immediately preceding argument.

7.6 The μ-Operations

For every definition by cases, the value of a function h is given at an arbitrary point either independently of h, or as the value of h at another point (where the arguments at this other point are obtainable from the arguments of the original point in an already known way). Then one can always find a non-recursive Algol procedure for computing h. Although such definitions could be called "primitive recursive" with respect to programming, it would not be correct to call primitive recursive those functions which are obtained from the initial functions by substitutions and this new kind of primitive recursions, because the surprising fact is that these functions are *exactly* the partial recursive functions. Indeed, we can show that the μ-operation, by means of which all the partial recursive functions can be obtained from the primitive recursive ones, does not extend the class of all functions which are primitive recursive in this sense.
In fact, if for such a function g

$$f(a_1, \ldots, a_r) = \mu_i[g(i, a_1, \ldots, a_r) = 0],$$

then f can also be defined by the substitution

$$f(a_1, \ldots, a_r) = h(a_1, \ldots, a_r, 0, g(0, a_1, \ldots, a_r)),$$

where h is defined by the new kind of primitive recursion

$$h(a_1, \ldots, a_r, i, w) = \begin{cases} i, & \text{if} \quad w = 0 \\ h(a_1, \ldots, a_r, i+1, g(i+1, a_1, \ldots, a_r)) & \text{otherwise.} \end{cases}$$

For the sake of simplicity, we shall discuss this in the special case

$$f(a) = \mu_i[g(i, a) = 0],$$

since a change in the number of variables does not affect the reasoning at all. In this case the claim is that $f(a)$ can also be defined by

$$f(a) = h(a, 0, g(0, a)),$$

with

$$h(a, i, w) = \begin{cases} i, & \text{if} \quad w = 0 \\ h(a, i+1, g(i+1, a)) & \text{otherwise.} \end{cases}$$

According to the definition, if the third argument of h is 0, then the value of h is equal to its second argument. Therefore, if

$$g(0, a) = 0,$$

then

$$f(a) = h(a, 0, 0) = 0.$$

Otherwise we have

$$f(a) = h(a, 1, g(1, a)),$$

and if here

$$g(1, a) = 0,$$

then

$$f(a) = h(a, 1, 0) = 1.$$

Otherwise we have

$$f(a) = h(a, 2, g(2, a)),$$

and if here

$$g(2, a) = 0,$$

then we have

$$f(a) = h(a, 2, 0) = 2,$$

and so on. Putting these together, we have

if $g(0, a) = 0$, then $f(a) = 0$;

if $g(0, a) \neq 0$, $g(1, a) = 0$, then $f(a) = 1$;

if $g(0, a) \neq 0$, $g(1, a) \neq 0$, $g(2, a) = 0$, then $f(a) = 2$;

etc. Hence $f(a)$ is the smallest i for which $g(i, a)=0$, provided that such an i exists at all. If there is no such i for a, then the computation of $f(a)$ never ends. Consequently $f(a)$ is not defined. This new kind of primitive recursive definition really gives us

$$f(a) = \mu_i[g(i, a) = 0].$$

Thus the simplest recursion with respect to programming is also the most general. It is much more difficult to decide what is simple in mathematics.

7.7 Eliminating Recursion from Algol 60

From the definition

$$\begin{cases} f(a) = h(a, 0, g(0, a)) \\ h(a, i, w) = \begin{cases} i, & \text{if } w = 0 \\ h(a, i+1, g(i+1, a)) & \text{otherwise} \end{cases} \end{cases}$$

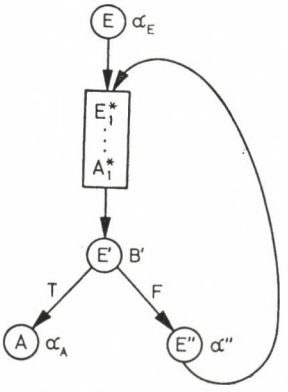
of the function

$$f(a) = \mu_i[g(i, a) = 0]$$

we can deduce a normal scheme for its computation, assuming that we already have a normal scheme determining g. The latter is represented by

the block

It requires a little thought to see that the cases are distinguished here according to whether $w=0$ or not and $w=0$ does not belong to the initial relations. Of course it would be easy to add all these simple relations to the initial ones. I shall choose, however, a different way: the reduction of this relation to the original initial relations. This is achieved by taking a new variable, which is given the value 0 once and for all. Thus we get the following normal scheme: –



Here the given block is modified as follows: – we have to take a as the argument of f, but a new variable v has to be added (for example, as the first term of the sequence of variables), and this has to be given the value 0 at the start. We actually have to compute h for the arguments $a, 0, g(0, a)$. Here first $g(0, a)$ is to be computed, while the others have to be preserved. That is why we have the input vertex E with

$$\alpha_E(a) = (0, a, 0, 0, a),$$

and the block computing $g(i, a)$ has to be modified accordingly. However, one always has to compute new h-values for the arguments

$$a, i, g(i, a)$$

(with i increasing), where first $g(i, a)$ has to be computed while preserving the others. Therefore, the block computing $g(i, a)$ is modified in that every sequence $(r_1, \ldots, r_s)$ occurring at a vertex has to be replaced by

$$(v, a, i, r_1, \ldots, r_s).$$

Thus, in particular, at A_1^* instead of w which is the value of g for the arguments i, a, the sequence

$$(v, a, i, w)$$

is obtained.

Then follows the decision whether or not $w=0$ (that is $w=v$). This is why the relation

$$B'(v, a, i, w) \equiv w = v$$

is associated with the logical vertex E′.

If the test fails, then the same argument is sent along the edge F to E″. Now, by the definition, the function h has to be computed for the arguments

$$a, i+1, g(i+1, a)$$

while v and a have to be preserved. That is why the function

$$\alpha''(v, a, i, w) = (v, a, i+1, i+1, a)$$

is associated with E″. If this value is received by the modified block for the computation of

$$(v, a, i, g(i, a)),$$

then the result obtained at A_1^* will be

$$(v, a, i+1, w),$$

where now $w=g(i+1, a)$. Here it has to be decided again whether or not $w=0$ (that is $w=v$). If not, then everything starts anew with an increasing value of i, until the relation

$$B'(v, a, i, w) \equiv w = v$$

becomes valid. Then the argument is sent along the edge T to A. Here, by the definition of h, one has to pass to its third term. Consequently, we associate with A the function

$$\alpha_A(v, a, i, w) = i.$$

Its value is the smallest i for which

$$w = g(i, a) = 0,$$

if such an i exists at all. Otherwise one never gets out of the cycle. For such an a the graph scheme does not determine any value.

Here, an initial function or relation was associated with every vertex. Thus it follows that the property of being computable by a normal scheme is preserved not only under substitutions and primitive recursions, but also under μ-operations.

Consequently, *every partial recursive numeric function is definable by a normal scheme.*

It was earlier pointed out in the preface that the functioning of a computer can always be considered as the computation of the values of a numeric function. In Ch. 4 it was shown that every machine computable numeric function is partial recursive. Now we have seen that every partial recursive numeric function is definable by a normal scheme. In section 7.3, it was mentioned that a function computable by a normal scheme is also computable by a non-recursive Algol procedure.

From all this we obtain the following result: – *Recursive procedures (including simultaneous ones) can always be eliminated from Algol 60 programs.* A similar result holds for other programming languages too.

Chapter 8

The Epi-language of Algol 60

8.1 Definitions in "Epi-Algol"

As is well known, for the description of the language Algol 60, a certain meta-language is used, which should now rather be called "epi-language", because "meta-language" is used in a different sense with respect to Algol 68. In this language, definitions of the following form occur:

⟨expression⟩ ::= ⟨term⟩|⟨expression⟩ ⟨additive operator⟩ ⟨term⟩.

This notation has come to be known as Backus Normal Form. This is only a part of the definition of ⟨expression⟩ occurring in "Epi-algol". The definition will be complete, if (as I shall do here for the sake of simplicity) one restricts oneself to expressions constructed from natural numbers and scalar variables denoted by small Latin letters, perhaps with numerical indices, by means of addition, arithmetical subtraction, denoted by "−" instead of "$\dot{-}$" and multiplication. Here ⟨expression⟩ means a general and not a concrete expression, and the same applies to ⟨term⟩, which, of course, still has to be defined. The stroke | stands here for "or", and the symbol "::=" has to be read as "is by definition". Hence the sense of the above definition is: "An expression is either a term, or it consists of an expression and a term connected by an additive operator."

Immediately this definition looks circular, since the notion under definition is used in it. The situation, however, is even more complicated. Indeed, in the definition of the notion ⟨term⟩ such further auxiliary notions will be applied, which in turn will be defined using the notion ⟨expression⟩. Therefore it is very important to check that we are not dealing here with senseless definitions, but with recursions.

I shall not use the symbol "|". Instead I shall list one by one the possible cases of a definition. For this purpose I introduce the symbol ":=" to be read as "is by one of the possible definitions".
Now the definition of our (restricted) notion of ⟨expression⟩ reads as follows:

1 ⟨expression⟩ := ⟨term⟩

2 ⟨expression⟩ := ⟨expression⟩ ⟨additive operator⟩ ⟨term⟩

3 ⟨term⟩ := ⟨factor⟩

4 ⟨term⟩ := ⟨term⟩×⟨factor⟩

5 ⟨additive operator⟩ := +

6 ⟨additive operator⟩ := −

7 ⟨factor⟩ := ⟨number⟩

8 ⟨factor⟩ := ⟨variable⟩

9 ⟨factor⟩ := (⟨expression⟩)

10 ⟨number⟩ := ⟨digit⟩

11 ⟨number⟩ := ⟨number⟩ ⟨digit⟩

12 ⟨variable⟩ := ⟨letter⟩

13 ⟨variable⟩ := ⟨variable⟩ ⟨digit⟩

14 ⟨digit⟩ := 0

..... (this should be written out for all the digits)

23 ⟨digit⟩ := 9
24 ⟨letter⟩ := *a*

.....

49 ⟨letter⟩ := *z*.

Here ⟨term⟩ appears in the definition of ⟨expression⟩. In the definition of ⟨term⟩, ⟨factor⟩ appears, and finally, in the definition of ⟨factor⟩, ⟨expression⟩ appears. Hence the circle is closed.
If in the 9th line, no parentheses were used, then the lines

⟨expression⟩ := ⟨term⟩

⟨term⟩ := ⟨factor⟩

⟨factor⟩ := ⟨expression⟩

would indeed form a circle, from which one could not get out, just as with the circle

$$\langle \text{number} \rangle := \langle \text{number} \rangle,$$

which would result if ⟨digit⟩ were omitted in line 11.

8.2 Mathematical Grammars

These problems belong to the field of *mathematical grammars.*

There is a trend in linguistics to define the concept of "grammatically correct sentences" (and other "category concepts") with the same precision with which the concept of the well-formed formulae is defined in mathematics, partly because of the needs of machine translation. This led to the creation of "mathematical grammars" of different kinds. The Epi-Algol rules belong to a particular class of these, known as "phrase structured grammars" [25].

In general, a phrase structured grammar is determined by four non-empty finite sets: –

$$\text{T, H, P, K.}$$

The elements of T (the terminal vocabulary) represent the *terminal concepts*, which stand by themselves in that they are not defined by means of any other notions. Examples of this in Epi-Algol are the digits 0, 1, ..., 9.

The elements of H (the auxiliary vocabulary) represent the *category names*, which in Epi-Algol appear in angled brackets. It is usual to distinguish several special types of these category names which form the set K. In the grammar of Epi-Algol, ⟨expression⟩ can be considered to be one of these special types.

The elements of P, called the *productions,* represent the grammatical rules which are used to "produce" (in other words "generate") the concepts of different categories. In the context-free phrase structured grammars (the case to which we restrict ourselves) these productions define equations in the sense of the "possible equality :=", having the form

$$h := w_1 w_2 \ldots w_n.$$

Here h is a member of H, and every w_i $(i=1, \ldots, n)$ is either an element of T or an element of H (in short $w_i \in T \cup H$). The productions correspond to the lines in our definition of ⟨expression⟩. (It is admissible to have empty productions of the form

$$h := .$$

[25] See for example B. N. Chomsky: *Syntactic Structures* 'S-Gravenhage (1957).

It is then advisable to denote the empty right-hand side by some symbol, for example by $|\_|$, and to add this symbol to the terminal vocabulary.) In particular, every auxiliary concept has to be defined in this way. Hence every element of H is the left-hand side of at least one production.

A chain $w_1 \dots w_n$ with

$$w_1, \dots, w_n \in T \cup H$$

is called a *construction,* and n is called its *order.* We say that a construction ψ is "directly generated" by a construction φ, if ψ results from φ by the "application of a production belonging to P". More precisely: φ has the form $v_1 \dots v_m$, P contains a production

$$v_i := w_1 \dots w_n$$

for some $i=1, \dots, m$, and ψ has the form

$$v_1 \dots v_{i-1} w_1 \dots w_n v_{i+1} \dots v_m .$$

We say that ψ is "generated" by φ, if there is a generating sequence

$$\varphi_1 = \varphi; \varphi_2; \dots; \varphi_r = \psi \tag{8.2.1}$$

of constructions, in which φ_i is directly generated by φ_{i-1} for every $i= =2, \dots, r$. A construction is called terminal if only members of T occur in it.

Now the exact meaning of the statement "a construction φ belongs to the category denoted by the name h" ($h \in H$) is as follows: φ is terminal and is generated by h. In this sense, it is said that a terminal construction of the Epi-Algol language (that is one consisting of digits, letters, parentheses and operation symbols) is an expression, factor or term, provided that it can be generated by ⟨expression⟩, ⟨factor⟩, or ⟨term⟩, respectively, with the given Epi-Algol rules as productions.

A natural problem is to decide whether a given terminal construction belongs to a certain category or not. The solution of this problem follows from the next result [26]: – If the category names (auxiliary concepts) of a phrase structured grammar are defined without circularity in such a way that they do not generate themselves (as is true in Epi-Algol), then *the property "to belong to a category" is primitive recursive in the word set over the terminal vocabulary* T *as alphabet.*

[26] R. Péter: *Über die Rekursivität der Begriffe der mathematischen Grammatiken,* Publ. Math. Inst. of the Hung. Acad. of Sci. **8** (1963) pp. 214–228.

8.3 Eliminating Circularity

First the required "freedom from circularity" has to be examined more closely.
In any case, one has to require that no sequence could be formed from the members of P of the form: –

$$h_1 := h_2$$
$$h_2 := h_3$$
$$\ldots\ldots\ldots$$
$$h_r := h_1.$$

Of course, for $r=1$, we exclude a production of the form $h_1 := h_1$. It can be shown that in Epi-Algol this requirement is satisfied.
If this requirement is satisfied and l is the number of elements of H, then for $r \geqq l$ we cannot have a sequence of the form

$$h_1 := h_2$$
$$h_2 := h_3$$
$$\ldots\ldots\ldots$$
$$h_r := h_{r+1}$$

from members of P even if $h_{r+1} \neq h_r$, since otherwise at least two terms of the sequence

$$h_1, \ldots, h_{r+1}$$

would coincide. If we had

$$h_i = h_{i+j}$$

for some $i < r+1$ and $0 < j \leqq r+1-i$, then the subsequence

$$h_i := h_{i+1}$$
$$h_{i+1} := h_{i+2}$$
$$\ldots\ldots\ldots\ldots$$
$$h_{i+j-1} := h_{i+j}$$

would generate h_i by itself.
Sequences of productions of the form

$$h_1 := h_2$$
$$h_2 := h_3$$
$$\ldots\ldots\ldots$$
$$h_r := h_{r+1},$$

where $h_1, \ldots, h_{r+1}$ are arbitrary elements of H, will be called "dangerous" for any $r \geqq 1$.

The freedom from circularity enables us to replace P by a production set P′, containing no "dangerous productions" at all, that is no productions of the form

$$h_1 := h_2 \quad (h_1, h_2 \in \mathrm{H}),$$

while the elements of H generate the same terminal constructions through the productions of P′ as through those of P.

The transition from P to P′ can be carried out in (at most) l steps. In the first step we form a production set P_1 out of $\mathrm{P} = \mathrm{P}_0$ in such a way that every element of P of the form

$$h_1 := h_2 \quad (h_1, h_2 \in \mathrm{H})$$

is replaced by those productions in which a construction directly generated by h_2 is put instead of h_2. Such constructions must exist, since h_2 is the left-hand side of at least one production.

Now, if the construction sequence

$$\varphi_1 = h_1;\ \varphi_2 = h_2;\ \varphi_3;\ \ldots;\ \varphi_t \tag{8.2.2}$$

generates through P the terminal construction φ_t by h_1, then φ_3 is directly generated by h_2, hence

$$h_1 := \varphi_3$$

was added to P_1. Therefore, the construction sequence

$$\varphi_1 = h_1;\ \varphi_3;\ \ldots;\ \varphi_t \tag{8.2.3}$$

generates the same φ_t through P_1. Conversely, if

$$h_1 := \varphi_3$$

is a new production added to P_1 because

$$h_1 := h_2$$

appeared in P, and the construction sequence (8.2.3) generates φ_t through P_1, then the sequence (8.2.2) generates the same φ_t through P.

If P_1 still contains dangerous productions, then P_2 is formed out of P_1 in the same way as P_1 was obtained from P_0, and so on.

Now if in P_i, for $i \geqq 1$, a dangerous production of the form

$$h_1 := h_2$$

appears, this can only happen if, for $i \geqq 1$, in P_{i-1}, there are dangerous productions

$$h_1 := h_3 \quad \text{and} \quad h_3 := h_2$$

forming a dangerous sequence of length two. If $i \geqq 2$, these can exist only if in P_{i-2} there are dangerous productions

$$h_1 := h_4, \quad h_4 := h_3$$

and

$$h_3 := h_5, \quad h_5 := h_2,$$

which together form a dangerous sequence of length four. In general, for every $i = 1, 2, \ldots$ we get that, if P_i contains a dangerous production, then $P = P_0$ contains a dangerous sequence of length 2^i. However, this is not possible for $2^i \geqq l$; therefore, if i is the smallest number with $2^i \geqq l$, then P_i contains no more dangerous productions. Hence we can put

$$P' = P_i,$$

since we have proved that the steps of transition from P to P′ leave unchanged the set of terminal constructions generated by elements of H.
The precaution we took was somewhat exaggerated. A production of the form

$$h_1 := h_2 \quad (h_1, h_2 \in H),$$

in which the constructions directly generated by h_2 are terminal, surely does not involve any danger. Hence these need not be eliminated. I did not want to interrupt the reasoning with this point.

8.4 An Example

Let us consider as an example, the sublanguage of Epi-Algol by means of which the category name ⟨expression⟩ was defined in section 8.1. In this case the terminal vocabulary T consists of the digits, letters, parentheses, and operation symbols. The auxiliary vocabulary H contains the category names

⟨expression⟩, ⟨term⟩, ⟨additive operator⟩,

⟨factor⟩, ⟨number⟩, ⟨variable⟩

⟨digit⟩, ⟨letter⟩;

Finally, P contains the productions used in the definition of ⟨expression⟩. Among these the following are dangerous:

⟨expression⟩ := ⟨term⟩

⟨term⟩ := ⟨factor⟩

⟨factor⟩ := ⟨number⟩

⟨factor⟩ := ⟨variable⟩.

At first glance the productions

$$\langle \text{number} \rangle := \langle \text{digit} \rangle$$

$$\langle \text{variable} \rangle := \langle \text{letter} \rangle$$

also look dangerous. In fact these are harmless, since ⟨digit⟩ generates directly only the terminal constructions

$$0, 1, \ldots, 9,$$

while ⟨letter⟩ generates only the terminal constructions

$$a, b, \ldots, z.$$

The really dangerous productions in the first step of the method described in the previous section are amended as follows: –

replace ⟨expression⟩ := ⟨term⟩

by

⟨expression⟩ := ⟨factor⟩

⟨expression⟩ := ⟨term⟩×⟨factor⟩;

then

⟨term⟩ := ⟨factor⟩

by

⟨term⟩ := ⟨number⟩

⟨term⟩ := ⟨variable⟩

⟨term⟩ := (⟨expression⟩);

then

⟨factor⟩ := ⟨number⟩

by

⟨factor⟩ := ⟨digit⟩

⟨factor⟩ := ⟨number⟩ ⟨digit⟩,

finally replace

⟨factor⟩ := ⟨variable⟩

by

⟨factor⟩ := ⟨letter⟩

⟨factor⟩ := ⟨variable⟩ ⟨digit⟩.

Among these new productions three are really dangerous. In the second step they will be replaced by new productions as follows: –

replace ⟨expression⟩ := ⟨factor⟩

by

⟨expression⟩ := (⟨expression⟩)
⟨expression⟩ := ⟨digit⟩
⟨expression⟩ := ⟨number⟩ ⟨digit⟩
⟨expression⟩ := ⟨letter⟩
⟨expression⟩ := ⟨variable⟩ ⟨digit⟩;

then replace

⟨term⟩ := ⟨number⟩

by

⟨term⟩ := ⟨digit⟩
⟨term⟩ := ⟨number⟩ ⟨digit⟩;

finally replace

⟨term⟩ := ⟨variable⟩

by

⟨term⟩ := ⟨letter⟩
⟨term⟩ := ⟨variable⟩ ⟨digit⟩.

Among these productions there are not any dangerous ones anymore. The elements of the production set P′ are therefore the following: –

⟨expression⟩ := ⟨expression⟩ ⟨additive operator⟩ ⟨term⟩
⟨expression⟩ := ⟨term⟩×⟨factor⟩
⟨expression⟩ := (⟨expression⟩)
⟨expression⟩ := ⟨digit⟩
⟨expression⟩ := ⟨number⟩ ⟨digit⟩
⟨expression⟩ := ⟨letter⟩
⟨expression⟩ := ⟨variable⟩ ⟨digit⟩
⟨term⟩ := ⟨term⟩×⟨factor⟩
⟨term⟩ := (⟨expression⟩)
⟨term⟩ := ⟨digit⟩
⟨term⟩ := ⟨number⟩ ⟨digit⟩
⟨term⟩ := ⟨letter⟩
⟨term⟩ := ⟨variable⟩ ⟨digit⟩
⟨additive operator⟩ := +
⟨additive operator⟩ := –
⟨factor⟩ := (⟨expression⟩)

⟨factor⟩ := ⟨digit⟩
⟨factor⟩ := ⟨number⟩ ⟨digit⟩
⟨factor⟩ := ⟨letter⟩
⟨factor⟩ := ⟨variable⟩ ⟨digit⟩
⟨number⟩ := ⟨digit⟩
⟨number⟩ := ⟨number⟩ ⟨digit⟩
⟨variable⟩ := ⟨letter⟩
⟨variable⟩ := ⟨digit⟩ ⟨variable⟩
⟨digit⟩ := 0
.....
⟨digit⟩ := 9
⟨letter⟩ := *a*
.....
⟨letter⟩ := *z*.

8.5 Primitive Recursion in Epi-Algol 60

Now we carry on our reasoning on this example to show that the property "to be a terminal construction generated by a given element of H" is primitive recursive.

Let M be the word set over the alphabet T, and t_1 be a fixed letter of this alphabet. Let us denote the characteristic functions of the properties: –

"to be an expression, an additive operator,
a term, a factor, a digit,
a number, a letter, a variable",

in this order, by

ex, ao, te, fa, di, nu, le, va

respectively.
Several of these can be shown to be primitive recursive very easily: –

$$\text{ao}(x) = \begin{cases} \wedge, & \text{if} \quad (x=+)\vee(x=-) \\ t_1 & \text{otherwise,} \end{cases}$$

$$\text{di}(x) = \begin{cases} \wedge, & \text{if} \quad x=0\vee x=1\vee\ldots\vee x=9 \\ t_1 & \text{otherwise,} \end{cases}$$

$$\text{le}(X) = \begin{cases} \wedge, & \text{if} \quad X=a\vee X=b\vee\ldots\vee X=z \\ t_1 & \text{otherwise} \end{cases}$$

(where the argument is denoted by capital X since x is also a letter).

It can be seen that every argument for which

$$\mathrm{ao}(x) = \Lambda, \quad \text{or} \quad \mathrm{di}(x) = \Lambda, \quad \text{or} \quad \mathrm{le}(X) = \Lambda$$

holds, must be a member of the alphabet T and thus cannot be equal to Λ. Therefore $\mathrm{nu}(x)$ and $\mathrm{va}(x)$ can be defined as primitive recursive functions in the following way:

$$\mathrm{nu}(x) = \begin{cases} \Lambda, & \text{if} \quad \mathrm{di}(x) = \Lambda \vee (\mathrm{nu}(\mathrm{at}(x)) = \Lambda \,\&\, \mathrm{di}(\mathrm{lb}(x)) = \Lambda) \\ t_1 & \text{otherwise,} \end{cases}$$

$$\mathrm{va}(x) = \begin{cases} \Lambda, & \text{if} \quad \mathrm{le}(x) = \Lambda \vee (\mathrm{va}(\mathrm{at}(x)) = \Lambda \,\&\, \mathrm{di}(\mathrm{lb}(x)) = \Lambda) \\ t_1 & \text{otherwise.} \end{cases}$$

As functions which are already known, these can be applied in the definitions of $\mathrm{ex}(x)$, $\mathrm{te}(x)$ and $\mathrm{fa}(x)$.

Here all the connected pieces of x must be considered as predecessors of x, not only its initial segments. Let $y \preceq x$ denote that y is a predecessor of x in this wider sense. Now the definitions of the above functions read as follows:

$$\mathrm{ex}(x) = \begin{cases} \Lambda, & \text{if } (\mathrm{E}y_1)(\mathrm{E}y_2)(\mathrm{E}y_3)[y_1, y_2, y_3 \preceq x \,\&\, \mathrm{ex}(y_1) = \Lambda \,\&\, \mathrm{ao}(y_2) = \Lambda \,\& \\ & \,\&\, \mathrm{te}(y_3) = \Lambda \,\&\, x = y_1 y_2 y_3] \vee \\ & \vee (\mathrm{E}y_1)(\mathrm{E}y_2)[y_1, y_2 \preceq x \,\&\, \mathrm{te}(y_1) = \Lambda \,\&\, \mathrm{fa}(y_2) = \Lambda \,\& \\ & \,\&\, x = y_1 \times y_2] \vee \\ & \vee (\mathrm{E}y)[y \preceq x \,\&\, \mathrm{ex}(y) = \Lambda \,\&\, x = (y)] \vee \\ & \vee \mathrm{di}(x) = \Lambda \vee (\mathrm{nu}(\mathrm{at}(x)) = \Lambda \,\&\, \mathrm{di}(\mathrm{lb}(x)) = \Lambda) \vee \\ & \vee \mathrm{le}(x) = \Lambda \vee (\mathrm{va}(\mathrm{at}(x)) = \Lambda \,\&\, \mathrm{di}(\mathrm{lb}(x)) = \Lambda) \\ t_1 & \text{otherwise.} \end{cases}$$

Here we used shorter notations, for example,

$(\mathrm{E}y_1)(\mathrm{E}y_2)[y_1, y_2 \preceq x \,\&\, \ldots]$ instead of $(\mathrm{E}y_1)[y_1 \preceq x \,\&\, (\mathrm{E}y_2)[y_2 \preceq x \,\&\, \ldots]]$.

Moreover

$$\mathrm{te}(x) = \begin{cases} \Lambda, & \text{if} \quad (\mathrm{E}y_1)(\mathrm{E}y_2)[y_1, y_2 \preceq x \,\&\, \mathrm{te}(y_1) = \Lambda \,\&\, \mathrm{fa}(y_2) = \Lambda \,\&\, x = \\ & = y_1 \times y_2] \vee (\mathrm{E}y)[y \preceq x \,\&\, \mathrm{ex}(y) = \Lambda \,\&\, x = (y)] \vee \\ & \vee \mathrm{di}(x) = \Lambda \vee (\mathrm{nu}(\mathrm{at}(x)) = \Lambda \,\&\, \mathrm{di}(\mathrm{lb}(x)) = \Lambda) \vee \\ & \vee \mathrm{le}(x) = \Lambda \vee (\mathrm{va}(\mathrm{at}(x)) = \Lambda \,\&\, \mathrm{di}(\mathrm{lb}(x)) = \Lambda) \\ t_1 & \text{otherwise,} \end{cases}$$

$$\mathrm{fa}(x) = \begin{cases} \Lambda, & \text{if} \quad (\mathrm{E}y)[y \preceq x \,\&\, \mathrm{ex}(u) = \Lambda \,\&\, x = (y)] \vee \\ & \vee \mathrm{di}(x) = \Lambda \vee (\mathrm{nu}(\mathrm{at}(x)) = \Lambda \,\&\, \mathrm{di}(\mathrm{lb}(x)) = \Lambda) \vee \\ & \vee \mathrm{le}(x) = \Lambda \vee (\mathrm{va}(\mathrm{at}(x)) = \Lambda \,\&\, \mathrm{di}(\mathrm{lb}(x)) = \Lambda) \\ t_1 & \text{otherwise.} \end{cases}$$

8.6 Predecessors in Algol 60

The values of the functions

$$\mathrm{ex}(x), \quad \mathrm{te}(x), \quad \mathrm{fa}(x)$$

are defined here by using values of the same function, as well as of the other two at properly preceding places. Among these predecessors not only initial segments occur, but also predecessors in the wider sense (as for example y_1, y_2 and y_3 in the first alternative with $\mathrm{ex}(x) \doteq \Lambda$). Even these are not necessarily immediate predecessors. Hence we are dealing here with a simultaneous course-of-values recursion. In an earlier paper [27] of mine, I have shown that such a definition can be reduced to course-of-values recursion of the separate functions to be defined. These, in turn, can be reduced to primitive recursions with the help of substitutions. All this, of course, is meant with the extended notion of predecessor. In what follows, I shall continue to use this extended notion of predecessor. For this application, this extension of predecessor offers itself as a natural notion, but the more restricted notation which we used so far is more convenient to work with. As was pointed out in section (3.3.2), on the method of coding, in number theory, the above definitions can be transformed into recursive definitons of the same type with the earlier notion of predecessor. Since the reasoning we applied in this particular example can be extended to the general case, we can obtain *primitive recursive definitions of the properties "to be in a category of a phrase structured grammar"*. In our example these properties are "to be an expression, term, factor" respectively.

The value of a primitive recursive function, however, can be computed at every argument in a finite number of steps. Consequently, a method must exist to decide whether or not an arbitrary chain of terminal elements is one case of a notion introduced in a phrase structured grammar (for example, in Epi-Algol).

[27] R. Péter: *Primitive-rekursive Wortbeziehungen in der Programmierungssprache "Algol 60"*, Publ. Math. Inst. of the Hung. Acad. of Sci. **6** (1961) pp. 137–144.

Chapter 9

Two-level Grammar in Algol 68

9.1 An Auxiliary Theorem

Let us consider an example of a construction sequence of the type (8.1.1), generating a terminal expression by the category name ⟨expression⟩ through the productions belonging to P′ of section 8.3. In this I shall use obvious abbreviations, like ⟨ex⟩ for ⟨expression⟩, ⟨adop⟩ for ⟨additive operator⟩, and so on: –

⟨ex⟩; (⟨ex⟩); (⟨te⟩×⟨fa⟩);

(⟨ex⟩)×⟨fa⟩); ((⟨ex⟩ ⟨adop⟩ ⟨te⟩)×⟨fa⟩);

((⟨di⟩ ⟨adop⟩ ⟨te⟩)×⟨fa⟩);

((1 ⟨adop⟩ ⟨te⟩)×⟨fa⟩); ((1+⟨te⟩)×⟨fa⟩);

((1+⟨va⟩ ⟨di⟩)×⟨fa⟩);

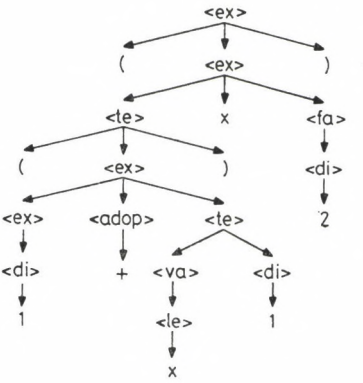
((1+⟨le⟩ ⟨di⟩)×⟨fa⟩); ((1+*x*⟨di⟩)×⟨fa⟩);

((1+*x*1)×⟨fa⟩); ((1+*x*1)×⟨di⟩);

((1+*x*1)×2).

The outer parentheses are used as a precaution: the expression obtained might have to be used further on.

The structure of this becomes more apparent if, from every auxiliary concept to which a production was applied, we draw edges pointing to the elements of the result, as can be seen on the following graph: –

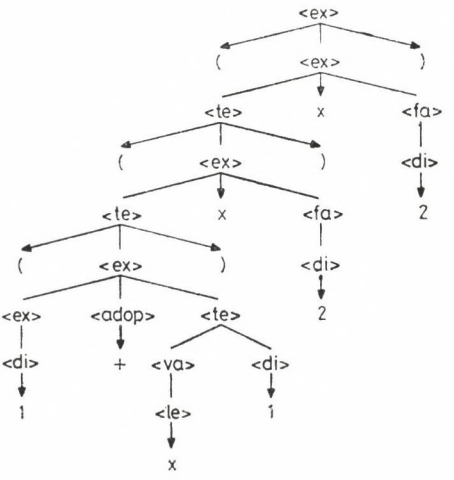


To every end point a single path leads from the initial point. If these paths are considered one after the other, from the left to the right, then their end points yield the following terminal expression: –

$$((1+x1)\times 2).$$

Along some of these paths, several instances of the same auxiliary concept can be found. For example, along the path leading to the first digit 1, ⟨ex⟩ occurs four times. This is not a coincidence. Since there are only a finite number of auxiliary concepts, it can be seen that such repetitions must occur on at least one of the paths leading from the initial point to the end points, provided that the graph represents the generation of a sufficiently long expression. I shall not go into the proof here.

Let us examine the resulting expression if, in the above-mentioned path, we apply to the third occurrence of ⟨ex⟩ the same production as we did to its second occurrence. Then the same generating steps are applied to the result as above: –



The terminal expression deducible from this is

$$(((\mathbf{1}+\mathbf{x1})\times 2)\times 2).$$

Let us compare this with the original

$$((\mathbf{1}+\mathbf{x1})\times 2).$$

The part $\mathbf{1}+\mathbf{x1}$ (printed boldface) occurs in both. Here it is generated by the fourth ⟨ex⟩, in the original it was generated by the third ⟨ex⟩. Apart from this part, the things generated by the second ⟨ex⟩ in the first graph, that is "("and "(×2" were doubled, while the remaining parts "(" and ")" were left unchanged.

It can be shown through a similar representation that, in general, to every language S, generated by a context-free phrase structured grammar (that is not containing any dangerous productions in the sense of section 8.3), there exists a natural number q such that every terminal construction belonging to a category of S, and consisting of at least q letters, can be written in the form

$$\alpha_1\beta_1\alpha_2\beta_2\alpha_3.$$

Here among the (empty or non-empty) subchains

$$\alpha_1, \beta_1, \alpha_2, \beta_2, \alpha_3$$

at least one of β_1 and β_2 is not empty. Moreover

$$\alpha_1\beta_1\beta_1\alpha_2\beta_2\beta_2\alpha_3$$

also belongs to S. I will refer to this result as the *Bar-Hillel–Perles–Shamir theorem* [28].

9.2 Two-level Phrase Structured Grammars

The grammar of the more recent programming language Algol 68 is a phrase structured grammar in a generalized sense, in that the corresponding production set is infinite, while the set of separate category names remains finite. To specify the infinite production set one uses a metalanguage. It was because of this terminology that I used earlier the term

[28] J. Bar-Hillel, C. Gaifman, E. Schamir: *On formal properties of simple phrase structure grammars,* Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung **14** (1961) pp. 143–172. The above theorem is merely a particular case of a more general theorem in this paper.

"Epi-Algol", instead of "Meta-Algol". Epi-Algol is generated by a phrase structured grammar in the original sense.

More precisely, a *two-level phrase structured grammar* is determined by five finite, non-empty sets

$$Z, M, P, V, K,$$

where

$$Z, M, P$$

denote the vocabulary, the auxiliary vocabulary and the production set of the meta-language. Their elements will be called *symbols, meta-symbols* and *meta-productions,* respectively. Furthermore, the elements of V and K are called *preproductions* and *category names,* respectively. By means of the first three sets one builds the infinite number of productions of the second level grammar, from which the terminal and auxiliary vocabularies of the second level are also obtained. The last two are the separate auxiliary concepts of the second level.

For the exact definition we introduce certain modifications in the notation, which will save us the use of angled brackets. The necessary separation of *symbol* sequences at the second level is done by commas, and the terminal concepts on the second level are distinguished from the auxiliary concepts in that they do not occur as left-hand sides of productions.

That is why in what follows, I shall use three different words for finite sequences: – "chain", "list", "sequence", according to whether the elements are respectively simply put one after each other, or separated by commas, or separated by semicolons. From the elements of Z one builds *chains,* and from these *symbol-chain lists.* Furthermore, from the elements of Z∪M *mixed chains* are built and from these *mixed chain lists* will be formed.

The elements of P (the meta-productions) have the form

$$m :: v,$$

where $m \in$ M and v is a mixed chain.

A terminal expression generated by a metasymbol $m \in$ M is simply called "a value of m". The productions of second level will be obtained by substituting such values into preproductions. More precisely: every element of V, (that is every preproduction), has the form

$$\vartheta :: \Theta,$$

where ϑ is a mixed chain and Θ is a mixed chain list. For example,

$$z_1 m_1 m_2 :: z_2 m_1 z_3 m_1, \quad z_1 m_3,$$

with

$$z_1, z_2, z_3 \in Z \quad \text{and} \quad m_1, m_2, m_3 \in M.$$

A *production* is obtained if in a preproduction every occurring meta-symbol (in our example m_1, m_2 and m_3) is replaced by one of its values, wherever it occurs. In our example m_1 occurs on both sides. Consequently, the left-hand side of a production is a symbol chain, while its right-hand side is a symbol-chain list.

The left-hand sides of the productions (of which a large finite number form the set K of distinct category names) are called *potential category names,* or more shortly *auxiliary concepts.* They constitute the auxiliary vocabulary at the second level. Those terms of the lists standing on the right-hand sides of productions, which do not occur as left-hand sides are called *terminal concepts.* They form the terminal vocabulary.

The original definitions can be transfered to the new notation in a natural way. For example, a symbol chain list Θ_2 is directly generated by the symbol chain list Θ_1 if it is obtainable from Θ_1 by means of a production, or more precisely if Θ_2 is obtained by replacing one auxiliary concept-term ϑ of Θ_1 by the right-hand list of a production whose left-hand side is ϑ. A symbol chain list Θ_r is generated by Θ_1 if there is a generating sequence

$$\Theta_1, \Theta_2, \ldots, \Theta_r \tag{9.2.1}$$

in which Θ_i is directly generated by Θ_{i-1} for every $i=2, \ldots, r$. A symbol-chain list is called terminal if each of its terms is a terminal concept. The terminal concepts generated by a category name (that is by an element of K) constitute the category designated by this name. By "the language generated in two levels", we mean the correspondence between the categories and their names.

9.3 An Example of a Two-level Language

Since the set K of category names is finite, the question arises whether a language generated at two levels could also be defined by means of finitely many productions, that is, at one level through a simple phrase structured grammar in the original sense.

This can be refuted by the following very simple counter-example [29]. Let us consider the two-level grammar determined by the sets

$$\mathrm{Z} = \{z_1; z_2; z_3\}, \quad \mathrm{M} = \{m\}, \quad \mathrm{P} = \{m := z_1 m z_1; m := z_2\},$$
$$\mathrm{V} = \{z_3 := m z_2 m\}, \quad \mathrm{K} = \{z_3\}.$$

[29] See R. Péter: *Zur zweistufigen Satzstruktur-Grammatik II.* Studia Sci. Math. Hung. **3** (1968) pp. 181–194.

Here the single meta-symbol m occurs on the right-hand side of the first meta-production only. Hence both meta-productions have to apply directly to this, resulting in

$$z_1 z_1 m z_1 z_1 \quad \text{and} \quad z_1 z_2 z_1,$$

where the second is already terminal, that is a value of m. Now, applying both meta-productions to the first, we obtain

$$z_1 z_1 z_1 m z_1 z_1 z_1 \quad \text{and} \quad z_1 z_1 z_2 z_1 z_1,$$

where the latter is again a value of m, and so on. We can thus see that all the values of the single meta-symbol m are as follows: –

$$\underbrace{z_1 z_1 \ldots z_1}_{n\text{-times}} z_2 \underbrace{z_1 z_1 \ldots z_1}_{n\text{-times}}, \quad (\text{for } n = 0, 1, 2, \ldots).$$

Substituting these values of m into the single preproduction, we obtain all the terminal constructions generated by the single category name z_3, in the form

$$\underbrace{z_1 \ldots z_1}_{n} z_2 \underbrace{z_1 \ldots z_1}_{n} z_2 \underbrace{z_1 \ldots z_1}_{n} z_2 \underbrace{z_1 \ldots z_1}_{n} \quad \text{for} \quad n = 0, 1, 2, \ldots.$$

If this language could be generated by a one-level phrase structured grammar, then according to the *Bar-Hillel–Perles–Shamir* theorem (quoted in section 9.1), for large enough n, the terminal construction consisting of $4n+3$ symbols and generated by z_3 could be written in the form

$$\underbrace{z_1 \ldots z_1}_{n} z_2 \underbrace{z_1 \ldots z_1}_{n} z_2 \underbrace{z_1 \ldots z_1}_{n} z_2 \underbrace{z_1 \ldots z_1}_{n} = \alpha_1 \beta_1 \alpha_2 \beta_2 \alpha_3.$$

Hence for $n' > n$ we should have

$$\underbrace{z_1 \ldots z_1}_{n'} z_2 \underbrace{z_1 \ldots z_1}_{n'} z_2 \underbrace{z_1 \ldots z_1}_{n'} z_2 \underbrace{z_1 \ldots z_1}_{n'} = \alpha_1 \beta_1 \beta_1 \alpha_2 \beta_2 \beta_2 \alpha_3$$

where at least one of the subchains β_1, β_2 is not empty. In both left-hand sides here, z_2 occurs exactly three times. However the doubling of β_1 and β_2 would increase the number of occurrences of z_2 if one of these happened to contain z_2. Therefore, both β_1 and β_2 must be those parts of subchains of order n of our first symbol chain, which contain the symbol z_1 only. The doubling of β_1 and β_2 therefore increases the number of occurrences of the symbol z_1 in at least one of these subchains, but in at most two subchains, while in the remaining two such subchains this number stays at n. Hence it cannot increase to n'.

Consequently, this language generated at two levels, cannot be generated at a single level.

9.3.1 The Primitive Recursivity of a Language

The counter-example is a very simple language. If z_1 and z_2 were abbreviations for the words "sweet" and "Mary", respectively, then we could say that this is the language of enchanted admirers of Mary, who can utter only such sighs as: –

Mary Mary Mary

sweet Mary sweet Mary sweet Mary sweet

sweet sweet Mary sweet sweet Mary sweet sweet Mary sweet sweet

. . . . . .

and so on indefinitely.

Membership of this language can be defined as a primitive recursive relation in the word set over the alphabet consisting of z_1 and z_2 as follows (choosing z_1 as the fixed element of the alphabet): –

Let us denote by $f_{z_1}(x)$ and $\mathrm{sm}(x)$ the characteristic functions of the properties "to consist solely of z_1", including the case "to be empty", and "to belong to the above language" (the "sweet Mary" language), respectively. The first of these can be defined as

$$f_{z_1}(x) = \begin{cases} \wedge, & \text{if} \quad x = \wedge \vee (f_{z_1}(\mathrm{at}\,(x)) = \wedge \,\&\, \mathrm{lb}\,(x) = z_1) \\ z_1 & \text{otherwise.} \end{cases}$$

This can be transformed into a normal primitive recursion. The same applies to many of the following definitions. The second is defined by the following definition-by-cases: –

$$\mathrm{sm}\,(x) = \begin{cases} \wedge, & \text{if} \quad (\mathrm{E}y)[y \preceq x \,\&\, f_{z_1}(y) = \wedge \,\&\, x = yz_2yz_2yz_2y] \\ z_1 & \text{otherwise.} \end{cases}$$

9.4 The General Question

What about the recursivity in general of a language generated at two levels [30] by the finite sets

Z, M, P, V, K

in the manner described in section 9.2?

This requires recursive definitions of the characteristic functions $k_i(x)$ of the properties "to belong to a category of the language" (that is, to be a

[30] See R. Péter: *Zur Frage der Rekursivität der im „Algol 68" verwendeten zweistufigen Grammatik*, Ann. Univ. Sci. Budapest **15** (1972) pp. 89–101.

terminal expression generated by an element k_i of K), in the word set W over the alphabet Z′ containing in addition to the elements of Z the auxiliary symbols ":=", "," and ";".
More precisely, an $x \in W$ satisfies this property if it is a terminal symbol chain list which is the last term of a generating sequence of symbol chain lists of the type (9.2.1), beginning with a $k_i \in$ K.
Therefore, we first have to study the recursivity of the notions "sequence of symbol chain lists", "last term", "terminal" and "to generate".

9.5 Recursivity in Symbol Chains

Of the following natural definitions it is not always immediately obvious that they determine primitive recursive word functions. Nevertheless every one of them can be reduced to primitive recursions and substitutions.
Let the elements of Z (the symbols) be

$$z_1, z_2, \ldots, z_t$$

and let z_1 be the fixed elements of our alphabet Z'. Then the characteristic functions

$$\mathrm{z}(x), \quad \mathrm{zk}(x), \quad \mathrm{zkl}(x), \quad \mathrm{zklf}(x)$$

of the properties

"to be a symbol, a symbol chain,
a symbol chain list,
a sequence of symbol chain lists", respectively

can be determined by means of the following definitions (leading to course-of-values recursions): –

$$\mathrm{z}(x) = \begin{cases} \wedge, & \text{if} \quad x = z_1 \vee \ldots \vee x = z_t \\ z_1 & \text{otherwise,} \end{cases}$$

$$\mathrm{zk}(x) = \begin{cases} \wedge, & \text{if} \quad \mathrm{z}(x) = \wedge \vee (\mathrm{E}y_1)(\mathrm{E}y_2)[y_1, y_2 \preceq x \,\&\, \mathrm{zk}(y_1) = \wedge \,\& \\ & \qquad \& \,\mathrm{z}(y_2) = \wedge \,\&\, x = y_1\, y_2] \\ z_1 & \text{otherwise,} \end{cases}$$

$$\mathrm{zkl}(x) = \begin{cases} \wedge, & \text{if} \quad \mathrm{zk}(x) = \wedge \vee (\mathrm{E}y_1)(\mathrm{E}y_2)[y_1, y_2 \preceq x \,\&\, \mathrm{zkl}(y_1) = \wedge \,\& \\ & \qquad \& \,\mathrm{zk}(y_2) = \wedge \,\&\, x = y_1, y_2] \\ z_1 & \text{otherwise,} \end{cases}$$

$$\mathrm{zklf}(x) = \begin{cases} \wedge, & \text{if} \quad \mathrm{zkl}(x) = \wedge \vee (\mathrm{E}y_1)(\mathrm{E}y_2)[y_1, y_2 \preceq x \,\&\, \mathrm{zklf}(y_1) = \wedge \,\& \\ & \qquad \& \,\mathrm{zkl}(y_2) = \wedge \,\&\, x = y_1;\, y_2] \\ z_1 & \text{otherwise.} \end{cases}$$

Here $\mathrm{zk}(x)$ was defined in the same way as the succeeding ones only for the sake of homogeneity. It could have been defined more simply.
Similarly, the symbols, the symbol chain, the symbol chain lists and the sequences of symbol chain lists can be arranged into primitive recursive sequences. For this we use the theorem [31], denoted by (HB), which says that two primitive recursive numeric functions

$$\sigma_1(n) \quad \text{and} \quad \sigma_2(n)$$

can be defined with the following property: – the pairs of natural numbers can be arranged in a sequence in such a way that the nth term of this sequence is the pair

$$(\sigma_1(n), \sigma_2(n)).$$

For the sake of homogeneity, let us arrange our large finite number of symbols into the infinite sequence

$$z_1, z_2, \ldots, z_t, z_t, z_t, \ldots .$$

Since the natural numbers are represented here by

$$\wedge, z_1, z_1 z_1, \ldots ,$$

this sequence can be defined as

$$z_{o(x)} = \begin{cases} \wedge, & \text{if} \quad x = \wedge \\ z_1, & \text{if} \quad o(x) = z_1 \\ z_2, & \text{if} \quad o(x) = z_1 z_1 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ z_t, & \text{if} \quad o(x) \geq \underbrace{z_1 z_1 \ldots z_1}_{t\text{-times}}. \end{cases}$$

It is a general fact that finite sequences consisting of the terms of a primitive recursive sequence $v_{o(x)}$ can themselves be arranged in a primitive recursive sequence $w_{o(x)}$. This can be seen as follows: –
One-term sequences have already been arranged in the primitive recursive sequence $v_{o(x)}$. Now, assuming that for some $n \geqq 1$ the n-term sequences have already been arranged in a primitive recursive sequence

$$w^{(n)}_{o(x)},$$

the $(n+1)$-term sequences can be enumerated as

$$w^{(n)}_{o(x)} * v_{o(y)} \quad (o(x), o(y) = 1, 2, \ldots).$$

[31] See D. Hilbert–P. Bernays: Grundlagen der Mathematik I. Berlin (1934) pp. 321 and 328.

Hence, using theorem (HB), they can be arranged into the primitive recursive sequence

$$w^{(n+1)}_{o(x)} = w^{(n)}_{\sigma_1(o(x))} * v_{\sigma_2(o(x))},$$

where the asterisk $*$ stands for the separating symbol between the terms (hence neither for "," nor for ";"). Consequently

$$w^{(o(y))}_{o(x)}$$

is obtained, by means of the definition

$$w^{(o(y))}_{o(x)} = \begin{cases} \wedge, & \text{if } y = \wedge \\ v_{o(x)}, & \text{if } o(y) = 1 \\ w^{(o(\mathrm{at}(y)))}_{\sigma_1(o(x))} * v_{\sigma_2(o(x))} & \text{otherwise,} \end{cases}$$

as a primitive recursive function of x and y, depending on $o(x)$ and $o(y)$ only. The values of this, for $o(y) = 1, 2, \ldots$, are the $o(y)$-term sequences built of the terms of the sequence $v_{o(x)}$. By a repeated application of theorem (HB), all these can be arranged in the primitive recursive sequence

$$w_{o(x)} = w^{(\sigma_2(o(x)))}_{\sigma_1(o(x))}.$$

Let $f_{o(x)}$ denote this primitive recursive sequence of all finite sequences of symbol chain lists. We shall apply it later.

9.6 Other Properties

The characteristic functions

$$\mathrm{lg}(x, y) \quad \text{and} \quad \mathrm{fg}(x, y)$$

of the relations "y is a term of the list x, or list sequence x" respectively can be defined by the following definitions-by-cases: –

$$\mathrm{lg}(x, y) = \begin{cases} \wedge, \text{ if } \mathrm{zkl}(x) = \wedge \,\&\, \mathrm{zk}(y) = \wedge \,\&\, (x = y \vee (\mathrm{E}u)[u \preceq x \,\&\, x = u, y] \vee \\ \quad \vee (\mathrm{E}u)[u \preceq x \,\&\, x = y, u] \vee (\mathrm{E}u_1)(\mathrm{E}u_1)[u_1, u_2 \preceq x \,\&\, x = \\ \quad = u_1, y, u_2]) \\ z_1 \quad \text{otherwise,} \end{cases}$$

$$\mathrm{fg}(x, y) = \begin{cases} \wedge, \text{ if } \mathrm{zklf}(x) = \wedge \,\&\, \mathrm{zkl}(y) = \wedge \,\&\, (x = y \vee (\mathrm{E}u)[u \preceq x \,\&\, x = \\ \quad = u; y] \vee (\mathrm{E}u)[u \preceq x \,\&\, x = y; u] \vee (\mathrm{E}u_1)(\mathrm{E}u_2)[u_1, u_2 \preceq x \,\&\, x = \\ \quad = u_1; y; u_2]) \\ z_1 \quad \text{otherwise.} \end{cases}$$

The last term $\mathrm{lfg}(x)$ of a sequence x of symbol-chain lists is determined by the definition

$$\mathrm{lfg}(x) = \mu_y [y \preceq x \,\&\, \mathrm{fg}(x, y) = \wedge \,\&\, (x = y \vee (\mathrm{E}u)[u \preceq x \,\&\, x = u;\ y])]$$

as primitive recursive.

For the notions "terminal" and "generated" we have to look more closely into the definition of our grammar.
Let the elements of M (the meta-symbols) be

$$m_1, m_2, \ldots, m_r,$$

and let

$$m_i(x) \quad (i = 1, 2, \ldots, r)$$

denote the characteristic function of the property "to be a value of m_i", that is to belong to the category denoted by m_i of the language generated in the first level by the phrase structured grammar (in the original sense). It was shown in the previous chapter that these are primitive recursive over the terminal vocabulary, which is here Z. The same is valid in the word set W over the extended alphabet Z'.
Let the elements of V (the preproductions) be

$$v_1, v_2, \ldots, v_s.$$

The productions of the second level are obtained from these by suitable substitutions of the values of the meta-symbols. Let

$$\mathrm{lp}_i(x) \quad (i = 1, 2, \ldots, s)$$

denote the characteristic function of the property "to be the left-hand side of a production resulting from v_i", and let

$$\mathrm{lpr}_i(x, y) \quad (i = 1, 2, \ldots, s)$$

be the characteristic function of the relation "x and y are the left- and right-hand sides, respectively, of a production resulting from v_i". If, for example, v_1 is the preproduction

$$z_1 m_1 m_2 := z_2 m_1 z_3 z_1, z_1 m_3,$$

then $\mathrm{lp}_1(x)$ and $\mathrm{lpr}_1(x, y)$ are determined with the help of the functions

$$m_1(x), m_2(x), m_3(x)$$

by the following definitions-by-cases: –

$$\mathrm{lp}_1(x) = \begin{cases} \wedge, & \text{if} \quad (\mathrm{E}y_1)(\mathrm{E}y_2)[y_1, y_2 \preceq x \,\&\, m_1(y_1) = \wedge \,\&\, m_2(y_2) = \wedge \,\& \\ & \quad \& \, x = z_1 y_1 y_2] \\ z_1 & \text{otherwise,} \end{cases}$$

and

$$\mathrm{lpr}_1(x, y) = \begin{cases} \wedge, & \text{if} \quad (\mathrm{E}u_1)(\mathrm{E}u_2)(\mathrm{E}u_3)[u_1, u_2 \preceq x \,\&\, u_3 \preceq y \,\&\, m_1(u_1) = \wedge \,\& \\ & \quad \& \, m_2(u_2) = \wedge \,\&\, m_3(u_3) = \wedge \,\&\, x = z_1 u_1 u_2 \,\&\, y = \\ & \quad = z_2 u_1 z_3 z_1, z_1 u_3] \\ z_1 & \text{otherwise.} \end{cases}$$

Using these, the characteristic functions

$$\mathrm{lp}(x) \quad \text{and} \quad \mathrm{lpr}(x, y)$$

of the property "to be the left-hand side of a production" (or "to be a potential category name"), and of the relation "x and y occur as the left- and right-hand sides respectively of the same production" can be defined as follows: –

$$\mathrm{lp}(x) = \begin{cases} \wedge, & \text{if} \quad \mathrm{lp}_1(x) = \wedge \vee \ldots \vee \mathrm{lp}_s(x) = \wedge \\ z_1 & \text{otherwise} \end{cases}$$

$$\mathrm{lpr}(x, y) = \begin{cases} \wedge, & \text{if} \quad \mathrm{lpr}_1(x, y) = \wedge \vee \ldots \vee \mathrm{lpr}_s(x, y) = \wedge \\ z_1 & \text{otherwise.} \end{cases}$$

Just as easily we could have given primitive recursive definitions for the property "to be a term of the right-hand side of a production" and with this of "to be a terminal concept"; however, these will not be needed.

A symbol chain list can be called "potentially terminal" if none of its terms occurs as the left-hand side of a production. Thus the characteristic function $t(x)$ of the property "to be a potentially terminal symbol chain list" can be defined as

$$t(x) = \begin{cases} \wedge, & \text{if} \quad \mathrm{zkl}(x) = \wedge \,\&\, (y)[y \preceq x \rightarrow (\mathrm{lg}(x, y) = \wedge \rightarrow \mathrm{lp}(y) = z_1)] \\ z_1 & \text{otherwise.} \end{cases}$$

Since, in direct generating, the left-hand side of a production, which occurs as a term of a list is replaced by the right-hand side of the same production, the characteristic function $\mathrm{dg}(x, y)$ of the relation "y is directly generated by the symbol chain list x" can be defined as follows:

$$\mathrm{dg}(x, y) = \begin{cases} \wedge, \text{ if } \mathrm{zkl}(x) = \wedge \,\&\, (\mathrm{E}u_1)(\mathrm{E}u_2)(\mathrm{E}u_3)(\mathrm{E}u_4)[u_1, u_2, u_3 \preceq x \,\&\, u_4 \preceq y \,\& \\ \quad \& \,\mathrm{lg}(x, u_2) = \wedge \,\&\, \mathrm{lpr}(u_2, u_4) = \wedge \,\&\, x = u_1 u_2 u_3 \,\&\, y = u_1 u_4 u_3] \\ z_1 \text{ otherwise.} \end{cases}$$

Finally, the characteristic function

$$\mathrm{gf}(x, y)$$

of the relation "y is a generating sequence of the type (9.2.1), beginning with x" is given by the following definition, which leads to a course-of-values recursion: –

$$\mathrm{gf}(x, y) = \begin{cases} \wedge, & \text{if} \quad \mathrm{zkl}(x) = \wedge \,\&\, \mathrm{zklf}(y) = \wedge \,\&\, (y = x \vee \\ & \vee (\mathrm{E}u_1)(\mathrm{E}u_2)[u_1, u_2 \preceq y \,\&\, \mathrm{gf}(x, u_1) = \wedge \,\& \\ & \& \,\mathrm{zkl}(u_2) = \wedge \,\&\, y = u_1;\, u_2 \,\& \\ & \& \,\mathrm{dg}(\mathrm{lfg}(u_1), u_2) = \wedge]) \\ z_1 & \text{otherwise.} \end{cases}$$

9.7 Recursive Enumerability

With this, we have a primitive recursive definition of every notion necessary for the formulation of the basic question concerning the language generated by a two-level grammar: – "If k is a category name, how can we decide what belongs to the category denoted by k?" This is true of x, if x is a potentially terminal symbol chain list, and there is a generating sequence of symbol chain lists which begins with k and terminates with x, for example, if

$$t(x) = \wedge \,\&\, (\mathrm{E}y)[\mathrm{gf}(k, y) = \wedge \,\&\, \mathrm{lfg}(y) = x]$$

holds. If the characteristic function of this property was also primitive recursive, then so would be the corresponding language. However for the y in $(\mathrm{E}y)$ [...], we might not be able to provide an upper bound, and an unbounded relation $(\mathrm{E}y)$ [...] cannot even be guaranteed to be general recursive.

We might have expected this on the basis of the similarity between generating a language and generating the theorems of an axiomatic mathematical theory. In the latter, a formula f is a theorem of the theory if there exists a sequence of formulae, starting with axioms and terminating with f, such that every term of the sequence can be "generated" from earlier terms with the application of certain rules of inference. In general an axiomatic theory is not recursively decidable. The corresponding "there exists" relation might not be general recursive. Concerning questions of decidability it would be senseless to use partial recursivity.

The recursivity of languages definable by two-level grammars, like Algol 68, is an open question.

The language generated in two levels in the way described above is, in any case, primitive-recursively enumerable in the sense that, for every category name, we can define a primitive recursive function the values of which are exactly the terminal constructions generated by this category name. Using the primitive recursive sequence $f_{o(x)}$ from section 9.5, in which all the sequences of symbol-chain lists are arranged, the following function does this for a $k \in K$: –

$$k(x) = \begin{cases} \mathrm{lfg}(f_{o(x)}), & \text{if} \quad t(\mathrm{lfg}(f_{o(x)})) = \wedge \,\&\, \mathrm{gf}(k, f_{o(x)}) = \wedge \\ \wedge & \text{otherwise.} \end{cases}$$

9.8 Two-level Language with Finite Terminal Concepts

In developing Algol 60, it was conceivable that one could use a two-level grammar containing only a finite number of terminal concepts.
However, a language generated by such a grammar can also be generated in one level [32].
The proof of this is rather complicated. In the proof one has to take into consideration the preproductions from which the separate productions were deduced (by substituting certain values of their meta-symbols). The basic idea of the proof is the following: – If there are only finitely many terminal concepts, then the terms of the right-hand side of a production, different from these and from the (finitely many) category names, can only have a regulatory kind of role. Specially they only determine which preproductions generate such productions as they are applicable to the term under consideration. By this is determined the order in which terminal concepts will occur in the terminal constructions generated by the category names. Since there are only a large finite number of combinations of the finitely many preproductions, from this point of view the regulating right-hand side terms of the productions can be divided into a large finite number of sets. What matters is only the set to which such a term belongs, not its concrete form.
A simple example might make this clearer. Let us consider the two-level grammar with

$$\mathrm{Z} = \{z_1; z_2; z_3; t_1; t_2\} \quad \mathrm{M} = \{m\} \quad \mathrm{P} = \{m :\!= z_1 m z_1 : m :\!= z_2\}$$
$$\mathrm{V} = \{m :\!= z_1 m z_1,\ t_1; z_1 m z_1 :\!= t_2; z_3 :\!= m\} \quad \mathrm{K} = \{z_3\}.$$

Since M and P are the same as in the "sweet Mary" language of section 9.3, the values of the single metasymbol m are again the symbol chains

$$\underbrace{z_1 \ldots z_1}_{n\text{-times}}\, z_2\, \underbrace{z_1 \ldots z_1}_{n\text{-times}} \quad \text{for} \quad n = 0, 1, 2, \ldots,$$

which I shall denote by a_n.
Clearly, if $m = a_n$, then

$$z_1 m z_1 = a_{n+1}.$$

Hence the following productions are obtained from V for $n = 0, 1, 2, \ldots$: –

$$a_n :\!= a_{n+1}, t_1$$
$$a_{n+1} :\!= t_2$$
$$z_3 :\!= a_n.$$

[32] See my paper quoted in footnote [26].

Among the right-hand side terms, only t_1 and t_2 do not occur as left-hand side terms. Hence only these two are the terminal concepts of the language. Therefore the terminal constructions generated by the single category name z_3 can be lists consisting of t_1 and t_2 only. The right-hand side terms a_n and a_{n+1} have simply a regulating role to decide the order in which t_1 and t_2 occur in these lists. What matters here is, for which combinations C of the left-hand sides of the preproductions will the terms a_{n+1} or a_n occur in the left-hand sides of the productions generated by the preproductions in C? In this simple example there are only two possibilities: a_0 occurs in the left-hand side of a production only if it is obtained from the first preproduction. Hence a_0 can be replaced by the list

$$a_1, t_1$$

only where a_1 is one of the values a_{n+1}, while a_{n+1} can always be replaced by both

$$a_{n+2}, t_1$$

and t_2, where a_{n+2} is again one of the values a_{n+1}. Since all the values a_{n+1} have the same effect, they can all be replaced by a single new symbol g. Hence we obtain the following five productions: –

$$a_0 := g, t_1$$

$$g := g, t_1$$

$$g := t_2$$

$$z_3 := a_0$$

$$z_3 := g.$$

These have the same effect as the original infinite number of productions. It would be easy to obtain further simplifications. However, here we shall deal only with the finiteness of the number of productions. At the stage we have now reached, we have sufficient information to obtain all the terminal constructions generated by z_3. Only a_0 and g are directly generated by z_3; by a_0 only

$$g, t_1$$

is directly generated, while by g

$$g, t_1 \quad \text{and} \quad t_2,$$

where the second is already terminal. To the first both the second and third productions can be applied, with the results

$$g, t_1, t_1 \quad \text{and} \quad t_2, t_1,$$

where the second is again terminal. From the first we obtain similarly

$$g, t_1, t_1, t_1 \quad \text{and} \quad t_2, t_1, t_1,$$

and so on. Clearly, every terminal construction generated by the category name z_3 begins with t_2, followed by a chain consisting solely of t_1.
It is easy to see that this simple language can also be generated by the following two productions:

$$z_3 := z_3, t_1$$

$$z_3 := t_2.$$

But the purpose of this example was to elucidate the elaborate general considerations, by means of which one can show that in the case of a large finite number of terminal concepts, a language generated in two levels can also be generated by a one-level phrase-structured grammar. So, if it is not circular, such a language is primitive recursive.

Chapter 10

Does Recursivity Mean Restriction?

10.1 The Recursivity of Everything Computable

It was shown in Ch. 4 that everything obtainable by a computer is partial recursive. Actually, a really *partial* recursive function might not be obtained at all. If one can decide for every argument whether the function f under consideration is defined there or not, then the situation is clear. If this decision is made in a general recursive way, then the agreement that the function take a fixed value wherever f is not defined turns the definition of f into the definition of a general recursive function. However, for proper partial recursive functions the possibility of finding such a decision procedure is hopeless. If a program for the computation of such a function is fed into a computer and, after the input of arbitary arguments, the computer starts calculating, one can never know whether the computer has failed to stop because the computation is too lengthy, or if it will work on forever, without computing anything.

One always strives to feed "reasonable" programs into the computer, whereby for arbitrary initial data the calculation will come to a halt after a (large) finite number of computing steps. With this, the above statement can be reduced to the following: – whatever can really be obtained by the use of a computer is general recursive. Moreover, after suitable coding, it can become a general recursive numeric function. Thus the question arises: – Does this mean an essential restriction on the abilities of the computer?

10.2 Church's Thesis

Assuming Church's well-known thesis [33] does not mean any restriction. According to this thesis, every numeric function is general recursive if its values are computable in a finite number of steps for all arguments. Of course, this is not an exact mathematical proposition, because the term "computable" is not exactly defined. Consequently, it can be neither proved nor disproved mathematically. There are many arguments for, and some against, the plausibility of Church's thesis. Perhaps the most striking argument against it is due to L. Kalmár [34]. He has proved that the validity of Church's thesis would imply the following hardly believable fact: There exists a simple proposition (namely that there is a natural number n, for which a fixed numeric function $\varphi(n, m)$ does not vanish for all m) which we know is true, but still cannot be proved in any way.

I myself agree with Kalmár's conviction that effective computability is one of those notions the definition of which can never be considered complete in the course of the development of mathematics.

As a matter of fact, up to now no effectively computable numeric function (that is one computable everywhere in a finite number of steps) has been found which is not general recursive. Therefore, computers, which in principle are capable of computing every general recursive function, yield the most that can be expected according to the present state of our knowledge. Let us hope, provided a counter-example to Church's thesis is made known, then, one hopes, the technological means will develop to modify computers to enable them to compute such functions.

[33] A. Church: *An unsolvable problem of elementary number theory*, Amer. Journ. Math. **58** (1936) pp. 345–363.

[34] L. Kalmár: *Solution of a problem of K. Schröter concerning the definition of the notion of general recursive functions*. MTA III. O. Közl. Publ. of class III. of the Hung. Acad. Sci. **7** (1957) pp. 19–38 (in Hungarian).

Chapter 11

Recursivity of Lisp 1.5

11.1 A Set of Numeric Structure

The recursive theory of the programming language Lisp 1.5 [35] indicated in section 3.7, can not only be dealt with by embedding it in a word set. A holomorphic set is a typical example for another case of a set with a numeric structure. As to general information on such sets, I refer to footnote [10].

Here we are going to study lists, that is, finite linear arrays which are built out of certain elements. These are elements of a word set over a finite alphabet containing letters, digits and several special symbols. However, the lengths of words used for this purpose are bounded. Hence the set A of elements is finite. Let this be denoted by

$$A = \{a_1, a_2, \ldots, a_t\}.$$

11.2 Basic Notions

In what follows, each of these is considered as a single symbol (and not as a chain of symbols). All the elements play the role of 0 in our holomorphic set. Since the set of 0-elements is customarily denoted by H_0, we put

$$H_0 = A.$$

The terms of a list are either elements or lists that have been constructed earlier. One does not have to consider lists of arbitrarily many terms, since they can be decomposed into pairs. To the first term of the list, the list of

[35] See the paper quoted in footnote [13], and R. Péter: *Die Rekursivität der Programmierungssprache „Lisp 1.5" in Spezialfällen der angeordneten freien holomorphen Mengen,* submitted to Acta Cybernetica on February 1, 1973.

the remaining terms can be chosen as the second item of the pair. The latter list can again be considered as a pair in a similar way. To the last term of the given list, the empty list is to be chosen as the mate. This is denoted by "NIL" and is also considered to be an element.

Thus instead of lists, we shall deal with symbolic expressions or in short S-expressions. In the first place, the elements are S-expressions. Moreover, if s_1 and s_2 are arbitrary S-expressions, then the pair

$$s = (s_1, s_2)$$

is also.

Thus if s corresponds to a list, then s_1 corresponds to its first term (the head) and s_2 to that (perhaps empty) list which results if the first term is removed from the original list (the tail).

Here s_1 and s_2 as functions of s will be denoted by

$$s_1 = \text{car}\,(s), \quad \text{and} \quad s_2 = \text{cdr}\,(s),$$

while s as a function of s_1 and s_2 will be denoted by

$$s = \text{cons}\,(s_1, s_2).$$

This two-place function cons plays the role of a successor function here. If s is an element of H_0, that is

$$s \in H_0,$$

then we say that the order of s is 0, that is

$$o(s) = 0.$$

If s_1 and s_2 are at most of order n, but at least one of them has order n, then the order of

$$s = \text{cons}\,(s_1, s_2)$$

is $n+1$, that is

$$o(s) = n+1.$$

The set of S-expressions of order n will be denoted by H_n, while H is the union of the sets H_n for $n=0, 1, 2, \ldots$.

Every element x of H is either an element or has the form

$$\text{cons}\,(x_1, x_2) = (x_1 \cdot x_2),$$

where x_1 and x_2 are uniquely determined:

$$x_1 (= \text{car}\,(x))$$

is that S-expression which results if one omits the opening parenthesis of the symbol chain x, and then copies its symbols (going from left to right)

until the numbers of the left and right parentheses coincide. Furthermore

$$x_2 (= \text{cdr}\,(x))$$

will be that chain of symbols which results if, from this remaining part of x, one omits the point at the beginning and the last closing parenthesis.
Every element is the only (not proper) predecessor of itself. The immediate predecessors of

$$x = \text{cons}\,(x_1, x_2)$$

are x_1 and x_2, and the proper predecessors of x are the predecessors of x_1 and x_2. Consequently the order of a proper predecessor y of x (denoted by $y \prec x$) is less than $o(x)$.
The natural numbers

$$0, \quad 1, \quad 2, \ldots$$

will be identified in H by a fixed member of each

$$H_0, \quad H_1, \quad H_2, \ldots$$

respectively, that is by

$$h_0 = \text{NIL}, \quad h_1 = \text{cons}\,(h_0, h_0), \quad h_2 = \text{cons}\,(h_1, h_1), \ldots .$$

Thus for every natural number i we have

$$i = o(i) = o(h_i) = h_i.$$

Moreover

$$o(x) \prec o(y)$$

is equivalent to

$$o(x) \prec\!\!\prec o(y)$$

and

$$x \preceq o(y)$$

implies that x is a natural number, that is

$$x = o(x).$$

11.3 Primitive Recursion in H

Now the scheme of primitive recursion in H reads as follows: –

$$\begin{cases} f(a, u_1, \ldots, u_n) = g_a(u_1, \ldots, u_n), & \text{if} \quad a \in H_0 \ (= A) \\ f(\text{cons}\,(x_1, x_2), u_1, \ldots, u_n) = \\ = g\,(x_1, x_2, u_1, \ldots, u_n, f(x_1, u_1, \ldots, u_n), f(x_2, u_1, \ldots, u_n)), \end{cases}$$

where

$$g_{a_1}, g_{a_2}, \cdots, g_{a_t}, g$$

are already defined functions.

11.3.1 Initial Functions

As initial functions we take the elements of H_0, the successor function "cons", and the characteristic function of the equality: –

$$\text{equal}\,(x, y).$$

Here b is said to be the characteristic function of a relation B, if everywhere b takes the value h_0 or h_1 according to whether B is valid at the corresponding point or not. Also here we say that if B is primitive recursive so is b. Thus we have

$$\text{equal}\,(x, y) = \begin{cases} h_0, & \text{if} \quad x = y \\ h_1, & \text{if} \quad x \neq y. \end{cases}$$

A function is primitive recursive in H if it can be obtained from the initial functions by means of finitely many applications of substitutions and primitive recursions.

11.4 Examples

Next we list several examples of primitive recursive functions in H.

1. The identity function

$$\text{id}\,(x) = x$$

can be obtained by the primitive recursion

$$\begin{cases} \text{id}\,(a) = a, & \text{if} \quad a \in H_0 \\ \text{id}\,(\text{cons}\,(x_1, x_2)) = \text{cons}\,(x_1, x_2), \end{cases}$$

where the constant

$$g_a = a$$

and the function

$$g = \text{cons}$$

are initial functions. Here g depends only on the two indicated variables, but the introduction of dummy variables, on which a function does not really depend, is also permitted in H.

2. The definitions of the immediate predecessors of x are

$$\begin{cases} \text{car}\,(a) = \text{NIL} = h_0, & \text{if} \quad a \in H_0 \\ \text{car}\,(\text{cons}\,(x_1, x_2)) = x_1, \end{cases}$$

$$\begin{cases} \text{cdr}\,(a) = h_0, & \text{if} \quad a \in H_0 \\ \text{cdr}\,(\text{cons}\,(x_1, x_2)) = x_2. \end{cases}$$

3. The natural numbers

$$h_0, \quad h_1, \quad h_2, \ldots.$$

are primitive recursive, since this is true for the element h_0, and its validity can be proved by induction h_n to h_{n+1}. This can be obtained by the substitution

$$\text{cons}\,(h_n, h_n).$$

4. The characteristic function

$$\text{atom}\,(x)$$

of the property "to be an element of H_0", that is to be an element, and its opposite

$$\overline{\text{atom}}\,(x)$$

can be defined by the following primitive recursions: –

$$\begin{cases} \text{atom}\,(a) = h_0, & \text{if} \quad a \in H_0 \\ \text{atom}\,(\text{cons}\,(x_1, x_2)) = h_1 \end{cases}$$

$$\begin{cases} \overline{\text{atom}}\,(a) = h_1, & \text{if} \quad a \in H_0 \\ \overline{\text{atom}}\,(\text{cons}\,(x_1, x_2)) = h_0. \end{cases}$$

These correspond to the functions

$$\text{sg}(x) \quad \text{and} \quad \overline{\text{sg}}(x)$$

in number theory as well as to the functions

$$\text{sig}(x) \quad \text{and} \quad \overline{\text{sig}}(x)$$

in word sets. They have their counterparts in every set of numeric structure. They can always be used to prove the following statements: –

i) The primitive recursive relations are closed under negations, conjunctions and implications.

ii) A function built up from primitive recursive functions by means of primitive recursive relations is also a primitive recursive function. The exact meaning of this was formulated in both the case of number theory and of word sets, and the theory can be generalized to other sets of numeric structure.

iii) Using i) and ii), one can show that if $B(u_0, u_1, \ldots, u_n)$ is primitive recursive, then so are

$$(\text{E}y)\,[y \preceq x \;\&\; B(y, u_1, \ldots, u_n)],$$

$$(y)\,[y \preceq x \rightarrow B(y, u_1, \ldots, u_n)]$$

and

$$\mu_y\,[y \preceq x \;\&\; B(y, u_1, \ldots, u_n)].$$

The meaning of their counterparts was given in section 3.6.2.

5. The characteristic function $\text{pred}(x, y)$ of the relation $y \preceq x$ occurring above has the following primitive recursive definition: –

$$\begin{cases} \text{pred}\,(a, y) = \text{equal}\,(y, a), & \text{if} \quad a \in H_0 \\ \text{pred}\,(\text{cons}\,(x_1, x_2), y) = \begin{cases} h_0, & \text{if} \quad \text{cons}\,(x_1, x_2) = \\ = y \vee \text{pred}\,(x_1, y) = h_0 \vee \text{pred}\,(x_2, y) = h_0 \\ h_1 \quad \text{otherwise.} \end{cases} \end{cases}$$

Consequently the relation

$$y \prec x \equiv y \preceq x \;\&\; y \neq x$$

is also primitive recursive.

6. Finally $o(x)$ is also primitive recursive in H. Indeed, if

$$x = \text{cons}\,(x_1, x_2),$$

then one of $o(x_1)$, $o(x_2)$ must be exactly one less than $o(x)$. Moreover for each number n, the successor of n is

$$\text{cons}\,(n, n).$$

Hence $o(x)$ can be defined by

$$\begin{cases} o(a) = h_0, & \text{if} \quad a \in H_0 \\ o(\text{cons}\,(x_1, x_2)) = \begin{cases} \text{cons}\,(o(x_1), o(x_1)), & \text{if} \quad o(x_2) \preceq o(x_1) \\ \text{cons}\,(o(x_2), o(x_2)) & \text{otherwise.} \end{cases} \end{cases}$$

11.5 The Order $o(x)$

We add three important remarks to the definition of $o(x)$: –

a) For

$$x = \text{cons}\,(x_1, x_2)$$

$o(x)$ was defined with the help of the earlier value $o(x^{-1})$, where in general x^{-1} denotes a fixed predecessor of x of order $o(x) - 1$, in our case this was a fixed one of x_1, x_2, the order of which is not less than that of the other.

It is useful that the scheme

$$\begin{cases} f(a) = g_a, & \text{if} \quad a \in H_0 \\ f(\text{cons}\,(x_1, x_2)) = g(x_1, x_2, f(x_1), f(x_2), f(\text{cons}^{-1}\,(x_1, x_2))) \end{cases}$$

(where parameters are admitted) remains within the class of functions which are primitive recursive in H. Indeed, applying the primitive recursive auxiliary function

$$g'(x_1, x_2, v_1, v_2) = \begin{cases} g(x_1, x_2, v_1, v_2, v_1), & \text{if} \quad o(x_2) \preceq o(x_1) \\ g(x_1, x_2, v_1, v_2, v_2) & \text{otherwise,} \end{cases}$$

the above function f is also definable by the primitive recursion

$$\begin{cases} f(a) = g_a, & \text{if} \quad a \in H_0 \\ f(\text{cons}\,(x_1, x_2)) = g'(x_1, x_2, f(x_1), f(x_2)). \end{cases}$$

In the particular case

$$g_a = a$$

and

$$g(x_1, x_2, v_1, v_2, v_3) = v_3,$$

x^{-1} itself is obtained as a primitive recursive function.
For every natural number n, we have

$$n^{-1} = n - 1.$$

b) Instead of the natural numbers it is more appropriate to use the function $o(x)$ in H. For instance, the $o(x)$th iterate of a primitive recursive function f at a place y has the following primitive recursive definition in H: –

$$\begin{cases} f^{(o(a))}(y) = y, & \text{if} \quad a \in H_0 \\ f^{(o(\text{cons}\,(x_1, x_2)))}(y) = f\big(f^{(o(\text{cons}^{-1}(x_1, x_2)))}(y)\big). \end{cases}$$

The iteration

$$^{(f)}\text{it}(x, y) = f^{(o(x))}(y),$$

yields an example of a primitive recursive sequence:

$$^{(f)}\text{it}_{o(x)}(y),$$

since it does not really depend on x, but only on $o(x)$.
Since $o(x)$ is always a natural number,

$$o\big(o(x)\big) = o(x),$$

that is every non-zero-th iterate of $o(x)$, is equal to $o(x)$.
c) In section 3.4.1, we referred to the fact that, what we proved there for word sets (namely that every numeric primitive recursive function can be represented by a primitive recursive word function) is valid in every set of numeric structure, in particular in H. More precisely, for every primitive recursive numeric function

$$\varphi(m_1, \ldots, m_n)$$

there is a function

$$f(u_1, \ldots, u_n)$$

primitive recursive in H such that for all $u_1, \ldots, u_n$

$$o\big(f(u_1, \ldots, u_n)\big) = \varphi\big(o(u_1), \ldots, o(u_n)\big).$$

The function

$$o\big(f(o(u_1), \ldots, o(u_n))\big)$$

can be considered as the representative of φ in H. The representatives of the numeric functions can be denoted in the same way as the originals. Through their characteristic functions, the numeric primitive recursive relations can also be represented by primitive recursive relations in H (which are denoted in the same way).

11.6 Coding Lists by Elements

In order to be able to handle course-of-values recursions, we have to code sequences of elements of H, by elements of H, in such a way that the terms of a sequence can be recovered from its code. For a set of numeric structure, in which one of the successor functions is of at least two variables (as in the present case), in my paper quoted in [10] I have constructed a rather simple example of such coding. In the present particular case, however, it is more natural to use another method, which can also be generalized. In this, a finite sequence is considered as a list (of S-expressions), with which we have previously associated an S-expression: –

with the list s_0 the S-expression $(s_0 \cdot \mathrm{NIL})$

with the list s_0, s_1 the S-expression $(s_0 \cdot (s_1 \cdot \mathrm{NIL}))$

with the list s_0, s_1, s_2 the S-expression $(s_0 \cdot (s_1 \cdot (s_2 \cdot \mathrm{NIL})))$

.....

and with the empty list e. g. $\mathrm{NIL}(=h_0)$.
Thus with the list

$$s_0, s_1, \ldots, s_n$$

the element

$$x = c_n(s_0, s_1, \ldots, s_n) = \mathrm{cons}\,(s_0, \mathrm{cons}\,(s_1, \ldots, \mathrm{cons}\,(s_n, h_0)\ldots))$$

is associated. It can be seen that

$$n = o(n) \leq o(x)$$

is satisfied here.
The terms of the list can be recovered from its code x as primitive recursive functions of x: –

$$s_0 = \mathrm{car}\,(x),\ s_1 = \mathrm{car}\,(\mathrm{cdr}\,(x)),\ \ldots,\ s_n = \mathrm{car}\,(\mathrm{cdr}^{(n)}\,(x)).$$

Moreover

$$\mathrm{cdr}^{(n+1)}\,(x) = h_0.$$

Consequently, the characteristic function list(x) of the property "x codes a list" can be defined as a primitive recursive function: –

$$\text{list}(x) = \begin{cases} h_0, & \text{if} \quad x = h_0 \vee (\text{E}y)[y \preceq o(x) \,\&\, \text{cdr}^{(o(y))}(x) \notin H_0 \,\& \\ & \qquad\qquad \& \,\text{cdr}^{(o(y)+1)}(x) = h_0] \\ h_1 & \text{otherwise.} \end{cases}$$

The "length" long(x) (the above n) can be obtained as follows:

$$\text{long}(x) = \mu_y[y \preceq o(x) \& \text{cdr}^{(o(y))}(x) \notin H_0 \& \text{cdr}^{(\sigma(y)+1)}(x) = h_0].$$

The expression

$$\mu_y[y \preceq z \,\&\, B(y, u_1, \ldots, u_n)]$$

needs a little explanation. Its value is obtained as the first term y of a certain list, enumerating all the predecessors of z, which satisfies the relation

$$B(y, u_1, \ldots, u_n)$$

and is h_0, if this relation is not satisfied by any predecessor of z. Thus we have

$$\text{long}(x) = h_0$$

exactly, if x codes the empty list or does not code any list at all. Since $y \preceq o(x)$, long(x) is necessarily a natural number.
A primitive recursive sequence which enumerates predecessors of x, and which if x codes a list, for

$$o(y) \preceq \text{long}(x)$$

yields the $o(y)$ th term of this list, can be defined as follows: –

$$k_{o(y)}(x) = \begin{cases} \text{car}\,(\text{cdr}^{(o(y))}(x)), & \text{if} \quad \text{list}(x) = h_0 \& o(y) \preceq \text{long}(x) \\ x & \text{otherwise.} \end{cases}$$

11.7 Course-of-values Recursion in H

If a list

$$\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_l = x \tag{11.7.1}$$

consists precisely of the predecessors of x (with $\bar{x}_i \neq x$ for $i < l$), then the "course-of-values function" of a function f is defined as

$$f^*(x) = c_l(f(\bar{x}_0), \ldots, f(\bar{x}_l)).$$

Every earlier value of f can be obtained from this as

$$f(\bar{x}_i) = k_i(f^*(x))$$

(with $i<l$). Therefore the scheme of course-of-values recursion in H reads as follows: –

$$f(a) = g_a, \quad \text{if} \quad a \in H_0$$

$$f(\text{cons}\,(x_1, x_2)) = g(x_1, x_2, f^*(x_1), f^*(x_2)),$$

where the functions g_a and g are primitive recursive (and might contain parameters).

It turns out to be helpful if we choose the sequence (11.7.1) in such a way that for $x \in H_0$

$$l = 0 \quad \text{and} \quad \bar{x}_0 = x,$$

and for

$$x = \text{cons}\,(x_1, x_2)$$

we first enumerate the predecessors of x_2 in their already given order, then the predecessors of x_1 in their given order, and finally put

$$x = \bar{x}_l.$$

It can be shown then that $l=l(x)$ as well as $f^*(x)$ will be primitive recursive. With this, moreover, we have

$$f(x) = k_{l(x)}(f^*(x))$$

is primitive recursive as well. Therefore, the course-of-values recursion does not extend the class of functions which are primitive recursive in H. This holds true in general for sets of numeric structure, after several further initial functions are chosen. In our case, however, these can be defined as primitive recursive.

As a simple application of the functions c_n and $k_i(x)$ I also mention the reducibility of the simultaneously recursive definition of several functions

$$f_0, f_1, \ldots, f_n$$

to the recursive definition of the single function

$$f = c_n(f_0, \ldots, f_n).$$

From this, the original function can be recovered by the substitutions

$$f_0 = k_0(f), \ldots, f_n = k_n(f).$$

11.7.1 More Recursions in H

In the paper quoted in[13] the characteristic function

$$\mathrm{eq}\,(x, y)$$

of the property "x and y are equal elements" was taken as an initial function, and from this

$$\begin{cases} \mathrm{equal}\,(a, y) = \mathrm{eq}(a, y), & \text{if} \quad a \in H_0 \\ \mathrm{equal}\,(\mathrm{cons}\,(x_1, x_2), y) = \begin{cases} h_0, & \text{if} \quad \mathrm{equal}\,(x_1, \mathrm{car}\,(y)) = \\ & \quad = h_0 \,\&\, \mathrm{equal}\,(x_2, \mathrm{cdr}\,(y)) = h_0 \\ h_1 & \text{otherwise} \end{cases} \end{cases}$$

was defined later. I want to add here two remarks.

1) I show that

$$\mathrm{eq}\,(x, y)$$

can be defined as a primitive recursive function in H.

First of all the characteristic function f_i of the property to be equal to the element

$$a_i \quad (i = 1, 2, \ldots, t)$$

can be defined by the primitive recursion

$$\left.\begin{array}{l} f_i(a_1) = h_1 \\ \cdots\cdots\cdots\cdots \\ f_i(a_{i-1}) = h_1 \end{array}\right\} \qquad \text{(only for } i = 1\text{)}$$

$$f_i(a_i) = h_0$$

$$\left.\begin{array}{l} f_i(a_{i+1}) = h_1 \\ \cdots\cdots\cdots\cdots \\ f_i(a_t) = h_1 \end{array}\right\} \qquad \text{(only for } i = t\text{)}$$

$$f_i(\mathrm{cons}\,(x_1, x_2)) = h_1.$$

Using these we can put

$$\mathrm{eq}\,(x, y) = \begin{cases} h_0, & \text{if} \quad (x = a_1 \,\&\, y = a_1) \vee \ldots \vee (x = a_t \,\&\, y = a_t) \\ h_1 & \text{otherwise.} \end{cases}$$

2) The above definition of

$$\mathrm{equal}\,(x, y),$$

which, by means of the primitive recursive auxiliary function

$$g(x, y) = \begin{cases} h_0, & \text{if} \quad x = h_0 \,\&\, y = h_0 \\ h_1 & \text{otherwise} \end{cases}$$

can also be written in the form

$$\begin{cases} \text{equal}\,(a, y) = \text{eq}\,(a, y), & \text{if} \quad a \in H_0 \\ \text{equal}\,(\text{cons}\,(x_1, x_2), y) = g(\text{equal}\,(x_1, \text{car}\,(y)), \text{equal}\,(x_2, \text{cdr}\,(y))), \end{cases}$$

is not a primitive recursion, since the argument y in it does not remain unchanged. First it is replaced by $\text{car}(y)$ and then by $\text{cdr}(y)$.
In my paper quoted in [10], I pointed out that, possibly after adding suitable auxiliary functions, such a definition can be reduced to course-of-values recursions. Thus, it can be reduced to primitive recursions as well, even in the case of a "nested recursion", in which the expressions substituted for the parameters may depend on earlier values of the functions to be defined. However it was used there in the sense that the characteristic function of the equality (in our case

$$\text{equal}\,(x, y)$$

itself) was an initial function. Now, in our particular case the adding of such further initial functions is not necessary. I shall illustrate this reduction with an example, which is applied in Lisp 1.5.

11.8 Examples

Let x and y correspond to lists of elements of the same length: –

$$(u_1, u_2, \ldots, u_n) \quad \text{and} \quad (v_1, v_2, \ldots, v_n),$$

and let z correspond to a third list. Let us attach to the beginning of this third list the pair-list

$$(\text{cons}\,(u_1, v_1),\ \text{cons}\,(u_2, v_2),\ \ldots,\ \text{cons}\,(u_n, v_n))$$

constructed from the first two lists. Let the S-expression corresponding to this list be denoted by

$$\text{pairlis}\,(x, y, z)$$

whose value is irrelevant if x, y, z are not of the above type.
If x is an element, it can only correspond to the empty list. Hence the same is true for y, and thus nothing is attached to the third list. Consequently we obtain the following definition: –

$$\begin{cases} \text{pairlis}(a, y, z) = z, & \text{if} \quad a \in H_0 \\ \text{pairlis}(\text{cons}(x_1, x_2), y, z) = \text{cons}(\text{cons}(x_1, \text{car}(y)), \text{pairlis}(x_2, \text{cdr}(y), z)). \end{cases}$$

Using the primitive recursive function

$$g(u_1, u_2, u_3) = \text{cons}\,(\text{cons}\,(u_1, \text{car}\,(u_2)), u_3)$$

we have the shorter definition

$$\begin{cases} \text{pairlis}(a,y,z) = z, & \text{if} \quad a \in H_0 \\ \text{pairlis}(\text{cons}(x_1,x_2),y,z) = g(x_1,y,\text{pairlis}(x_2,\text{cdr}(y),z)) \end{cases} \tag{11.8.1}$$

For this definition, in which

$$\text{cdr}(y)$$

is substituted for the parameter y, I shall illustrate the steps of the reduction to course-of-values recursions. In this simple example it goes easily, but the methods indicated can also be applied to nested recursions.

From (11.8.1) values of the following types are obtained for pairlis (x, y, z): –

$$\begin{aligned} &z, && \text{if} \quad x \in H_0, \\ &g(x_1,y,z), && \text{if} \quad x = \text{cons}(x_1,x_2) \;\&\; x_2 \in H_0, \\ &g(x_1,y,g(x_{21},\text{cdr}(y),z)), && \text{if} \quad x_2 = \text{cons}(x_{21},x_{22}) \;\&\; x_{22} \in H_0, \\ &g(x_1,y,g(x_{21},\text{cdr}(y),g(x_{221},\text{cdr}(\text{cdr}(y)),z))), \\ & && \text{if} \quad x_{22} = \text{cons}(x_{221},x_{222}) \;\&\; x_{222} \in H_0, \end{aligned}$$

and so on.

It can be seen that the function values are built from nestings of the functions

$$\text{cdr}(u), \quad g(u_1,u_2,u_3),$$

where no function is substituted for u_1. Let a function $f(x, y, z)$ satisfy the following conditions: –

(1) for every y and z there are x' and x'' with

$$f(x', y, z) = y \quad \text{and} \quad f(x'', y, z) = z,$$

(2) for every y, z, u there is an x with

$$f(x, y, z) = \text{cdr}(f(u, y, z)),$$

(3) for every y, z, u_1, u_2, u_3 there is an x with

$$f(x, y, z) = g(u_1, f(u_2, y, z), f(u_3, y, z)).$$

Then $f(x, y, z)$ has all such nestings among its values, in particular all the values of pairlis (x, y, z).

Now, such an $f(x, y, z)$ can be defined by means of the primitive recursive functions

$$k_i(u), \quad c_n(u_0, u_1, \ldots, u_n)$$

for $i=0, 1, 2, 3$; $n=0, 1, 3$, where for $i \leqq n$

$$k_i(c_n(u_0, u_1, \ldots, u_n)) = u_i$$

in the following way: –

$$\begin{cases} f(a,y,z) = y, & \text{if } a \in H_0 \\ f(\text{cons}(x_1,x_2),y,z) = \begin{cases} z, & \text{if } o(k_0(x_1)) = h_0 \\ \text{cdr}(f(k_1(x_1),y,z)), & \text{if } o(k_0(x_1)) = h_1 \\ g(k_1(x_1),f(k_2(x_1),y,z),f(k_3(x_1),y,z)) & \text{otherwise.} \end{cases} \end{cases} \tag{11.8.2}$$

Indeed, properties (1)–(3) are satisfied by

$$x' = h_0 \quad \text{and} \quad x'' = \text{cons}\,(\underbrace{c_0(h_0)}_{x_1}, x_2),$$

$$x = \text{cons}\,(\underbrace{c_1(h_1,u)}_{x_1}, x_2),$$

$$x = \text{cons}\,(\underbrace{c_3\,(h_2,u_1,u_2,u_3)}_{x_1}, x_2),$$

respectively by choosing x_2, for example $x_2 = h_0$.

Furthermore, by a definition of type (11.8.2), (I shall return to th equestion of reformulating these as course-of-values recursions) one can obtain a function $w(x, u_1, u_2)$, which unfolds the nested values of the function f in the sense that, for all values of the arguments,

$$f(x,f(u_1, y, z),f(u_2, y, z)) = f(w(x, u_1, u_2), y, z)$$

holds.

With the use of this function, we can finally, through primitive recursion, define a function $q(x)$ which, so to say, sifts out the value of pairlis (x, y, z) from the value of $f(x, y, z)$. Similarly, for all the values of the arguments,

$$\text{pairlis}\,(x, y, z) = f(q(x), y, z)$$

and the function $w(x, u_1, u_2)$ can be defined primitive recursively in the same way as pairlis(x, y, z).

According to the definition of $k_{o(y)}(x)$ the values

$$k_i(x_1) \quad (i = 0, 1, 2, 3)$$

occurring in (11.8.2) are predecessors of x_1. If $k_i(x)$ is the $v(x, i)$th in the list (11.8.1) of the predecessors of x, and $f^*(x, y, z)$ denotes the course-of-values function of $f(x, y, z)$, then by section 11.7

$$f(k_i\,(x_1), y, z) = k_{v(x_1,i)}(f^*(x_1, y, z)).$$

If we substitute the right-hand sides of these identities instead of their left-hand sides in (11.8.2) for $i=0, 1, 2, 3$, we can see that since $v(x, i)$ is primitive recursive, a course-of-values recursion is obtained. This shows that for a general set of well-behaved numeric structure the function corresponding to $v(x, i)$ has to be added to the initial functions.

In our special case, however, $v(x, i)$ can be defined in a primitive recursive way in H.

Thus pairlis (x, y, x) is primitive recursive in H. It can be shown in a similar way that recursions, in which substitutions occur for the parameters (even if nested values of the function to be defined occur among these), do not lead out of the class of functions primitive recursive in H.

11.9 General and Partial Recursive Functions in H

In every set H of "numeric structure", hence also in the set of S-expressions, one can introduce the general recursive functions similarly as in the number theoretic case. The values of these can be obtained everywhere from defining systems of equations by means of finitely many substitutions of elementary terms (in our particular case S-expressions) for variables and substitutions of one side of an equality for the other. By omitting the requirement "everywhere", we obtain the partial recursive functions in H. (All of these can also be defined by primitive recursions and suitable "unbounded μ-operations".)

Chapter 12

Decision Tables

12.1 Decision Tables versus Flow Charts

For some time it has been a tendency in practice to use decision tables [36] instead of flow charts, if, in the flow charts several logical vertices would follow one after the other, thus making the structure and flow of the calculations difficult to follow [37].

12.2 An Example

We return now to the idea of a graph scheme which was introduced in Ch. 6 for the computation of the kth binary digit s_k of the sum of two numbers given in the binary form: –

$$\dots a_2 a_1 a_0 + \dots b_2 b_1 b_0,$$

where arbitrarily many digits 0 can stand left to the last digit 1. One has to take into consideration that s_n for any n depends not only on a_n and b_n but also on the remainder r resulting from the already executed addition of the digits to the right.

The associates of the vertices will be denoted in the same way as it is customary in (non-exact) practice. They will be written into squares and hexagons, which represent the mathematical and logical vertices, respectively. In the mathematical vertices statements of the form

$$c \Rightarrow v$$

[36] See R. Péter: *Mathematische Fassung der sogenannten „Entscheidungs-Tabellen"*, Acta Cyb. **2** (1973), pp. 89—108.

[37] See R. Thurner: *Entscheidungs-Tabellen,* Düsseldorf (1972), with the references given there.

figure, meaning that a variable v has to be given the value c (disregarding the fact that possibly v has already been given a value earlier). In the logical vertices, questions of the form

$$c = b?$$

are written.

These are obtained in the following way from the mathematical and logical functions associated with the vertices of the appropriate graph scheme.

We compute the digits $s_0, s_1, \ldots$ of the sum, step by step, until we reach s_k. We introduce auxiliary variables n, r, s to denote the step number, the current remainder, and the current digit of the sum, which will vary in the course of the computation. When we say that s_0 is "computed in step 0" (where of course the remainder is 0) we mean that both n and r have to take the initial value 0. For s we can also take the irrelevant initial value 0.

The input vertex of the graph scheme has to be a mathematical vertex, with which, since the initial data form the sequence

$$(k, a_0, \ldots, a_k, b_0, \ldots, b_k),$$

the mathematical function

$$\alpha_1(k, a_0, \ldots, a_k, b_0, \ldots, b_k) = (k, a_0, \ldots, a_k, b_0, \ldots, b_k, 0, 0, 0)$$

is associated. Initially, in the flow chart, r and n are declared to have the value 0, for this

$$0 \Rightarrow r$$

will be written in the input vertex, and the simple edge starting from here will lead to another mathematical vertex with

$$0 \Rightarrow n.$$

Here the procedure branches according as the remainder is 0 or not. Hence there must follow a logical vertex. In the case of the graph scheme, this is associated with the relation

$$B_1(k, a_0, \ldots, a_k, b_0, \ldots, b_k, n, r, s) \equiv r = 0,$$

in the case of the practical flow diagram with the question

$$r = 0?$$

the edges starting at this logical vertex, according as the answer is "yes" or "no", will be marked by T and F in the graph scheme and by Y and N in the flow chart.

Then in both cases another branching follows, according as $a_n = b_n$ or not. Thus we have a logical vertex again, with the relation

$$B_2(k, a_0, \ldots, a_k, b_0, \ldots, b_k, n, r, s) \equiv a_n = b_n$$

in the graph scheme, and with the question

$$a_n = b_n?$$

in the flow chart.

In the next step there appears an auxiliary variable s which has to be given a value 0 or 1. In the graph scheme, this is accomplished by the functions

$$\alpha_i(k, a_0, \ldots, a_k, b_0, \ldots, b_k, n, r, s) = (k, a_0, \ldots, a_k, b_0, \ldots, b_k, n, r, 0),$$

$$\alpha_j(k, a_0, \ldots, a_k, b_0, \ldots, b_k, n, r, s) = (k, a_0, \ldots, a_k, b_0, \ldots, b_k, n, r, 1),$$

and in the flow chart, by the statements

$$0 \Rightarrow s,$$

$$1 \Rightarrow s.$$

This process continues repeatedly in the same way. In the graph scheme $2k+6$-term sequences

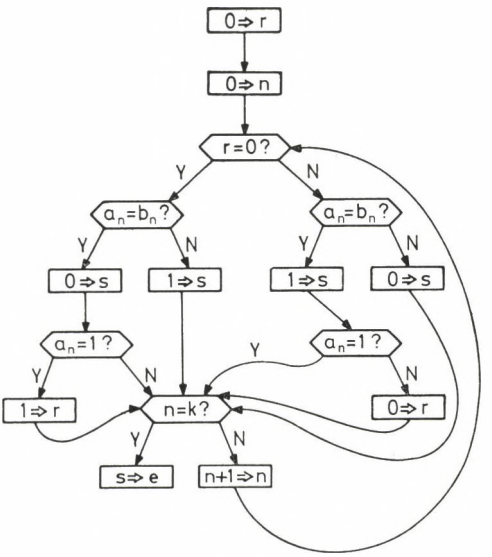
$$(k, a_0, \ldots, a_k, b_0, \ldots, b_k, n, r, s)$$

will occur, with the exception of the output vertex, where the 1-term sequence s_k is obtained as the function value. In the flow chart, this is expressed (after introducing an auxiliary variable e for the result) by the statement

$$s \Rightarrow e.$$

Meanwhile, before each step in the computation, the question is put whether $n=k$? If so, one proceeds to the output. If not, then n is increased by 1, and one goes back to the first branching point.

It is easy to see that the flow chart constructed according to the above instructions does compute the required digit s_k of the sum: –



12.3 Changing Flow Charts into Decision Tables

The flow chart belonging to the above simple problem is nevertheless still rather complicated. So the parts consisting of several logical vertices will be replaced by decision tables.

A decision table (or simply a table) is divided into four quadrants as follows: –

I			II		
III			IV		

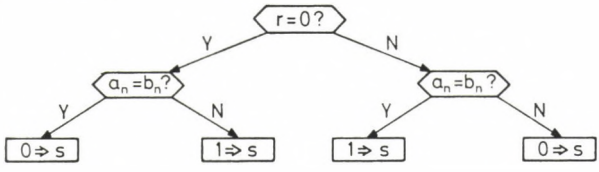
In quadrant I different questions, and in quadrant III different statements will be indicated. The other two quadrants II and IV will be divided into a certain number of columns. In the upper part (that is in II) every column contains a variation of Y, N and the "empty" symbol. In the lower part (that is IV) every column contains a variation of X and the "empty" symbol. To explain the meaning of such tables, let us consider an example. Suppose that I and III, and one of the columns are as follows: –

F_1	Y	
F_2		
F_3	N	
A_1		
A_2	X	
A_3	X	
A_4	X	

This means that if the answer to question F_1 is "Yes", and to F_3 is "No", then (independently of the answer to F_2) the statements A_2, A_3 and A_4 have to be executed.

Clearly the upper halves of two columns cannot be identical, because then, if we want to avoid contradictions, their lower halves would also be identical. Hence one of them would be superfluous.

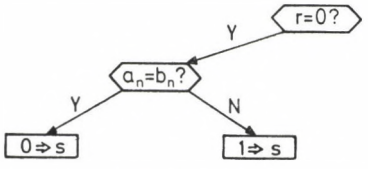
Let us consider a part of the flow chart given in section 12.2, containing several connected logical vertices.



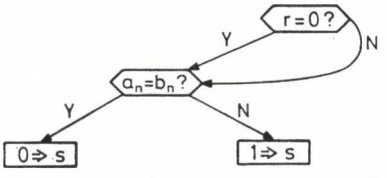
This can be replaced by a table, by first traversing all possible directed paths of edges starting at the initial vertex. These will be called "lines". The questions found along the way (each one occurs only once) are written in I and the statements are written in III. For every line, a column is filled in as follows: – The row of a question is empty if the question does not occur along this line, Y and N is written if the question occurs and the edge on the line following the corresponding logical vertex is marked by Y or N, respectively. Finally, for every statement X; or, nothing, is written, according as a vertex with this statement is traversed by the line or not.
By always choosing the leftmost line first, we obtain the following table in our example: –

$r=0?$	Y	Y	N	N
$a_n=b_n?$	Y	N	Y	N
$0\Rightarrow a$	X			X
$1\Rightarrow s$		X	X	

Now if we wanted to reconstruct the above subgraph from this table, this could not be done in a unique way. From the first two columns we can still uniquely recover the part



as well as the fact that the line belonging to the third column starts with the edge N at the initial vertex. This edge however, could lead to the middle vertex with the question "$a_n=b_n$?", and further it could lead along the edge Y starting there, which contradicts the final statement of the actual third column: –



Therefore it is advisable to drop the requirement that the questions in I and the statements in III are different. We consider the rows of the table as belonging to the different vertices rather than to different questions or

statements. In what follows I will often say "points" instead of vertices. Then the table belonging to the above subgraph looks as follows: –

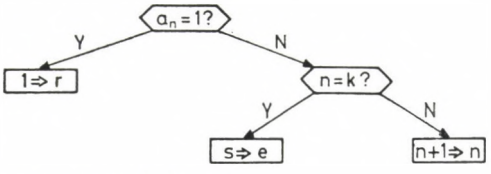
T_1^*:

$P_{1.1}$	$r=0?$	Y	Y	N	N
$P_{1.2}$	$a_n=b_n?$	Y	N		
$P_{1.3}$	$a_n=b_n?$			Y	N
$P'_{1.1}$	$0\Rightarrow s$	X			
$P'_{1.2}$	$1\Rightarrow s$		X		
$P'_{1.3}$	$1\Rightarrow s$			X	
$P'_{1.4}$	$0\Rightarrow s$				X

where $P_{1,i}$ and $P'_{1,i}$ denote the ith logical vertex and the ith mathematical vertex, respectively, which are used in the construction of table T_1^*. From this, the subgraph can be reconstructed in only one way.

12.4 Systems of Tables

Considering the whole graph of section 12.2 we see that the continuations of the subgraph, dealt with in section 12.3 again lead to logical vertices. Starting from one of these vertices (for example the one on the left), let us consider again the subgraph consisting of those lines, which from here lead to the first mathematical vertex or (if this were the case) to a vertex already encountered. (If the mathematical endpoint of a line is followed by further mathematical vertices, then the line has to be extended to the first new logical vertex or return to a point already encountered, respectively.) Thus we obtain the subgraph

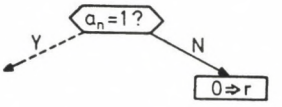


to which the following table belongs

T_2^*:

$P_{2.1}$	$a_n=1?$	Y	N	N
$P_{2.2}$	$n=k?$		Y	N
$P'_{2.1}$	$1\Rightarrow r$	X		
$P'_{2.2}$	$s\Rightarrow e$		X	
$P'_{2.3}$	$n+1\Rightarrow n$			X

Here all points are different from the points of T_1^*.
Now only one logical vertex remained in the graph of section 12.2. Starting with this, similarly we can deduce the following subgraph: –



The dotted edge, marked by Y, leads to the logical point $P_{2,2}$ of the above subgraph corresponding to T_2^*. Its continuation is the part of this subgraph starting at this point. This corresponds to the following "subtable" of T_2^*: –

$T_{2.2}^*$:

$P_{2.2}$	$n=k?$	Y	N
$P'_{2.2}$	$s\Rightarrow e$	X	
$P'_{2.3}$	$n+1\Rightarrow n$		X

Therefore, it is convenient to add statements of the form "go to $T_{i,j}$" as "exits" from tables, which require the execution of the subtable of T_i starting at the point $P_{i,j}$.
Then the table belonging to the last subgraph looks as follows: –

T_3^*:

$P_{3.1}$	$a_n=1?$	Y	N
$P'_{3.1}$	$0\Rightarrow r$		X
	go to $T_{2.2}^*$	X	

where $P_{3,1}$ and $P'_{3,1}$ are different from the points of both T_1^* and T_2^*.
The addition of an "exit part" to the table (which is not assumed to belong to the "lower part" of the table) is also useful because it shows where the last edge belonging to a column must lead to.
Putting into T_3^* the augmented form $T_{2,2}$ instead of $T_{2,2}^*$, we obtain

T_3:

$P_{3,1}$	$a_n=1?$	Y	N
$P'_{3,1}$	$0\Rightarrow r$		X
	go to $T_{2,2}$	X	X

We have similar augmented versions of T_1^* and T_2^* as well.

The mathematical points on the line leading from the input vertex to the first logical point still do not occur in any of the tables. For these we construct the following table with a single column, and empty upper part: –

T_0:

$P'_{0.1}$	$0 \Rightarrow r$	X
$P'_{0.2}$	$0 \Rightarrow n$	X
	go to T_1	X

where T_i for $i>0$ means the subtable $T_{i,1}$ of T_i.
Thus the following system of tables is associated with the graph of section 12.2: –

T_0:

$P'_{0.1}$	$0 \Rightarrow r$	X
$P'_{0.2}$	$0 \Rightarrow n$	X
	go to T_1	X

T_1:

$P_{1.1}$	$r=0?$	Y	Y	N	N
$P_{1.2}$	$a_n=b_n?$	Y	N		
$P_{1.3}$	$a_n=b_n?$			Y	N
$P'_{1.1}$	$0 \Rightarrow s$	X			
$P'_{1.2}$	$1 \Rightarrow s$		X		
$P'_{1.3}$	$1 \Rightarrow s$			X	
$P'_{1.4}$	$0 \Rightarrow s$				X
	go to T_2	X			
	go to $T_{2.2}$		X		X
	go to T_3			X	

Here we have

$T_{2.2}$:

$P_{2.2}$	$n=k$?	Y	N
$P'_{2.1}$	$s \Rightarrow e$	X	
$P'_{2.2}$	$n+1 \Rightarrow n$		X
	stop	X	
	go to T_1		X

T_2:

$P_{2.1}$	$a_n=1$?	Y	N	N
$P_{2.2}$	$n=k$?		Y	N
$P'_{2.1}$	$1 \Rightarrow r$	X		
$P'_{2.2}$	$s \Rightarrow e$		X	
$P'_{2.3}$	$n+1 \Rightarrow n$			X
	go to $T_{2.2}$	X		
	stop		X	
	go to T_1			X

T_3:

$P_{3.1}$	$a_n=1$?	Y	N
$P'_{3.1}$	$0 \Rightarrow r$		X
	go to $T_{2.2}$	X	X

The computation procedure is represented by these tables in a somewhat clearer way than by the graph of section 12.2.

It is also important that several people can work on the separate tables. Some of the tables can even be extended (or changed in some other way), without disturbing their connections. The statements "go to T_i" or "go to $T_{i.j}$" then really call for walking – namely to the desk of the person working on table T_i or $T_{i,j}$, respectively. So certain edges of the flow diagram can be represented by such walks.

12.5 Normalizing Flow Charts

It can happen that on a line being used to build a column of a table, two different points are associated with the same question F in such a way that in the corresponding column, the answers to this question are either superfluous or contradictory. However, the basic graph can always be replaced by an other one, for which such situations do not occur. The graph in this respect is said to be normalized.

I will not go into the details of this normalization here.

I have one more remark. If a line returns to the mathematical point $P'_{0,2}$ (to which the edge starting at the input vertex leads), then afterwards the part

$P'_{0.2}$	$0 \Rightarrow n$	X
	go to T_1	X

of T_0 has to be executed. This table is called the subtable $T'_{0,2}$ of T_0 belonging to $P'_{0,2}$. Thus in the exit of a table statements of the form "go to $T'_{i,j}$" can also occur.

12.6 Regular Tables

As in the example above, we can obtain from every normalized flow chart a system of tables giving the same result. I will list here the characteristic properties of such systems of tables, which I will call *regular*. These also reflect the fact that always the leftmost line was chosen for constructing the next column of a table.

(a) A table system consists of finitely many tables without common points

$$T_1, T_2, \ldots, T_n \quad \text{and perhaps } T_0.$$

T_0 (and only T_0, if it occurs) does not have an upper part. Moreover none of the tables has an exit "go to T_0". If T_0 does not occur, then none of the tables has the exit "go to T_1". For every other table T_i, however, there is at least one column exit "go to T_i" perhaps in the form "go to $T_{i,1}$".

(b) As column exits of a table T_m ($m \leqq n$) statements of the form

$$\text{stop}, \quad \text{go to } T_k, \quad \text{go to } T_{i,j}, \quad \text{go to } T'_{i,j},$$

can serve.

(c) The exit "stop" belongs to only one table column.
Next we describe what is meant by saying that the tables of the system are "regular". This concerns both the upper and lower parts of the tables and requires that the following properties be satisfied:
(d) In the first row, belonging to the first point of the table, there are no empty places, since every line used for the construction of the table starts at this point.
(e) In the upper part of the first column the non-empty symbols must all be Y's and follow each other without a gap. In the last column, and only in the last, no Y smymbol occurs.
(f) (1) For every appropriate i the contents of the $(i+1)$th column coincides with the contents of the ith column, up to the last Y symbol of the latter, instead of which N occurs in the $(i+1)$th column. (2) The first non-empty symbol after this N in the upper part of the $(i+1)$th column belongs to the first such row, in which none of the 1st through to the ith columns contain a Y or N symbol, since after a line branches from the earlier one only new points are traversed by the new line. In the first portion of the upper part of the $(i+1)$th column the non-empty symbols — which are all Y's follow each other without a gap.
(g) The questions really to be considered in a column (that is the ones belonging to non-empty symbols) as well as the questions following these after possibly empty places in the lower parts of other tables, are all different. This follows from the normalization of the graphs mentioned in section 12.5. This property also applies to the exits.
(h) In the lower part of every column, the X symbols follow each other without a gap, in the first column from the first row on, for every appropriate i. On the other hand, in the $(i+1)$th column they follow from the row just below the row in which the last X occurred in the ith line because the mathematical points of lines used to construct the columns are all different.

12.6.1 Subtables

The subtables have to be constructed as follows: –
The subtable $T_{i,j}$ of T_i is obtained by omitting the first $(j-1)$ rows, then all the columns which are empty in the jth row, and finally all the rows in which after this row none of the symbols Y, N, X remain.
In constructing $T'_{i,j}$, one has to do the same, after the whole upper part of T_i is omitted. Here of course, the "jth row" means the jth row of the remaining part of T_i. Thus $T'_{i,j}$ has always only a single column.

For example, with the particular table T_1 of section 12.4 we have

$T_{1.3}$:

$P_{1.3}$	$a_n = b_n$?	Y	N
$P'_{1.3}$	$1 \Rightarrow s$	X	
$P'_{1.4}$	$0 \Rightarrow s$		X
	go to $T_{2.2}$		X
	go to T_3	X	

and

$T'_{1.3}$:

$P'_{1.3}$	$1 \Rightarrow s$	X
	go to T_3	X

Clearly, $T'_{0,1}$ is the table T_0 itself, and for every $i \neq 0$ $T_{i,1}$ is equal to T_i. From a regular table system it is easy to construct a flow chart leading to the same result.

12.7 Turning Tables into Regular Tables

Tables occurring in practice, and in the literature, are in general not regular. It is important, however, to be able to turn these into graph schemes as well, since the latter, as is shown in section 7.3, can immediately be translated into certain programming languages.

This can be achieved by turning these systems of tables into regular systems. If the connections between the tables of an arbitrary table system are given in a reasonable way, they can always be formulated by means of the exits introduced above.

Requirement (c), which is the one most often violated in practice, can also be dropped. If there are several points in the graph, from which no edges originate, this can only be a fragment of a graph scheme (a graph scheme serving the same ends can always be constructed, however), but the effect of such a fragment can also be translated into programming languages.

For similar reasons, the requirement that no edge may lead to a certain point can also be dropped.

In any case, we have to restrict ourselves to table systems satisfying property (g) of section 12.6.

According to the above, we do not have to worry any more about the exits from the tables. In what follows, however, we show that every table, containing different questions and statements only, which does not contain two columns with identical upper parts (even "implicitly" in a sense to be clarified soon), can be respresented by a regular table having the same effect.

The lower part of any table T_i can easily be made regular. Let us assume that the number of X symbols in the first column is x_1, in the second x_2, ..., in the last x_t. Then we take new mathematical points

$$P'_{i,1}, \ldots, P'_{i,x_1}, P'_{i,x_1+1}, \ldots, P'_{i,x_1+x_2}, \ldots, \ldots, P'_{i,x_1+x_2+\ \ldots\ +x_t}.$$

In the same order, we take rows corresponding to these points instead of the earlier rows of the lower part of T_i. Then the X symbols of the first column, together with the corresponding statements, are put one by one into the rows belonging to $P'_{i,1}, \ldots, P'_{i,x_1}$. The X symbols of the second column together with the corresponding statements (among which earlier ones might occur) are put by one into the rows belonging to $P'_{i,x_1+1}, \ldots, P'_{i,x_1+x_2}$, and so on.

This makes (h) of section 12.6 valid, and then it remains to deal with the upper parts of tables.

Concerning the upper parts of table columns, it will be useful to consider the empty symbol in such a way that the statements in the columns are independent of the corresponding question, that is they yield the same for both answers "yes" and "no". Therefore it is usual to split each column containing an empty symbol into two, which differ from the original only in that the first replaces the empty symbol by Y, the second by N. It could happen, however, that in doing this the upper part of a new column coincides with the upper part of an old one.

Therefore the essential difference between the upper parts of two columns must be understood as the existence of at least one row in which one of the columns has Y, and the other has N. If this holds, then the table does not have two columns with the same upper parts even implicitly.

Furthermore, it is also customary to add new columns to a table with a new statement called "error", to emphasize that the variation of answers to the questions given in this column is not appropriate for our purposes. Instead of applying the new statement "error" it would serve the same ends to prescribe in the exit the return of the last edge belonging to the column to its initial point, thereby producing an infinite cycle. This would show then that the result of the procedure represented by the table is undefined for the corresponding variation of answers.

With the above splitting and adding of new columns, the upper part of any table can be transformed in such a way that the upper parts of the columns will yield all the possible variations of the symbols Y and N. If we have n questions, their number is 2^n.

If these variations are arranged in such a way that one of them precedes the other if and only if, at the first place where they differ, it contains Y (and the other N), and the columns are arranged accordingly, this will precisely correspond to the leftmost choice of the lines according to which the columns of the table corresponding to a flow chart were constructed. We still have to ensure the validity of requirement (f) (2) of section 12.6, that is the reflection of the fact that, after every choice of an edge starting at a branching point only new points will be traversed by the corresponding line.

12.7.1 An Example

Let us consider as an example the case of 3 questions F_1, F_2, F_3. The upper part of the table containing all the variations of answers in the above order is the following:

F_1	Y	Y	Y	Y	N	N	N	N
F_2	Y	Y	N	N	Y	Y	N	N
F_3	Y	N	Y	N	Y	N	Y	N

In order to satisfy (f) we build from this the following upper table: –

F_1	Y	Y	Y	Y	N	N	N	N
F_2	Y	Y	N	N				
F_3	Y	N						
F_3			Y	N				
F_2					Y	Y	N	N
F_3					Y	N		
F_3							Y	N

This is already the upper part of a regular table.

12.8 Normal Systems of Tables

In the different particular cases, it is not always necessary to fill in all the empty places, or to form all possible variations. In actual practice, one strives for the simplest possible transition to a corresponding flow chart.
Let us consider for example a decision table with applications to company organisation, which is given on p. 19 of the book quoted in footnote [37]. Using the notation

$$F_1, F_2, F_3, F_4 \quad \text{and} \quad A_1, A_2, A_3, A_4$$

for the questions and statements, respectively (whose meaning is irrelevant to our investigations), this can be written as follows: –

F_1	Y	Y		N
F_2	Y	N	N	Y
F_3				N
F_4		Y	N	
A_1	X			
A_2				X
A_3		X		
A_4			X	

Now we have to examine the properties given in section 12.6.
Firstly, because of the empty place in the first row, (d) is not satisfied. Therefore the third column has to be split into two (we could have switched the first two rows instead): –

F_1	Y	Y	Y	N	N
F_2	Y	N	N	N	Y
F_3					N
F_4		Y	N	N	
A_1	X				
A_2					X
A_3		X			
A_4			X	X	

In the fourth column, no Y occurs, hence (e) is not satisfied. This can be remedied by switching the last two columns: –

F_1	Y	Y	Y	N	N
F_2	Y	N	N	Y	N
F_3				N	N
F_4		Y	N		
A_1	X				
A_2				X	
A_3		X			
A_4			X		X

In the 4th and 5th columns the last requirement of (f) is not satisfied, namely that after a branching point, only Y edges can occur on the initial part of a line belonging to a column. Therefore a new row with the statement "error" has to be added: –

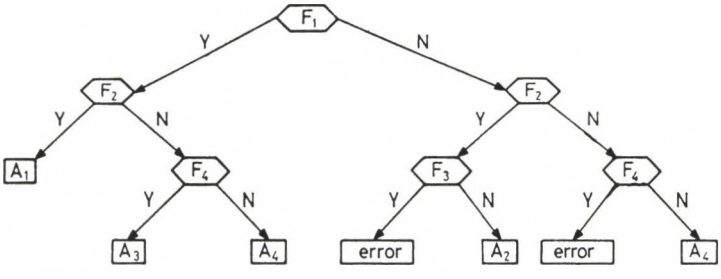
F_1	Y	Y	Y	N	N	N	N
F_2	Y	N	N	Y	Y	N	N
F_3				Y	N		
F_4		Y	N			Y	N
A_1	X						
A_2					X		
A_3		X					
A_4			X				X
error				X		X	

Finally, the non-empty symbols of all the columns have to be placed into the rows as prescribed by (f) and (h), together with their corresponding question and statement symbols: –

P_1	F_1	Y	Y	Y	N	N	N	N
P_2	F_2	Y	N	N				
P_3	F_4		Y	N				
P_4	F_2				Y	Y	N	N
P_5	F_3				Y	N		
P_6	F_4						Y	N
P_1'	A_1	X						
P_2'	A_3		X					
P_3'	A_4			X				
P_4'	error				X			
P_5'	A_2					X		
P_6'	error						X	
P_7'	A_4							X

This is already a regular table with 7 columns and not $2^4 = 16$ columns which would be needed if all the possible answers to the 4 questions were to be used.

From this table, the lines of the corresponding flow diagram (starting at P_1) can be read off immediately. Thus we obtain



12.8.1 Comparison with Partial Recursive Functions

The transition from a graph scheme to an ordinary flow chart, which in section 12.2 was illustrated with an example, can of course be reversed. Instead of the questions and statements at the points, we can return (perhaps after a suitable coding) to logical and mathematical functions.

According to the above, every normal scheme (a notion which was introduced in section 7.5) can be represented by a normal system of tables having the same effect. By the latter, we mean a system of regular tables such that, in quadrant I. of each table initial relations, and in quadrant III. initial functions are contained. Moreover the converse of this statement is also valid.

Consequently the functions definable by normal systems of tables coincide with the partial recursive functions, and thus with the machine computable functions, since for the functions computable by normal schemes this was shown in Chapters 6 and 7.

Index

FROM THE REVIEWS
OF THE GERMAN VERSION

REKURSIVE FUNKTIONEN IN DER KOMPUTER-THEORIE

by

RÓZSA PÉTER

In German — 1976 — 190 pages — 17×25 cm
Cloth — ISBN 963 05 0509 6

"... This book is distinguished by its unique style, every researcher in recursive function theory who appreciates Peter's by now classical work on "Recursive Functions" is familiar with. The lines of thought are followed in a truly genetic way, illustrated and motivated by numerous examples. It is this that makes the book an excellent text even for beginning students of computer science. It is a pleasure to read this book, which certainly deserves a more widespread distribution than is assured by the fact that up to now it is available in German only."

ZENTRALBLATT FÜR MATHEMATIK, Berlin

"... This book is a valuable first course in the field of theoretical computer science..."

REVUE ROUMAINE DE MATHÉMATIQUES PURES ET APPLIQUÉES, Bucarest

Distributors:
KULTURA
H-1389 Budapest, P.O.B. 149

ISBN 963 05 2257 8

253.

(A művész köszönőlevele)

Budapest, 1961. június 12.

Hárs György elvtársnak
Budapest

Népszabadság Szerkesztősége
Kedves Hárs Elvtárs!
A Népszabadságban megjelent rólam szóló, meleg, baráti hangú írásodat ezúton köszönöm meg.

Szívélyes üdvözlettel

254.

(Az albérlő levele)

A „raktár"-helyiséget egyéb hely hiányában nem áll módomban biztosítani. A „raktár"-helyiséget egyébként megegyezésünk szerint vettem annak idején igénybe, mint az albérlemény tartozékát, tehát a *lakbért* ezután is fizettem! Felhívom nb. figyelmét, hogy ha hasonló „főbérlői" allürökkel kívánja megzavarni azt a csendet, melyet *egyedül* az én figyelmességem (elsősorban korára való tekintettel) és türelmem teremtett meg, úgy kénytelen leszek olyan – jogomban álló – retorziókkal élni, amelyekből (az eddigi helyzethez viszonyítva) *csak kára, vesztesége, bosszankodása* stb. stb. fog származni.
Örülnék, ha ezt *komolyan* megfontolná és békében hagyna élni.

[Budapest, 19]61. IX. 8. G.

255.

(A művész levele)

Kállai Gyula miniszterhelyettes [!] elvtársnak

Kedves Kállai barátom!
Nem panaszkodni akarok, csak a tényeket közölni.
Rossz helyzetben vagyok, bár egész évben dolgoztam, tíz új plasztikát csináltam és kiállításra készülök. Igaz, hogy van kétezer forint nyugdíjam; ez azonban a megélhetésemhez sem elég, és a kiállításra készülődés minden filléremet felemésztette. Dolgozni akarok, és nem engedem, hogy élve eltemessenek. Hacsak kevés anyagot vásárolok, akkor már felborul az amúgy is nagyon labilis anyagi helyzetem. Már ideje volna ruhát csináltatnom, kopott vagyok; nem hiszem, hogy vonzó látvány egy Kossuth-díjas rossz ruhában. De ezt nem bántam volna, a munkához volt szükségem pénzre. Szerényen ezer forintot kértem a Képzőművészeti Alaptól. Szilárd igazgató azonban nem írta alá a kiutalást azzal az indokkal, hogy van kétezer forint adósságom és nem vagyok kereső művész . . . Ez igaz, mert nem engednek keresni, bár dolgozom és nem megvetendő alkotásokat készítek. Ha nem tudnám, hogy nálam érdemtelenebbeknek sok-sok ezer forint adósságuk van az Alapnál, nem szólnék Szilárd igazgató kemény döntése ellen. Nekem ebben a mi népi államunkban nincs lehetőségem levegőhöz jutni? Pedig ezért a rendszerért én tettem is valamit, egy egész élet munkájával küzdöttem érte. Most készítettem két kis portrét, Bartókról és Lisztről, melyeket a Kerámia Szövetkezet szeretne sokszorosítva árulni. Ez megadná azt a kevés összeget, ami szükséges ahhoz, hogy szerény életmódom mellett is meglevő pénzzavarom megszűnjék. Már előre félek, mert

a Képzőművészeti Alap zsürije elé kell küldenem, oda, ahol tíz éve minden munkámat visszautasítják – nálam sokkal kisebb szobrászok. Pedig ez a két fej szocialista realista, teljesen érthető művészeti nyelven van megalkotva. Tíz év: nem vagyok fiatalember, életem utolsó termékeny esztendeit rabolják el tőlem.
Nagyon kérlek, intézkedj, hogy segítsenek rajtam. De ne engedd, hogy úgy tegyenek, mint tavaly, amikor segítséged következtében vettek tőlem egy szobrot, de a tízezer forintból hatezret egyösszegben adósságra levontak. Igaz, hogy most csak kétezer forint adósságom van.
Nagyon kérlek, tégy valamit, hogy ebből a keserves elakadásból kikerülhessek.
Remélem, hogy helyt adsz kérésemnek, hisz tudod, hogy nem fordulok hozzád, csak ha nagy bajban vagyok.

Elvtársi üdvözlettel régi híved és barátod

256.

(Levél a Magyar Tudományos Akadémia Bartók Archivumától)

Budapest, 1962. január 18.

Tisztelt Művész Úr!
Minthogy tudomásunk van az Ön Bartók Béla-szobráról, és mivel intézetünk figyelmét szeretnők kiterjeszteni a Bartókkal kapcsolatos képzőművészeti alkotásokra is, rendkívül lekötelezne bennünket, ha a szobor megtekintésére módot nyújtana. Legcélszerűbb lenne, ha a művet az Archívum érdeklődő munkatársainak be tudná mutatni, illetve, ha ifj. Bartók Béla úr megtekinthetné azt. Mindenesetre kérjük, hogy a lehetőségek tisztázása véget szíveskedjék bennünket telefonon megkeresni a délelőtti vagy a koradélutáni órákban (161–522).
Szíves válaszát várva maradunk teljes tisztelettel

Dr. Szabolcsi Bence
igazgató

257.

(Levél a Művelődésügyi Minisztériumból)

Értesítem, hogy a Bartók Béláról készített portréját bronzban 2500 Ft értékben megvásároltam.
Jelen levelemmel egyidejűleg a Képzőművészeti Alapnál intézkedtem, hogy a tiszteletdíjat levonás nélkül fizessék ki az Ön részére.

Budapest, 1962. február 27.

Szentesi Antal s. k.
osztályvezető h.

258.

(Levél az Országgyűlési Könyvtár igazgatójától)

Kedves Dezső!
Az Élet és Irodalom e heti számában megjelent a Független Magyarországért-plakátod fényképe. Amennyiben még nem küldtek honoráriumot érte, úgy je-

lentkezzél a szerkesztőségben. A rendelkezésre bocsátott fényképet nekem fogják visszaküldeni, és majd eljuttatom Hozzád.
Sajnálom, hogy műteremkiállításodat nem tekinthettem meg, pont akkor előadást tartottam.
Az Ady-fejet illetően megbízottad jelentkezhet a napokban.

Budapest, 1962. március 22.

Szívélyes üdvözlettel
Vértes György

259.

(Levél a Magyar Forradalmi Munkás-Paraszt Kormány Elnökhelyettesének Titkárságától)

Kedves elvtárs!
Kállai elvtárshoz írt levelét megvizsgáltattuk a Művelődésügyi Minisztériummal. A Képzőművészeti Osztály a Bartók-szobor bronz példányát megvásárolja, s a Képzőművészeti Alap igazgatójával megbeszéli, hogy a vásárlási összegből ne vonja le az Ön tartozásait.
Egyébként Kállai elvtárs a kiállítása megnyitójára szóló meghívót megkapta, de betegsége miatt azon nem tudott részt venni.

Budapest, 1962. március 24.

Elvtársi üdvözlettel:
Nagy László

260.

(Levél a Jókai Színház igazgatójától)

Budapest, 1962. április 10.

Kedves Bokros-Bierman elvtárs!

Műtermi kiállítására szóló meghívóját örömmel vettem kézhez, de – sajnos – abban az időben nem voltam Pesten, külföldön léptem fel és így megtisztelő meghívásának nem tehettem eleget.
Abban a reményben, hogy alkalmam lesz az utóbbi időben készült műveit látnom, maradok művészetének és Önnek

tisztelő híve
Keres Emil

261.

(Üzenet az albérlőnek)

Budapest, 1962. április 11.

Furcsa teremtmény az ember. Ha kell, számolni sem tud. 76,– Ft-nak nem 30,– Ft a 70%. Igaz, Ön 23 napot nem töltött lakásában – de gondoskodott arról, hogy másik két személy pótolja Önt távollétében.

Furcsa teremtmény az ember! Nem mindig buta, csak néha azért korlátolt. A 70%-ot pótolja ki, s csak azt mondom: furcsa teremtmény az ember!

B. B.

262.

(Levélfogalmazvány az Újkori Történeti Múzeum igazgatójához)

[Budapest, 1]962. V. 18.

Tisztelt Igazgató elvtárs!

Ismerve az Ön által vezetett intézet érdeklődési körét, úgy gondolom, helyesen teszem, ha felhívom a figyelmét két – tulajdonomban levő munkámra.
Áchim András-emlékmű terve az egyik, melyet 1924-ben készítettem és 1925-ben egy a Mentor-beli kiállításon mutattam be.
A másik az Egységfront c. plakettem. 1930-ban készült.
Amennyiben érdekli Önöket a nevezett két munkám, kérem, keressenek fel műtermemben.
Jövetelük időpontját előzőleg egy lapon közöljék velem.

1962. V. 18. postára téve.

263.

(Levélfogalmazvány Csehszlovákiába)

Kedves Dömötör Teréz!

Azért fordulok Önhöz levelemmel, hogy a kossuthi mártír-emlékműről ittjártakor tudomásomra hozott értesüléseit levél útján újra kikérjem. Nagyon kérem, írjon le pontosan mindent, amit ebben az ügyben tud.
Lehetséges ugyanis, hogy rövidesen személyesen is felkeresem a szóban forgó emlékművet.

1962. V. 18. postára téve.

264.

(A művész értesítése)

Budapest, 1962. augusztus 23.

Kállai Gyula elvtársnak
Budapest

Kedves Gyula!

Az elmúlt tél folyamán ígéretet tettem Neked, hogy készítek számodra egy kisplasztikát. Erre mostanában került sor: elkészítettem részedre egy Bartók-bronz kisplasztikát, melyet szeretnék Neked személyesen átadni. Kérlek, közöld velem, hogy mikor és hol adhatnám ezt át Neked.
Válaszod várva, vagyok

elvtársi üdvözlettel

265.

(A bronzöntő levele)

Igen tisztelt Művész Úr!
Elnézést kérek a zavarásért, valószínűleg el tetszett felejtkezni rólam, ti. a kis Bartók-fejekért még 450 Ft járandósága van a Művész úrnak. Nagyon megkérem, szíveskedjék postafordultával elintézni.
Még egyszer elnézést kérek.

B[uda]pest, 1963. aug. 26.

maradtam teljes tisztelettel
Baumgartner József

266.

(Ügyvédi jegyzőkönyv)

Tényvázlat

Felvéve az 1. sz. ÜMK-ban 1963. október hó 8. napján, Bp. V. ker. Kecskeméti u. 13. II. em. dr. Krámer István ügyvéd által.
Megjelenik Bokros Birman Dezső nyugdíjas szobrászművész és előadja a következőket:
Ismeretlen tettes ellen feljelentést tettem a XIII. ker. Teve u. 6. rendőrségen, mert egy nagyértékű szobromat ellopták a lakásomból.
Gyanakodom az albérlőmre, baráti körére, ill. a feljelentésben is szereplő tanítványomra.
Albérlőm egy ízben már lopott tőlem plasztelint, ezzel az üggyel már fordultam a rendőrséghez. Akkor ő a plasztelint vissza is adta.
Kérem, hogy nézzenek utána a rendőrségen: hogyan áll ez az ügy, és az esetleges lépéseket megtenni szíveskedjenek.

Ügyfél tudomásul veszi, hogy a munkadíj a későbbiekben, az eljárások mennyiségétől függően, ill. a tevékenység minősége megállapítása után lesz megállapítva.
Kéri ügyintézőül kijelölni dr. Krámer István ügyvédet.
Tudomásul veszi, hogy 10 Ft illetékbélyeget a meghatalmazásra le kell róni.

Kmf.

267.

(Megállapodás a Napbanéző című szoborra vonatkozólag)

Megállapodás

Mely köttetett egyrészről Bokros Birman Dezső szobrászművész, másrészről Laczkovich Alice között.
Bokros Birman Dezső szobr. m. Napbanéző c. szobra 2.20-as méretben való elkészítéséért, a szobor gipszben való átadásáért Laczkovich Alice 14 000, azaz Tizennégyezer Ft-ot kap fent nevezett Bokros Birman Dezső szobr. m.-től. A fizetés három részletben történik. Az első részlet, 5000, azaz Ötezer Ft a meg-

állapodás aláírásakor, a második részlet, 4000, azaz Négyezer Ft az agyagszobor elkészülésekor, a harmadik részlet, 5000, azaz Ötezer Ft a szobor gipszben való átadásakor történik.

B[uda]p[est,] 1964. szept. 23.

268.

(Levél a Magyar Nemzeti Galériától)

Kedves Mester!
Engedd meg, hogy hetvenötödik születésnapodon a Magyar Nemzeti Galéria dolgozói nevében sok szeretettel köszöntselek.
A többi magyar művészettörténészhez hasonlóan a Galériában működők is meghatott tisztelettel gondolnak Rád ma, amikor Benned egyszemélyben üdvözölhetik az élő hagyományt és az eleven alkotó géniuszt, s mint fiatalabb kortársak köszönhetik meg Neked munkásságod szép eredményeit, képzőművészetünk maradandó termésének gyöngyszemeit.
Azt kívánjuk valamennyien, hogy jó egészségben folytasd áldásos tevékenységedet, további remekművekkel gyarapítsd hazánk kulturális kincstárát.

Budapest, 1964. november 19.

Baráti üdvözlettel híved
dr. Pogány Ö. Gábor
főigazgató

269.

(Levél a Hazafias Népfront XIII. ker. Bizottságától)

Budapest, 1965. jan. 15.

Kedves Mester!

Örömmel vettük értesítését, hogy szívesen látja kerületi Népfront Bizottságunk Kisiparos Akcióbizottságának látogatását az Ön műtermében.
A meghívásnak eleget téve, 1965. január 23-án, szombaton du. 4 órakor látogatjuk meg Önt.

Hazafias üdvözlettel:
Garami Győzőné
titkár

270.

(Részlet egy újságcikkből)

Magyar Nemzet
kedd, 1965. január 26.

Bokros Birman Dezső Kossuth-díjas szobrászművész, a Magyar Népköztársaság érdemes művésze, életének 75. évében villamosszerencsétlenség következtében meghalt. Az elhunyt művészt a Magyar Képzőművészek Szövetsége és a Magyar Népköztársaság Képzőművészeti Alapja saját halottjának tekinti. Temetéséről később történik intézkedés.

JEGYZETEK A LEVELEZÉSHEZ

1 Az MTA Művészettörténeti Kutató Csoportjának Adattárában: Művészettörténeti Dokumentációs Központ (a továbbiakban: MDK) C–I–18/578. Nyomtatott űrlapon kiállított anyakönyvi kivonat. Száma: 32/1964. Felül gépírással: „Személyazonossági igazolvány céljára illetékmentes." – Az itt közlésre kerülő különböző dokumentumokban a művész nevét nem egyformán írták. Ezen nem változtattunk. A levelezésben előforduló nevek közül néhányat csak monogrammal jelölünk, anélkül, hogy erre esetenként külön felhívnánk a figyelmet.

2 MDK–C–I–18/498. „Modern Iparművészet Dombormű Vállalat" stb. felírású nyomtatott lapon kiállított, okmánybélyeggel ellátott bizonyítvány.

3 MDK–C–I–18/567. Az „Országos Magy. Kir. Iparművészeti Iskola" okmánybélyeggel ellátott hivatalos bizonyítványa.

4 MDK–C–I–18/59. „Budapest Székesfőváros Tanácsa" nyomtatott felírású hivatalos papír.

5 MDK–C–I–18/63.1–2. Címzés a borítékon: Bokros Birman Dezső szobrászművész úr, Budapest, XIV., Ajtósi Dürer sor 13. Feladó: Márton Ödön, Budapest, II. Bimbó út 5.

6 MDK–C–I–18/570. A Svéd Vöröskereszt magyarországi főmegbízottjának védőlevele. A lap alján Bokros felragasztott fényképe; három körpecsét.

7 MDK–C–I–18/577. Stencilezett űrlap, nyomtatott fejléccel. Ügyiratszám: 221.785.1945. XX. ü.o.786/6. Lent balra: „A kiadvány hiteléül Szabó s. hiv. igazgató".

8 MDK–C–I–18/69.1–2. A „Munkás Kultúrszövetség Országos Központja, Budapest" felírású nyomtatott levélpapíron írt levél.

9 MDK–C–I–18/72. A „Magyar Kommunista Párt Központi Vezetősége Propaganda Osztály" felírású nyomtatott levélpapíron írt levél. Aláírás, körpecsét.

10 MDK–C–I–18/73. Gépelt levél másodpéldánya, pontosabb címzés és aláírás nélkül. (A hátlapon ceruzával, Bokros írásával: „RÁDIÓ FELOLVASÁSRÓL ÍRNI".)
A levél szövegéből kiderül, hogy Bokros testvére a címzett.

11 MDK–C–I–18/75.1–2. „Dr. Gegesi Kiss Pál egyetemi ny. r. tanár" felírású nyomtatott levélpapíron géppel sokszorosított levél. Címzés a borítékon.

12 MDK–C–I–18/530. Ceruzával írt levélfogalmazvány.

13 MDK–C–I–18/76. A Magyar Nemzeti Múzeum régi (1945 előtti) nyomtatott levélpapírján írott levél.

14 MDK–C–I–18/77.1–2. „Magyar Vallás- és Közoktatásügyi Minisztérium" domborított felírású levélpapíron írott hivatalos levél. Ügyiratszám a borítékon: 131.880/1946.VII.

15 MDK–C–I–18/79. Gépelt elszámolástervezet. Egy korábbi (1946. november 28.), MDK–C–I–18/78. lt.sz.-on szereplő laphoz képest módosított, felemelt végösszegű elszámolás.

16 MDK–C–I–18/80. Dr. Gegesi Kiss Pál egyetemi ny. r. tanár nyomtatott levélpapírján géppel sokszorosított levél.

17 MDK–C–I–18/81.1–2. A „Magyar Tájékoztatásügyi Minisztérium, Belföldi Osztály" felírású borítékban hivatalosan küldött levél.

18 MDK–C–I–18/82.1–2. A „Magyar Vallás- és Közoktatásügyi Minisztérium" domborított felírású, címeres levélpapírján írt, géppel sokszorosított levél. Ügyiratszám: 27.799/1947. VII.ü.o. Címzés a levél alján és a borítékon. Balra lent: „A kiadvány hiteléül ..... Irodavezető".

19 MDK–C–I–18/83. Géppel írt levél másodpéldánya. Közelebbi címzés nélkül. A hátlapon több, ceruzával írt feljegyzés.

20 MDK–C–I–18/84.1–2. Géppel sokszorosított meghívó a „Fészek" Művészek Klubja nyomtatott borítékjában.

21 MDK–C–I–18/85.1–2. Dr. Gegesi Kiss Pál egyetemi ny. r. tanár nyomtatott levélpapírján írott levél.

22 MDK–C–I–18/86. Postai levelezőlap. Feladó: Bán Béla, Bp. V., Bajcsy-Zsilinszky út 50. Bán Béla (1909–1972) festőművész.

23 MDK–C–I–18/93. Dr. Gegesi Kiss Pál egyetemi ny. r. tanár nyomtatott levélpapírján írott levél.

24 MDK–C–I–18/87. Dr. Gegesi Kiss Pál egyetemi ny. r. tanár nyomtatott levélpapírján írt levél.

25 Gera Éva tulajdona. „Magyar Kommunista Párt Központi Vezetősége Értelmiségi Osztály, Budapest" felírású nyomtatott levélpapíron.

26 MDK–C–I–18/88.1–2. A „48-as Lánchíd Bizottság" nyomtatott meghívója, eredeti aláírással.

27 MDK–C–I–18/91.1–2. A „Magyar Vallás- és Közoktatásügyi Minisztérium" stencilezett nyomtatványa, eredeti aláírással. Száma: 84.977/1947/VII.

28 MDK–C–I–18/92.1–2. Dr. Gegesi Kiss Pál egyetemi ny. r. tanár nyomtatott levélpapírján írt levél.

29 MDK–C–I–18/94. A „Földmunkások és Kisbirtokosok Országos Szövetsége" nyomtatott levélpapírján írt levél. Aláírás, körpecsét.

30 MDK–C–I–18/95. „Magyar Tájékoztatásügyi Miniszter" felírású nyomtatott levélpapíron írt levél.

31 MDK–C–I–18/98.1–2. „Budapest Székesfőváros Polgármestere" felírású, nyomtatott levélpapíron küldött értesítés. Ügyiratszám: 222.949/1947–XI.ü.o. Balra lent: „A kiadmány hiteléül Bp. 1947. júl. 23. Beniczky Sándor s. hív. igazgató".

32 MDK–C–I–18/97. A „Magyar Vallás- és Közoktatásügyi Minisztérium" hivatalos értesítése. Ügyiratszám: 87.369/1947.VII. Előadó: dr. Borecky László min. titkár. Balra lent: „A kiadmány hiteléül Szilágyi rovatvezető".

33 MDK–C–I–18/99.1–2. „Magyar Közlekedésügyi Minisztérium Sajtószolgálatának Vezetője" felírású nyomtatott levélpapíron. A borítékon ceruzával készült vázlatok a plaketthez.

34 MDK–C–I–18/96. Géppel sokszorosított levél.

35 MDK–C–I–18/100.1–2. „Budapest Székesfőváros Képtára" felírású nyomtatott borítékban küldött levél. Címzése: Bokros Biermann Dezső szobrászművész úrnak, Budapest, VI., Eötvös u. 38.

36 MDK–C–I–18/101. „Magyar Kommunista Párt Központi Vezetősége Értelmiségi Osztály, Budapest" felírású nyomtatott levélpapíron írt levél.

37 MDK–C–I–18/102. A „Magyar Vallás- és Közoktatásügyi Minisztérium Művészeti Ügyosztálya" felírású nyomtatott levélpapíron.

38 MDK–C–I–18/103. „Magyar Kommunista Párt Központi Vezetősége Értelmiségi Osztály, Budapest" felírású nyomtatott levélpapíron írt levél. Aláírás, körpecsét.

39 MDK–C–I–18/104. Gépelt levél, aláírással.

40 MDK–C–I–18/108. Gépelt levél. Címzés: T. Bokros Biermann Dezső úrnak, szobrászművész, Budapest, V., Katona J. u. 28. sz.

41 MDK–C–I–18/110. Géppel írt megállapodás.

42 MDK–C–I–18/111. Aláírás nélküli átvételi elismervény.

43 MDK–C–I–18/113. „Magyar Kommunista Párt Központi Vezetősége Értelmiségi Osztály, Budapest" felírású nyomtatott levélpapír. Aláírás, körpecsét. A megszólítás felett: „Bokros-Bierman elvtársnak, Budapest".

44 MDK–C–I–18/118. Tábori postai levelezőlap. Feladó: Szegi Pál, Bp. II., Branyiszkó út 11/c. Szegi Pál (1902–1958) művészeti író, 1949–1953 között a Szabad Művészet című folyóirat főszerkesztője.

45 MDK–C–I–18/115. Gépírásos levél.
Szalatnai Rezső író, műfordító, irodalomkritikus.

46 MDK–C–I–18/117.1–2. „Magyar Kommunista Párt Központi Vezetősége Értelmiségi Osztály, Budapest" felírású nyomtatott levélpapíron írt levél. Aláírás, körpecsét. Megszólítás helyett: „Bokros-Birmann elvtárs, Budapest".

47 MDK–C–I–18/119. Postai levelezőlap. Ceruzával írt szöveg és címzés. Feladó nélkül, de az „Ubul" aláírás Kállai Ernőre utal. A postabélyegző kelte: 1948. II. 6.
Kállai Ernő (1890–1954) művészeti író, kritikus.

48 MDK–C–I–18/121. Az Expressions elnevezésű genfi galéria ebben a levélben rögzíti Bokros műveinek a galériában történő kiállítási feltételeit.
A levél fordítása:
„Uram,
Művei 1948. február 23–március 4. között galériánkban rendezendő kiállításának feltételeit az alábbiakban rögzítjük:
1. 300,– fr (háromszáz frank), amelyből 150,– fr-ot (százötven frankot) megkaptunk és 150,– fr (százötven frank) részünkre történő befizetése legkésőbb 1948. február 25-ig;
2. közleményt fogunk megjelentetni a Tribune de Genève-ben és a Journal de Genève-ben, és 300 meghívót postán küldünk szét;

3. a kiállítás 1948. február 23-án 11 órakor nyílik; a rendezés költségei önöket terhelik. Fogadja uram, megbecsülésünk kifejezését."

49 MDK–C–I–18/123.1–2. A „Magyar Vallás- és Közoktatásügyi Minisztérium Külföldi Kulturális Kapcsolatok" stb. felírású nyomtatott levélpapíron írott levél. Címzés a borítékon: „M. Désiré Bokros-Birman, Pension Elisa, 12 rue de Chantepoulet. Genève. Suisse."

50 MDK–C–I–18/9.2. Bokros ceruzával írt szövege.

51 MDK–C–I–18/9.1–4. Gépírásos szöveg; a hátoldalon idegen kézírás.

52 MDK–C–I–18/9.3–4. Gépírásos szöveg. Fent Bokros írásával: „Pánnal átolvastatni és magyar nyelvre áttenni."

53 MDK–C–I–18/517.1–2. Az „Union Internationale de Radiodiffusion" nyomtatott borítékjában küldött levél. Postabélyegző nélkül. Címzése: „Monsieur Bokros-Birman, Pension Elisa, Chantepoulet, E. V."

A levél fordítása:

„Kedves Bokros-Birman Úr,

Örülök, hogy a Journal de Genève-ben ma reggel megjelent a cikk (bizonyára látni fogja). Nagyon elégedett vagyok, hogy Rheinwaldnak tetszik az Ady-büszt.

A fotókat adja át Kristóffynak az Illustré részére, vagy adja le nálam egyik nap, ha erre jár. Remélem, hogy a dolognak ez a része menni fog. Ha tudok, holnap felugrok magához a penzióba. Tegnap nagyon el voltam foglalva, egy katonai vizsgálat miatt, én szegény baka!

Bocsásson meg elsietett soraimért és fogadja szívélyes üdvözletemet

Gilbert Trolliet

Ha nem vagyok otthon, ott lehet hagyni a borítékot az ajtóm előtt (a belsőnél), ha a levélszekrény túl kicsi lenne.

Viszem majd Komlós úrnak a Hubaynak szóló levelet. Egy vagy két fényképről is fogok beszélni a Présence részére (le Revue)."

54 MDK–C–I–18/512.1–2. Franciául írt levél magyar fordítása. Gépelt szöveg; javítások tintával.

55 MDK–C–I–18/126.1–2. A „Magyar Vallás- és Közoktatásügyi Minisztérium" dombornyomott felírású levélpapírján, Bokros Birman Dezsőnek Párizsba küldött levél. Ügyiratszám: 245.118/1948.X. Előadó: Dr. Boronkay Antal miniszteri titkár. Címzés a levél alján és a borítékon.

56 MDK–C–I–18/128.1–2. Bokros kézírásos levélfogalmazványa.

57 MDK–C–I–18/129.1–2. A „Magyar Vallás- és Közoktatásügyi Minisztérium Külföldi Kulturális Kapcsolatok" stb. felírású levélpapírján írott levél. Címzés a borítékon: „M. Désiré Bokros Birmann c/o M. Etienne Lelkes, Institut Hongrois, 18 rue Pierre Curie, Paris 5e. France."

58 MDK–C–I–18/927. „ifj. Fischer Tibor tervező és tanácsadó építész" felírású nyomtatott levélpapíron írott levél. A szövegrész indigóval készült. A címzés eredeti gépelésű: Bokros Birmann Dezső úrnak, szobrászművész, Budapest, V., Katona József u. 28.

59 MDK–C–I–18/130.1–2. Sima levélpapíron, írógéppel írt levél.

60 MDK–C–I–18/89. Bokros kézzel írt levélfogalmazványa a Párizsi Magyar Intézet igazgatójához, Lelkes Istvánhoz. A tinta helyenként elmosódott. A kihagyott szó: „propänga" – propaganda?

61 MDK–C–I–18/523. Német Aladár nyomtatott levélpapírján kézzel írt (nem Bokros kézírása) megállapodás, a művész aláírásával.

62 MDK–C–I–18/134. Géppel írt, ceruzával javított lap.

63 MDK–C–I–18/135. Géppel írt levél másodpéldánya.

64 MDK–C–I–18/136. Postai levelezőlap.

65 MDK–C–I–18/137.1–2. „Magyar Művészet" stb. felírású nyomtatott levélpapíron írt levél.

66 MDK–C–I–18/138. Gépírásos levél másodpéldánya. A címzett: Szenes Árpád, Párizsban élő magyar származású festőművész.

67 MDK–C–I–18/139.1–2. „A Vallás- és Közoktatásügyi Minisztertől" felírású nyomtatott levélpapíron és borítékban küldött értesítés. Balra lent: Bokros Biermann Dezső úrnak, szobrászművész, Budapest, Katona József u. 28.

68 MDK–C–I–18/140. „Institut Hongrois" stb. felírású nyomtatott levélpapíron Párizsból küldött levél. Száma: 1435/1948.

69 MDK–C–I–18/142. Géppel írt levél másodpéldánya.

70 MDK–C–I–18/145. Géppel írt levél másodpéldánya.

MDK–C–I–18/146. „Institut Hongrois" stb. felírású, Párizsból küldött levél. Száma: 1665/1948.

72 MDK–C–I–18/147.1–2. „Sárospataki Szabadművelődési Akadémia" felírású nyomtatott levélpapíron írt levél.

73 MDK–C–I–18/150.1–2. „Institut Hongrois" stb. felírású nyomtatott levélpapíron írt levél.

74 MDK–C–I–18/153.1–2. „Budapesti Építési Hivatal" stb. felírású nyomtatott levélpapíron írt levél. Balra fent: Bokus (sic!) Birman Dezső szobrászművész úrnak, Budapest, Katona József utca 28. II. 11.

75 MDK–C–I–18/155.1–2. Szegi Pál kézzel írt levele.

76 MDK–C–I–18/156.1–2. Gépírásos körlevél. Címzés a borítékon.

77 MDK–C–I–18/159.1–2. Az „Institut Hongrois" nyomtatott levélpapírján írott levél.

78 MDK–C–I–18/165. A Vallás- és Közoktatásügyi Minisztérium értesítése. Ügyiratszám: 220.859/1948.VII. Balra lent: „A kiadvány hiteléül Doma Sándor irodavezető h."

79 MDK–C–I–18/506. A levél felső jobb sarkában: Tallós P. István, Magyaróvár, Városkapu tér 5.

80 MDK–C–I–18/176. Tallós P. István postai levelezőlapja. Feladó feltüntetése nélkül.

81 Gera Éva tulajdona. A „Magyar Képzőművészek Szabadszervezete" nyomtatott levélpapírján. Aláírás csak írógéppel; körpecsét.

82 MDK–C–I–18/499. Az 1949. évi távirat szövegét betű szerint közöljük.

83 Gera Éva tulajdona. Géppel írt levél másodpéldánya.

84 MDK–C–I–18/181. A „FÉSZEK Művészek Klubja elnöke" nyomtatott felírású levélpapíron írt gépelt meghívó. A megszólítás felett: Bokros-Biermann szobrászművész úrnak, Budapest.

85 MDK–C–I–18/182. „A magyar köztársasági elnök titkára" nyomtatott felírású levélpapíron írt értesítés. Balra lent: Bokros Biermann Dezső úrnak, Budapest.

86 Gera Éva tulajdona. A „Magyar Dolgozók Pártja Központi Vezetősége, Főtitkárság" felírású nyomtatott levélpapíron. Száma: FM/2347. Megszólítás helyett: Bokros Birman Dezső elvtársnak, Budapest, V., Katona József u. 28. II. 12.

87 MDK–C–I–18/183. A „Magyar Művészeti Tanács" nyomtatott levélpapírján írt levél. Száma: 334/1949. Képző- és iparművészeti szaktanácsok. Előadó: Dr. Zombori Miklós. Balra lent: „A kiadvány hiteléül: 1949. április 1. [olvashatatlan aláírás]."

88 MDK–C–I–18/184.1–2. A „Magyar Művészeti Tanács" nyomtatott levélpapírján írt levél. Címzés a borítékon.

89 MDK–C–I–18/185.1–2. A „Magyar Művészeti Tanács" nyomtatott levélpapírján írt levél. A borítékon a címzés alatt: „Távollétében titkára által is felbontandó!"

90 MDK–C–I–18/186. A Vallás- és Közoktatásügyi Minisztérium engedélye. Száma: 267.799/1949.X.

91 MDK–C–I–18/187. Ceruzával írt levél.

92 MDK–C–I–18/192.1–2. A Vallás- és Közoktatásügyi Minisztérium levele. Címzés a borítékon és a levél végén: Bokros Biermann József (sic!) úrnak, Budapest. Balra lent: „A kiadmány hiteléül Grylka János irodavezető."

93 MDK–C–I–18/193. Az Építés- és Közmunkaügyi Minisztérium (a „Közmunkaügyi" kiütve) nyomtatott levélpapírján írott levél. Száma: 10068/1949.eln./b. Körpecsét, aláírás.

94 MDK–C–I–18/196.1–2. Füst Milán kézírásos levele.

95 MDK–C–I–18/197. A „Magasépítési Tervező Intézet Lakóépülettervező Iroda" nyomtatott levélpapírján írott levél. Száma: III/1608/1949.KF/OK.

96 MDK–C–I–18/524. Összehajtogatott kockás füzetlapra ceruzával írt levélfogalmazvány.

97 MDK–C–I–18/198. Luigi Cicutti kézírásos levele Rómából.

98 MDK–C–I–18/149. 1–2. A Szabad Száj c. szatírikus hetilap nyomtatott levélpapírján írt levél.

99 MDK–C–I–18/200. Gépírásos levél.

100 MDK–C–I–18/201. Gépírásos levél.

101 MDK–C–I–18/206. A „Művészeti Szövetségek Háza" nyomtatott levélpapírján írott levél. Körpecsét, aláírás. Megszólítás helyett: Bokros Biermann Dezső úrnak, Bp.

102 MDK–C–I–18/209. Gépírásos levél másodpéldánya.

103 MDK–C–I–18/212. Gépelt levél másodpéldánya. Hátoldalán ceruzával készült, odavetett vázlatok a Sztálin-szoborhoz. Szövegek: „Örök hála és hűség a felszabadító Szovjetunió, a dicső Szovjet Hadsereg, népünk barátja és tanítója, a nagy Sztálin iránt!" „Beküldés május 15" stb.

104 MDK–C–I–18/211. Postai levelezőlap. Címzés: Bokros B. Rezső (sic!) szobrász, Katona József u. 8. Feladó: Vedres, Kiss J. altáb. 55. A postabélyegző kelte: 50. 1. 23.

105 MDK–C–I–18/216. „Budapest Főváros polgármestere" szövegű gépelt fejléc alatt a levél ügyszáma: 3835/59/2/1950. XI. Tárgy: „Előleg kiutalása Sztálin generalisszimusz szobrának elkészítésére meghívott és zárt pályázaton résztvevő szobrászművészek részére." Balra lent: „A kiadmány hiteléül: Budapest, 1950. március 28. A Polgármesteri XI. Ügyosztály. Lajtos György irodavezető".

106 MDK–C–I–18/219. „Budapest Főváros Polgármestere, Központi Lakáshivatal" hivatalos értesítése nyomtatott űrlapon. Iktatószám: ad 3276/B/311. Aláírás írógéppel, s. k. jelzéssel, irodavezető szignója, pecsét.

107 MDK–C–I–18/220. A Népművelési Minisztérium hivatalos megbízása. Előadó: Holba Tivadar. Iktatószám: 1711–B–10. Aláírás írógéppel, s. k. jelzéssel. Irodavezető aláírása (Lenhard), pecsét.

108 MDK–C–I–18/223. A Budapest–Kőbányai Középítő Vállalat levele. Jelzés: Műsz. o. 427. szám. Szénási/KBné. Aláírás: Gellért. Cégbélyegző. Címzés: Bokros N (sic!) szobrászművész elvtársnak, Budapest, Lehel tér 2. „D" épület.

109 MDK–C–I–18/224. Országos Nyugdíjintézet hivatalos értesítése. Jelzése: Dr. Szabó/gye. II/4.341.613/1950.

110 MDK–C–I–18/227.1–2. A Művészeti Alkotások N. V. levele. Címzés és utóirat kivételével gépeléssel sokszorosított levél. Jelzés: KM. Ügyintéző: Dr. Fehér. Aláírás tollal. Az utóiratban kézírásos kiegészítés: „Bartics élmunkás portré c."

111 MDK–C–I–18/225. Gépírásos levél másodpéldánya. Címzett nem szerepel.

112 MDK–C–I–18/534. Redő Ferenchez, a Népművelési Minisztérium Képzőművészeti Osztálya akkori vezetőjéhez írt levéltervezet javított másodpéldánya.

113 MDK–C–I–18/228. Gépírásos levél. Jelzés: Dolgozók Nyilvántartása. NJ/SzS.160/1951. Ügyintéző: Nagy János. Aláírás tollal és géppel. Cégbélyegző.

114 MDK–C–I–18/226.1–2. Kézzel írott levél, kézírással címzett borítékban.

115 MDK–C–I–18/230. Gépírásos szöveg másodpéldánya.

116 MDK–C–I–18/253.1–2. Földalatti Vasút Beruházási Vállalat fejléces papírján írt felszólítás. Ügyiratszám: 3525/1951. Ügyintéző: dr. Horváth/Szné. Aláírás tollal és géppel; cégbélyegző. Kézzel címzett boríték, hátul gépírással: „A Népművelési minisztérium a Közlekedés- és Postaügyi minisztériumnál eszközölje ki, hogy Bokros Birmannt mentsék fel a visszafizetés alól."

117 Gera Éva tulajdona. Gépírásos szöveg ceruzával javított másodpéldánya.

118 MDK–C–I–18/255.1–2. „Népművelési Miniszter" felírású, nyomtatott fejléces papíron írt levél. Száma: 2533/1951. Olvashatatlan aláírás, körbélyegző.

119 Gera Éva tulajdona. Idegen kézírású levélfogalmazvány.

120 Gera Éva tulajdona. A „Magyar Dolgozók Pártja Központi Vezetősége" nyomtatott levélpapírján. Száma: K/K/58369/951.

121 MDK–C–I–18/232. Géppel írt levél másodpéldánya.

122 MDK–C–I–18/233.1–2. Hivatalos levél. Jelzés: 52/4477/Hné/Nné. Aláírás: olvashatatlan.

123 MDK–C–I–18/234. Magyar Művészettörténeti Munkaközösség levele. Ügyiratszám: 56/1952. Aláírás, pecsét.

124 MDK–C–I–18/235. A Magyar Népköztársaság Képzőművészeti Alapja levele. Hiv. szám: 87710–2–79. Előadó: Bokor/BLné. Aláírás géppel, alatta: mb. Bokor Vilmos. Pecsét.

125 MDK–C–I–18/236.1–2. A Népművelési Minisztérium nyomtatott fejléces papírján írt levél. Aláírás tollal és géppel. Pecsét.

126 MDK–C–I–18/237. Gépírásos, kézzel több helyen javított, kiegészített fogalmazvány. – Amenophis király említett portréja a Művészeti Lexikon I. kötete szerint jelenleg a Staatliche Museen zu Berlin tulajdonában van.

127 MDK–C–I–18/238. Gépírásos levél másodpéldánya.

128 MDK–C–I–18/239. A Művészeti Dolgozók Szakszervezete nyomtatott fejléces papírján írt levél. Aláírás tollal és géppel. Bélyegző: Magyar Művészeti Dolgozók Szakszervezete képzőművészeti és iparművészeti felelős.

129 MDK–C–I–18/256. Erőmű Beruházási Vállalat fejléces papírján írt levél. Ügyiratszám: 13.100/I/In/GG/Szné. Ügyintéző: Gela.

130 MDK–C–I–18/240. A művész gépírásos levelének másodpéldánya.

131 Gera Éva tulajdona. Gépírásos szöveg; kiegészítések idegen kézírással.

132 Gera Éva tulajdona. Géppel írt levél másodpéldánya.

133 MDK–C–I–18/257.1–2. A „Magyar Dolgozók Pártja Központi Vezetősége Agitációs és Propaganda Osztálya" fejléces papírján írt levél. Ügyintéző: TI/Mné.

134 MDK–C–I–18/241. Postai levelezőlap.

135 MDK–C–I–18/242. Postai levelezőlap. Feladó: Bukor Béla Tibor. XII., Fohász lépcső 14. A postabélyegző kelte: 1952. VIII. 19.
Bukor Béla Bokros tanítványa és személyi titkára, akiről a művész a jól ismert portréfejet mintázta.

136 Gera Éva tulajdona. Géppel írt levéltervezet; ráírások ceruzával, pl.: „újra megíratni és elküldeni".

137 Gera Éva tulajdona. Géppel írt levél másodpéldánya.

138 MDK–C–I–18/243. Gépírásos levél másodpéldánya. Hátoldalán ceruzával írt feljegyzések.

139 MDK–C–I–18/575.8. Orvosi javaslat. Hosszú bélyegző: „Kútvölgyi úti Állami Kórház és Rendelőintézet Ideggyógyászati Osztálya, Budapest, XII., Kútvölgyi út 4." Háromszögletű bélyegző: „Budapest Fővárosi Tanácsa Rendelőintézete. XII/2. Ideg."

140 MDK–C–I–18/244.1–2. Füst Milán kézzel írt levele.

141 MDK–C–I–18/245. Géppel írt lap; a szöveg alatt ceruzával írt feljegyzések.

142 MDK–C–I–18/247. Gépírásos levél másodpéldánya.

143 MDK–C–I–18/248.1–2. A Villamosművek Központi Jogi Csoportjának felszólítása. Ügyiratszám: 1532/53. Aláírás géppel, felette tollal. Tollal írt kiegészítés: „mert a bizottság a pályázatát nem fogadta el."

144 MDK–C–I–18/248.2. Géppel írt felszólítás. Aláírás. A hátlapon a művész ceruzával írt sorai: „A felszólítás vétele utáni napon elmondtam az ügyvédemnek észrevételeimet! erre H. azt mondta – ja, ez más, tehát ezt nem is lehet per útján elintézni. Ezt az ügyet kiveszem a peres ügyekből – ezt nem lehet az alperes által előadottak alapján perelni – április óta nem is volt róla szó."

145 MDK–C–I–18/250. A művész kézzel írt levele (fogalmazványa?).

146 MDK–C–I–18/251. Géppel írt levél (fogalmazvány?).

147 MDK–C–I–18/252.1–2. A „Magyar Dolgozók Pártja Központi Vezetősége" nyomtatott fejléces papírján írt levél. Aláírás tollal. A boríték címzése ua., felül géppel: „Magyar Dolgozók Pártja Országos Központja."

148 MDK–C–I–18/259.1–2. Füst Milán levele.

149 MDK–C–I–18/260. A Szerzői Jogvédő Hivatalhoz intézett gépírásos levél másodpéldánya.

150 MDK–C–I–18/261. A Fővárosi Emlékmű Felügyelőség írógéppel sokszorosított felszólítása. Az összehajtott lap külső oldalán címzés: Bokros Birmann S. (sic!) szobrászművész kartársnak, Budapest, XIII., Élmunkás tér 2/d.

151 MDK–C–I–18/262.1. Géppel írt levél másodpéldánya.

152 MDK–C–I–18/262.2. Géppel írt eredeti levél.

153 MDK–C–I–18/263. Postai levelezőlapon ceruzával írt levél. Címzés: Bokros Birman Dezső. Kossuth-díjas szobrászművész. Balatonfüred, Szívkórház. Feladó neve és címe. A postabélyegző kelte: 1953. VIII. 20.

154 MDK–C–I–18/264. Hivatalos végzés, kitöltött űrlapon. Száma: 809–I–98/1953. Előadó: Szilágyi. Hivatk. szám: 1419–M–14. Balra lent: „A kiadmány hiteléül: Bp. 1953. aug. 28. Kocsi Imre irodavezető".
A Véghatározat szövegét kihagyásokkal közöljük.

155 MDK–C–I–18/266. Gépelt levél másodpéldánya, aláírással. A hátlapon több, ceruzával írt feljegyzés.

156 MDK–C–I–18/267.1–2. Gépelt levél másodpéldánya.

157 MDK–C–I–18/376. Márffy Ödön (1878–1959) festőművész kézírásos levele.

158 Gera Éva tulajdona. Gépírásos levél másodpéldánya.

159 MDK–C–I–18/268.1–2. A „Magyar Képzőművészek és Iparművészek Szövetsége" nyomtatott fejléces papírján írt levél. Iktatószám: 2055/1953.

160 MDK–C–I–18/312. Gépírásos levél másodpéldánya.

161 MDK–C–I–18/269. Gépelt levél másodpéldánya.

162 MDK–C–I–18/271. Gépelt levéltervezet.

163 MDK–C–I–18/270. Gépelt levél másodpéldánya.

164 MDK–C–I–18/273.

165 MDK–C–I–18/274.1–2. Füst Milán levele. Címzés a kockás füzetlapból ragasztott borítékon.

166 MDK–C–I–18/275. A Művészeti Dolgozók Szakszervezete Képzőművész és Iparművész Tagozatának levele, melyet minden valószínűség szerint Bokros kérésének támogatására a művésznek adtak át. Bélyegző, aláírás.

167 MDK–C–I–18/276.1–2. Gépelt levél vagy levéltervezet másodpéldánya, aláírás nélkül.

168 MDK–C–I–18/277. A Budapesti 16. sz. Ügyvédi Munkaközösség nyomtatott fejléces papírján írt levél. Ügyszám: 2897. Ügyintéző: Cs. D.

169 MDK–C–I–18/278. A Népművelési Minisztérium nyomtatott fejléces papírján írt levél. Ikt. sz.: 1418–B–20. Előadó: Cseh Miklós. Aláírás írógéppel; pecsét. Balra lent: „A kiadvány hiteléül: Babotay irodavezető."

170 MDK–C–I–18/279. A Budapesti 16. sz. Ügyvédi Munkaközösség nyomtatott fejléces papírján írt levél. Ügyszám: 2897. Ügyintéző: Cs. D.

171 MDK–C–I–18/280.1–2. A Szabad Nép szerkesztőségének nyomtatott fejléces papírján írt levél.

172 MDK–C–I–18/281.1–2. A Magyar Foto nyomtatott fejléces papírján írt levél. Jel: VÁ. Ügyintéző: Busztin. Cégbélyegző, aláírás.

173 MDK–C–I–18/282. Postai levelezőlapon ceruzával írt levél. Feladó neve és címe.

174 MDK–C–I–18/283. Összehajtott papírlapon, tollal írt levél.
Dutka Ákos költő Dutka Mária (Baby) művészettörténész édesapja.

175 MDK–C–I–18/284.1–2.

176 MDK–C–I–18/575a. A „Budapesti XIII. ker. Tanács Végrehajtó Bizottságának Szociálpolitikai Csoportja" által kiadott, gépelt fejléces véghatározat. Ügyiratszám: 831/B–306/1954. Előadó: Eperjessy E. Balra lent: „A kiadmány hiteléül: [olvashatatlan aláírás], Budapest, 1954. szept. 8."

177 MDK–C–I–18/285. Levél a Budapesti 16. sz. Ügyvédi Munkaközösség nyomtatott fejléces papírján. Ügyszám: 2897. Ügyintéző: Cs. D.

178 Gera Éva tulajdona. A Népművelési Minisztérium nyomtatott levélpapírján írt levél. Iktatószám: 87713–3–35/1954. Előadó: Faludi György. Balra lent: „A kiadmány hiteléül: [olvashatatlan aláírás], irodavezető."

179 Gera Éva tulajdona. Géppel írt levél másodpéldánya.

180 MDK–C–I–18/286.2. A Herendi Porcelángyár levele. Jel: GO/Dné. Ügyintéző: Geisse Ottó. Cégbélyegző. Két olvashatatlan aláírás.

181 MDK–C–I–18/287. A Magyar Népköztársaság Képzőművészeti Alapja levele. Száma: 8628. Előadó: Zöldné.

182 MDK–C–I–18/289. Géppel írt levél másodpéldánya.

183 MDK–C–I–18/290. Gépelt levél másodpéldánya.

184 MDK–C–I–18/291. A XIV. kerületi Tanács Végrehajtó Bizottsága által kiállított hivatalos írás. Bélyegző, aláírás.

185 MDK–C–I–18/292. Vértes György gépírásos levele. A jobb felső sarokban ceruzával felírva Vértes György címe.
Vértes György szerkesztő, újságíró, az Országgyűlési Könyvtár nyugalmazott igazgatója.

186 MDK–C–I–18/293.1–2. Vértes György gépírásos levele. Címzés a borítékon.

187 MDK–C–I–18/294.1–2. Gépírásos levél. Címzés a borítékon.

188 MDK–C–I–18/295.1–2. A Szépművészeti Múzeum Szoborosztályáról írott levél. Címzés a borítékon. Megszólítás helyett a levélen: Bokros-Birman Dezső szobrászművész.

189 MDK–C–I–18/272.1–2. Gépelt igazolás fogalmazványa és a tisztázat másodpéldánya. A fogalmazványra ceruzával felírva: „Hétfő Hantoshoz".

190 MDK–C–I–18/296.1–2. Gépírásos levél. Címzés a borítékon.

191 MDK–C–I–18/297. Géppel írt és géppel aláírt levél.

192 MDK–C–I–18/299. Gépírásos levél másodpéldánya.

193 MDK–C–I–18/300. Az Országos Szépművészeti Múzeum átvételi elismervénye. Bélyegző, aláírás.

194 MDK–C–I–18/302. Az Országos Szépművészeti Múzeum levele. Ügyiratszám: 863–03–223/955. Bélyegző, aláírás.

195 MDK–C–I–18/303. Gépelt levél, aláírva.

196 MDK–C–I–18/304. Gépelt levéltervezet, aláírva.

197 MDK–C–I–18/305. Levél „Népművelési Minisztérium, Miniszterhelyettes" felírású nyomtatott levélpapíron. Aláírás, körpecsét.

198 MDK–C–I–18/307.1–2. Gépelt levél másod- és harmadpéldánya, aláírva.

199 MDK–C–I–18/308. A Budapesti 16. sz. Ügyvédi Munkaközösség nyomtatott levélpapírján írt levél. Ügyszám: 2897. Ügyintéző neve. – A levélben közölt részletes elszámolást kihagytuk!

200 MDK–C–I–18/532. Gépírásos feljegyzés.

201 MDK–C–I–18/309.1–2. A Külkereskedelmi Minisztérium nyomtatott levélpapírján írt levél. Ügyiratszám: VZ–2571–1955.

202 MDK–C–I–18/310. Gépírásos levél másodpéldánya.

203 MDK–C–I–18/313. Gépírásos levél másodpéldánya. – A levélben közölt részletes elszámolást kihagytuk!

204 MDK–C–I–18/315. Gépírásos levél másodpéldánya.
205 MDK–C–I–18/316. Az Országos Szépművészeti Múzeum nyomtatott levélpapírján írt levél. Ügyiratszám: 863–03–18/956. Aláírás, körpecsét.
206 MDK–C–I–18/317. Az Országos Szépművészeti Múzeum levele. Ügyiratszám: 863–01–42/956. Ügyintéző: Lázár Gyuláné. Bélyegző, aláírás.
207 Gera Éva tulajdona. Gépírásos levél másodpéldánya.
208 MDK–C–I–18/318. Gépírásos levél másodpéldánya.
209 MDK–C–I–18/319. Az Országos Szépművészeti Múzeum levele. Ügyiratszám: 863–13–9/956. Bélyegző, aláírás.
210 MDK–C–I–18/320.1–2. Címzés a borítékon: Bokros Birman Dezső. Kossuth-díjas szobrászművész, Sárospatak, Rákóczi-vár. Feladó: B. b. T. Bp. Élmunkás tér 2/d.
211 MDK–C–I–18/322.1–2. Címzés a borítékon: Bokros Birman Dezső. Kossuth-díjas szobrászművész, Budapest, Élmunkás tér 2/d. Ungarn. Feladó neve és bécsi címe.
212 MDK–C–I–18/325. Kézzel írt levél.
213 MDK–C–I–18/326.1–2. Géppel írt levél. Címzés a borítékon.
214 MDK–C–I–18/332. „Művelődésügyi Minisztérium, Miniszterhelyettes" felírású nyomtatott levélpapíron küldött értesítés. Száma: 27/1958.M.h.t. Aláírás, körpecsét.
215 MDK–C–I–18/333. Postai levelezőlap. A postabélyegző kelte: 1958. jan. 24.
216 MDK–C–I–18/335.1–2. Román György festőművész kézírásos levele. Címzés és feladó a borítékon. A postabélyegző kelte: 1958. febr. 28.
217 MDK–C–I–18/338. Postai levelezőlap. Címzés, feladó az előlapon. A postabélyegző kelte: 1958. ápr. 15.
218 MDK–C–I–18/340. Géppel írt levél másodpéldánya.
219 MDK–C–I–18/344. Kockás füzetlapon kézzel írt levélfogalmazvány.
220 MDK–C–I–18/349. Gépírásos levél.
221 MDK–C–I–18/350. A Sárospataki Rákóczi Múzeum nyomtatott levélpapírján írott levél.
222 MDK–C–I–18/354. Postai levelezőlap.
223 MDK–C–I–18/355. Az Élet és Irodalom szerkesztőségének nyomtatott levélpapírján írott levél.
224 MDK–C–I–18/357.1–2. Névjegy nagyságú kartonon írt levél. Címzés a borítékon. A postabélyegző kelte: 959. jan. 12.
225 MDK–C–I–18/363. A Művelődésügyi Minisztérium levelének a Magyar Népköztársaság Képzőművészeti Alapjánál készített hiteles másolata. Balra lent: „A kiadmány hiteléül: [olvashatatlan aláírás] s. k., irodavezető. A másolat hiteles: Kőszegi".
226 MDK–C–I–18/364. Kézírásos meghatalmazás (nem Bokros kézírása).
227 MDK–C–I–18/500.1–2. A zárójelentés hivatali száma a borítékon: 82221/59.
228 MDK–C–I–18/365. A Szépirodalmi Könyvkiadó nyomtatott levélpapírján írt levél. A levél jele: DM/BA. Olvashatatlan aláírás, cégbélyegző.
229 MDK–C–I–18/574. A „Képzőművészek, Iparművészek és Művészeti Dolgozók Szakszervezete" nyomtatott levélpapírján írt levél. Aláírás, körpecsét.
230 MDK–C–I–18/368. Stencilezett űrlapon küldött hivatalos értesítés. Száma: kjő 85721/1958–5. Balra lent: „A kiadmány hiteléül: Molnár Lászlóné irodavezető, 136/B. Közjegyző által elrendelt közvetlen letiltás megszüntetése. Kiadmány."
231 MDK–C–I–18/373.1–2. Gépírásos levél. Címzés és feladó a borítékon.
232 MDK–C–I–18/378. Bokros Szenes Árpádhoz, a Párizsban élő magyar származású festőművészhez írott gépírásos levelének másodpéldánya.
233 MDK–C–I–18/381. A Magyar Újságírók Országos Szövetségéből küldött levél.
234 MDK–C–I–18/384. Gépelt levél másodpéldánya.
235 Gera Éva tulajdona. Gépírásos levél, aláírással.
236 MDK–C–I–18/536. Idegen kéz által, tollal írt levéltervezet.
237 MDK–C–I–18/385. A Magyar Szocialista Munkáspárt nyomtatott levélpapírján.
238 MDK–C–I–18/387. A Művelődésügyi Minisztérium Képzőművészeti Osztályának értesítése. Ügyiratszám: 72.059/1960.IX.
239 MDK–C–I–18/389. Postai levelezőlap. Feladó: dr. Bokor Lajos, Magyar Távirati Iroda.
240 MDK–C–I–18/386.1–2. Keleti Arthur névjegykartonon írt, kézírásos levele. A postabélyegző kelte: 1960. I. 18.
241 MDK–C–I–18/390. Gépelt levél másodpéldánya.
242 MDK–C–I–18/391.1–2. Szenes Árpád Párizsban élő magyar származású festőművész kézzel írott levele. (Felesége: Maria Helena Vieira da Silva portugál származású festőművésznő.) A postabélyegző kelte: 1960. I. 26.
243 MDK–C–I–18/393. Gépírásos levél másodpéldánya.

244 MDK–C–I–18/395. Gépírásos levél másodpéldánya.
245 MDK–C–I–18/493. A XIII. kerületi Tanács Végrehajtó Bizottsága hivatalos papírján fogalmazott határozat. Ügyiratszám: 510/1/1960.szab.
246 MDK–C–I–18/470.1–2. Gépelt levél. Feladója Balázs Anna írónő.
247 MDK–C–I–18/394.1–3. Balázs Anna írónő levele Román György festőművészhez. – A levélnek csak Bokros Birmanra vonatkozó részét közöljük.
248 MDK–C–I–18/396. Gépírásos levél másodpéldánya.
249 MDK–C–I–18/399. Postai levelezőlapon a művész kézzel írt sorai.
250 MDK–C–I–18/401.1–2. Géppel írt levéltervezet első- és másodpéldánya.
251 MDK–C–I–18/531. Ceruzával, valószínűleg a nevezett által írt igazolástervezet.
252 MDK–C–I–18/403.1–2. „Institut Hongrois" stb. felírású nyomtatott levélpapíron géppel írt levél.
253 MDK–C–I–18/404. Gépelt levél másodpéldánya.
254 MDK–C–I–18/406. Ceruzával írt levél.
255 MDK–C–I–18/533. Géppel írt levéltervezet.
256 MDK–C–I–18/415. A Magyar Tudományos Akadémia Bartók Archívumának nyomtatott levélpapírján.
257 MDK–C–I–18/416. A Művelődésügyi Minisztérium Képzőművészeti Osztályának levele. Ügyiratszám: 72.366/1962. Előadó: Kmetty Jánosné. Balra lent: „A kiadmány hiteléül: [olvashatatlan aláírás], irodavezető".
258 MDK–C–I–18/417. Az Országgyűlési Könyvtár nyomtatott levélpapírján írt levél. A „plakát" szó nyilván elírás; Bokros Független Magyarországért című plakettjéről van szó.
259 MDK–C–I–18/392. A „Magyar Forradalmi Munkás-Paraszt Kormány Elnökhelyettesének Titkársága" nyomtatott levélpapírján.
260 MDK–C–I–18/418. A Jókai Színház nyomtatott levélpapírján.
261 MDK–C–I–18/419.1–2. Gépírásos üzenet két példányban, mindkettő ceruzával szignálva.
262 MDK–C–I–18/422. Bokros titkárának ceruzával írt levélfogalmazványa. Cím: Gerelyes igazgató elvtárs, Újkori Történelmi Múzeum, József nádor tér 2.
263 MDK–C–I–18/423. Bokros titkárának ceruzával írt levélfogalmazványa. Cím: Dömötör Teréz, Samarja (na Astove), Csehszlovákia.
264 MDK–C–I–18/426. Géppel írt levél másodpéldánya.
265 MDK–C–I–18/436.1–2. Baumgartner József bronzöntő levele.
266 MDK–C–I–18/554. Gépelt lap.
267 MDK–C–I–18/447. Idegen kézírással (Laczkovich Alice) készült, Bokros által aláírt megállapodás.
268 MDK–C–I–18/453. A Magyar Nemzeti Galéria nyomtatott levélpapírján írt levél. Aláírás, körpecsét.
269 MDK–C–I–18/463. 1–2. A Hazafias Népfront XIII. ker. Bizottsága nyomtatott levélpapírján írt levél. Aláírás, körpecsét.

KÉPEK JEGYZÉKE

Rövidítések: J. = Jelezve
J. n. = Jelzés nélkül
MNG = Magyar Nemzeti Galéria
tul. = tulajdona

A SZÖVEG KÖZÖTT:

I. A művész rózsadombi műtermében, 1919 körül
II. Egy lap a Jób-mappából, 1920
III. A művész 1937-ben
IV. A művész 1947-ben
V. A művész Párizsban, 1948-ban
VI. Művészek között a sárospataki alkotóházban, 1949-ben
VII. A szobrász és önportréja
VIII. A 75 éves művész

---

1. Alvó leány, 1916
Márvány, 113 cm
J.: Bokros Birman, 1916
MNG 58.19–N

2. Torzó, 1917
Gipsz
Ismeretlen helyen

3. Jónap Andorné, 1915 körül
Gipsz, 33 cm
J. n.
Jónap Andorné tul.

4. Ülő fiú akt, 1919
Bronz, 30 cm
J.: Bokros Birman, 1919
Dr. Podoski József tul.

5. Ülő női torzó, 1922
Terrakotta, 25,5 cm
J.: B. B., 922
Frankfurt József tul.

6. Guggoló nő, 1921
20 cm
J.: B. B., 1921
Szegi Pálné tul.

7. Négykézlábálló, 1921
Bronz
Ismeretlen helyen

8. Akrobata, 1921
Ismeretlen helyen

9. Híd, 1921
Gipsz, 21 cm
J. n.
Rosta Jánosné tul.

10. Anya és lánya, 1922
Gipsz, 93 cm
J.: Bokros Birman 922
Bokros Birman hagyatéka

11. László Mihály, 1923
Bronz, 36 cm
J. n.
Kiscelli Múzeum tul.

12. Ölelkezők, 1923
Bronz, 30 cm
J.: B. B. 1923
Boros István tul.

13. Önportré, 1923
Gipsz, 35 cm
J.: Bokros Birman 1923
Kovács György tul.

14. Ady Endre, 1924
Bronz, 35,5 cm
J.: Bokros Birman 924
MNG 56.113–N

15. Áchim András-emlékmű terv, 1924
Gipsz, 37 cm
J.: B. B. 1924
Bokros Birman hagyatéka

16. Ady-síremlék terv, 1927
Gipsz
Ismeretlen helyen

17. Ady-fej a síremléktervhez, 1927
Bronz, 17,5 cm
J. n.
Szmetana Ernő tul.

18. Bronz dombormű az Ady-síremlék tervhez
J.: B. B.
Ismeretlen helyen

19. Galiczáné, 1926
Gipsz, 34 cm
J.: Bokros Birman 1926
Bokros Birman hagyatéka

20. Keleti Artúr, 1927
Gipsz, 48 cm
J. n.
Keleti Artúr hagyatéka

21–22. K. Füredi Róza, 1927
Bronz, 38,5 cm
J.: B. B. 1927
Vadas Lászlóné tul.

23–24. Szágelné, 1927
Bronz, 30 cm
J.: B. B. 1927
Bolgár Béláné tul.

25. Újvári Péter, 1927
Gipsz, 37 cm
J. n.
MNG 59.105–N

26. Kalapos önportré, 1927
Bronz, 32 cm
J.: B. B. 1927
MNG 63.35–N

27–28. Fiú akt, 1928
Ismeretlen helyen

29. Don Quijote, 1929
Bronz, 32 cm
J.: Bokros Birman
MNG 56.67–N

30. Don Quijote, 1929
Bronz, 117 cm
J.: Bokros 1929
MNG 56.135–N

31–32. Don Quijote-fej (részlet)

33. Birman Izsóné, 1929
Gipsz, 29 cm
J. n.
Birman Izsó tul.

34. Gegesi Kiss Pál, 1930
Bronz, 29 cm
J.: Bokros 1930
MNG 66.14–N

35. Szőllősi Endre, 1930
Terrakotta, 32 cm
J.: Bokros Birman
Gádor István tul.

36. Masaryk, 1930
Gipsz, 29 cm
J.: Bokros 1930
Hauswirth Magda tul.

37. Gádor Istvánné, 1931
Terrakotta, 43 cm
J.: Bokros 1931
Gádor Istvánné tul.

38. „Teremtés", 1932
Bronz, 44 cm
J.: Bokros-Birman 1932
MNG 66.5–N

39. Schultheisz Baba, 1932
Bronz, 24 cm
J.: B. B. 1932
Schultheisz Miksa tul.

40. Futók, 1933
Bronz, 20×21 cm
J.: Bokros 1933
Bokros Birman hagyatéka

41. Scheiber Hugó, 1933
Bronz, 31 cm
J.: Bokros Birman 1933
MNG 66.80–N

42. Madame Sans Gêne, 1934
Bronz, 33 cm
J.: B. B. 1934
MNG 59.104–N

43. Köszöntő, 1935
Bronz, 39,5 cm
J. n.
MNG 66.4–N

44. Pán Imréné, 1935
Gipsz, 36 cm
J. n.
Bokros Birman hagyatéka

45. Bíró Henrik, 1936
Bronz, 38 cm
J.: Bokros Birman 1936
MNG 59.102–N

46. Önportré, 1939
Gipsz, 11×11 cm
J.: B. B. 1939
Frankfurt József tul.

47–48. Napbanéző bányász (változat), 1941
Bronz, 37 cm
J.: Bokros Birman 1941
Dr. Gegesi Kiss Pál tul.

49–50. Álló női akt
Gipsz, 41 cm
Dr. Gegesi Kiss Pál tul.

51. És vidd magaddal . . ., 1940
Gipsz, 16,5 cm
J.: B. B. 1940
Bokros Birman hagyatéka

52. József Attila, 1942
Bronz, 16,5 cm
J.: Bokros Birman 1942
MNG 56.140–N

53–54. Tékozló fiú megtérése, 1941
Bronz, 34 cm
J. n.
Zsidó Múz. tul.

55. Világ proletárjai egyesüljetek, 1941
Terrakotta, 20×30 cm
J. n.
Bokros Birman hagyatéka

56. Független Magyarországért, 1942
Terrakotta, 24,5×35 cm
J.: B. B. 1942
MNG 65.22–N

57. Kubikos, 1941
Bronz, 16 cm
J.: Bokros Birman
MNG 56.141–N

58. Aszfaltozó, 1943
Bronz, 19,6 cm
J.: Bokros Birman 1943
MNG 52.855–N

59. Tomi, 1942
Bronz, 25 cm
J.: B. B. 1942
MNG 57.32–N

60. Rokkant katona, 1944
Bronz, 27 cm
J.: Bokros 1944
MNG 55.862–N

61–63. Ruth és Noémi, 1944
Bronz, 25 cm
J.: B. B. D.
Dr. Gegesi Kiss Pál tul.

64. Glück Marianne, 1945
Gipsz, 30 cm
J.: B. B. 1945
Dr. Glück Tiborné tul.

65. Dózsa György, 1946
Gipsz, 40 cm
J. n.
Bokros Birman hagyatéka

66. Duna-völgyi népek kórusa, 1946
Bronz, 23,5 cm
J. n.
MNG 57.33–N

67. Bocskoros paraszt, 1948
Gipsz, 67 cm
J.: Bokros Birman
Bokros Birman hagyatéka

68. Kucsmás paraszt, 1948
Gipsz, 69 cm
J.: Bokros Birman
Bokros Birman hagyatéka

69. Nő teknősbékával, 1947
Bronz, 25 cm
J. n.
Dr. Gegesi Kiss Pál tul.

70. Gáspár Endre, 1947
Bronz, 32 cm
J.: B. B. 1947
MNG 56.138–N

71. Vasmunkás (vázlat)
Gipsz, 25 cm
J. n.
Bokros Birman hagyatéka

72. Vasmunkás, 1948
Bronz, 220 cm
Felállítva: a SZOT Székház előtt

73–74. Bukor Béla, 1948
Bronz, 32 cm
J.: Bokros Birman
MNG 56.134–N

75–76. Téglahordó, 1949
Gipsz, 48 cm
J.: B. B. 1949
Bokros Birman hagyatéka

77. Ulysses, 1949
Bronz, 20 cm
J. n.
MNG 69.16–N

78. Sógorom, 1946
Bronz, 18×16 cm
J.: B. B. 1946
Bokros Birman hagyatéka

79. Mednyánszky, 1955
Bronz, 36 cm
J.: Bokros Birman 1955
MNG 54.1948

80. Halászfiú, 1955
Bronz, 24 cm
J. n.
Bokros Birman hagyatéka

81. Önportré, 1955
Gipsz, 45 cm
J.: Bokros Birman 1955
MNG 57.31–N

82. Cica, 1957
Plasztelin, 8,5 cm
J. n.
Bokros Birman hagyatéka

83. Munkásfiú, 1957
Gipsz, 70 cm
J. n.
Bokros Birman hagyatéka

84. Vetkőző nő, 1957
Bronz, 17 cm
J. n.
MNG 68.39–N

85. Démoszthenész, 1957
Gipsz, 24 cm
J. n.
Bokros Birman hagyatéka

86. Szenes Zsuzsa, 1959
Terrakotta, 24 cm
J.: Bokros 1959
Erdélyiné Szenes Zsuzsa tul.

87. Bartók Béla, 1960
Gipsz, 16 cm
J. n.
Bokros Birman hagyatéka

88. Meditáló, 1960
Gipsz, 89 cm
J.: 1960
Bokros Birman hagyatéka

89. Szputnyiknézők, 1962
Gipsz, 51,5 cm
J. n.
Bokros Birman hagyatéka

90. Szputnyiknézők (részlet)

91. Kontyos női fej, 1962
Gipsz, 37 cm
J. n.
Bokros Birman hagyatéka

92. Álló férfi, 1964
Gipsz, 84 cm
J. n.
Bokros Birman hagyatéka

KÉPEK

1. Alvó leány, 1916

2. Torzó, 1917

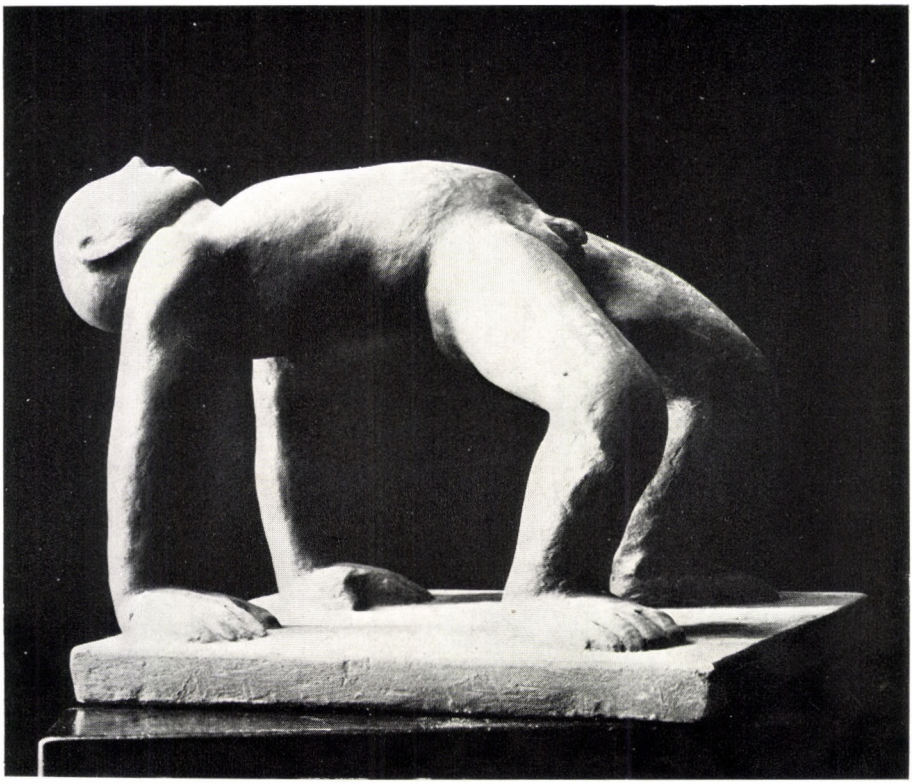
3. Jónap Andorné, 1915 körül

4. Ülő fiú akt, 1919

5. Ülő női torzó, 1922

6. Guggoló nő, 1921

7. Négykézlábálló, 1921



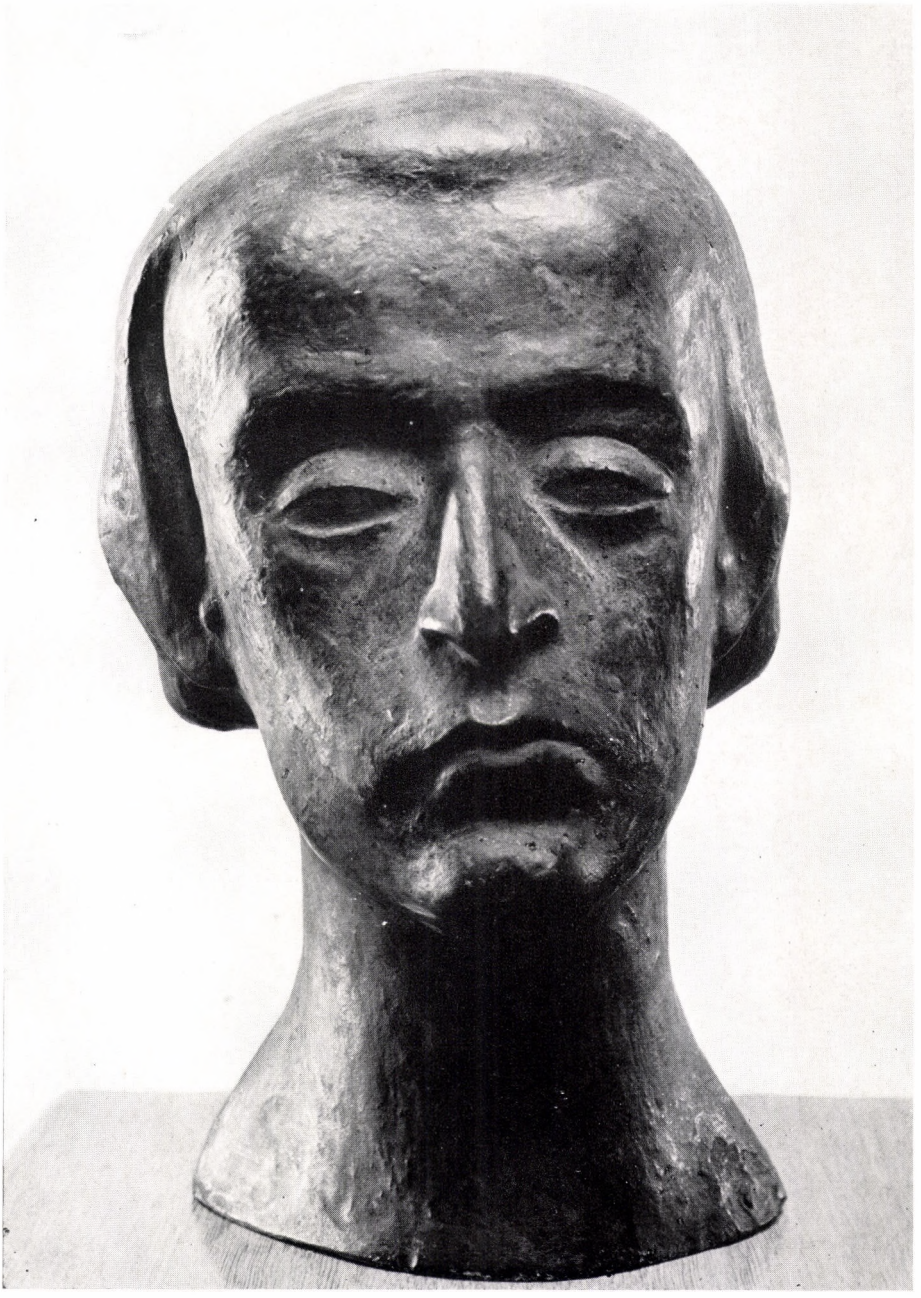
8. Akrobata, 1921

9. Híd, 1921

10. Anya és lánya, 1922

11. László Mihály, 1923

12. Ölelkezők, 1923



13. Önportré, 1923

14. Ady Endre, 1924

15. Áchim András-emlékmű terv, 1924
16. Ady-síremlék terv, 1927

17. Ady-fej a síremléktervhez, 1927

18. Bronz dombormű az Ady-síremlék tervhez

19. Galiczáné, 1926

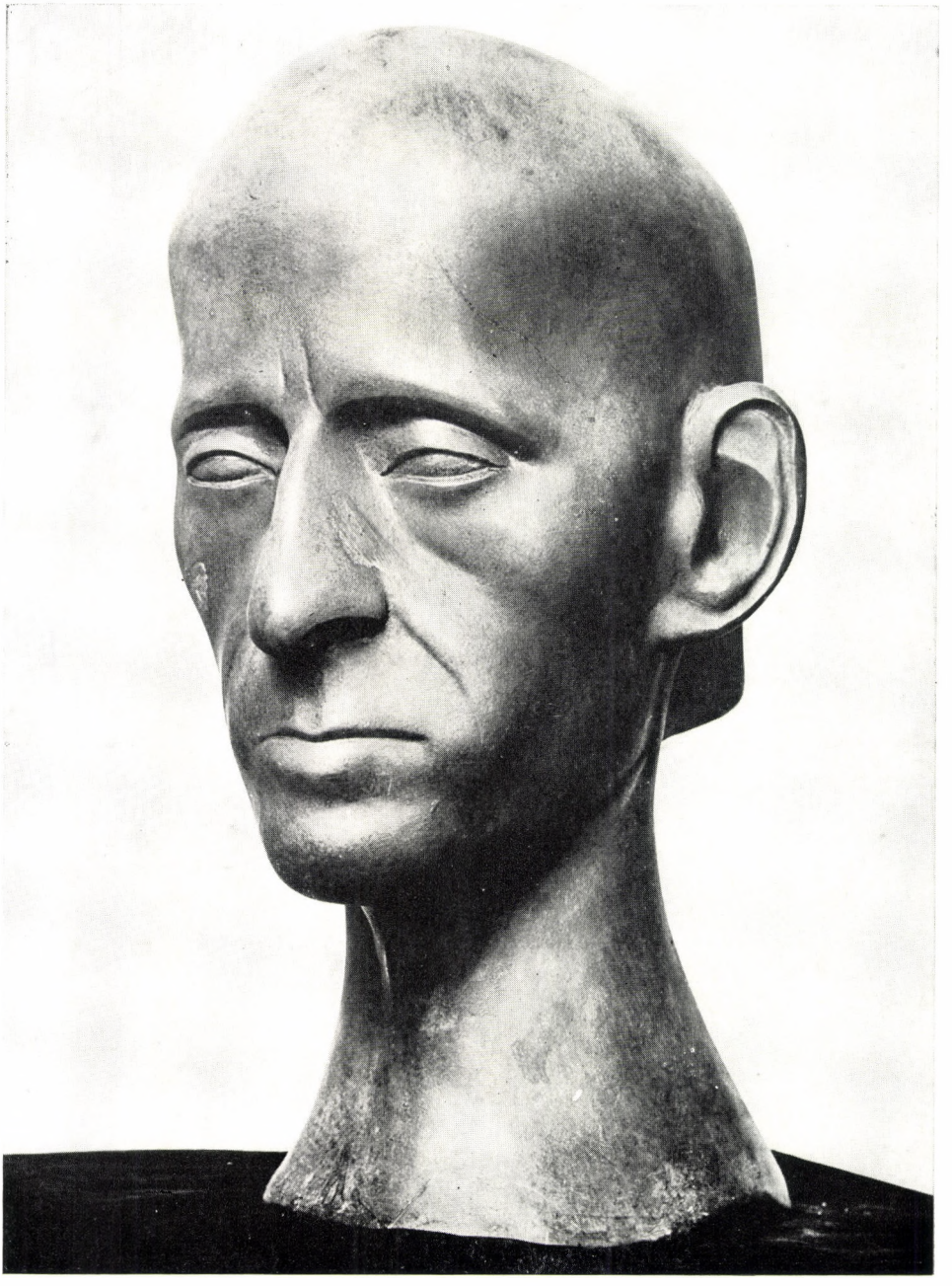
20. Keleti Artúr, 1927

21. K. Füredi Róza, 1927

22. K. Füredi Róza, 1927

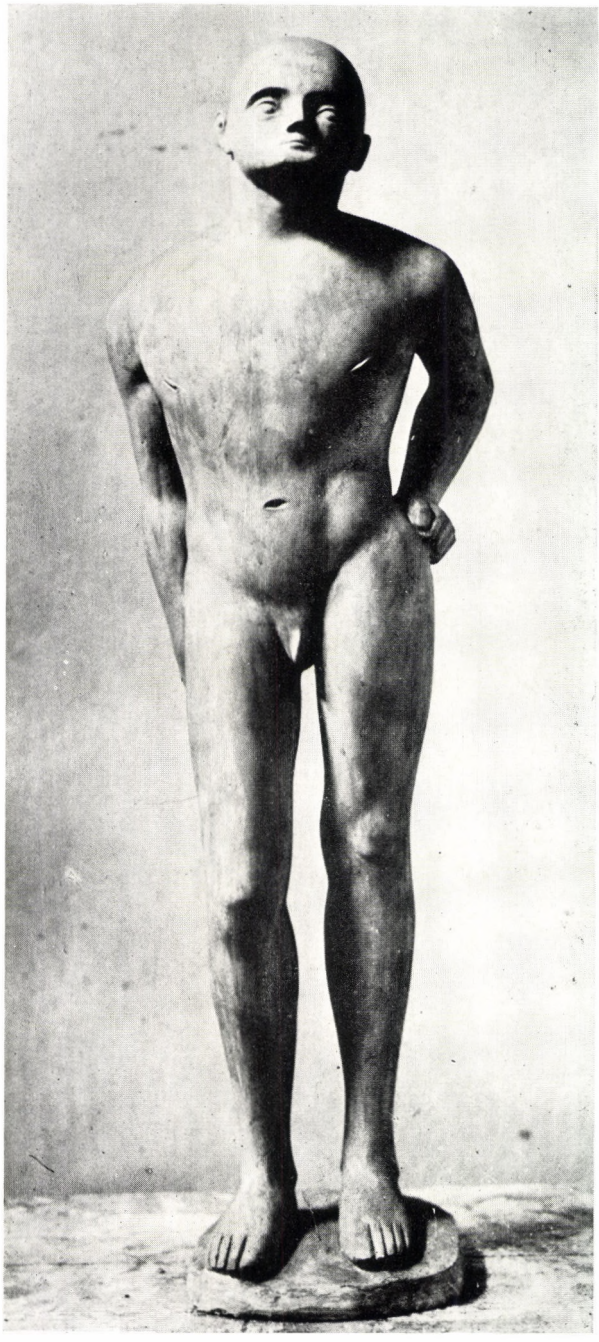
23. Szágelné, 1927

24. Szágelné, 1927

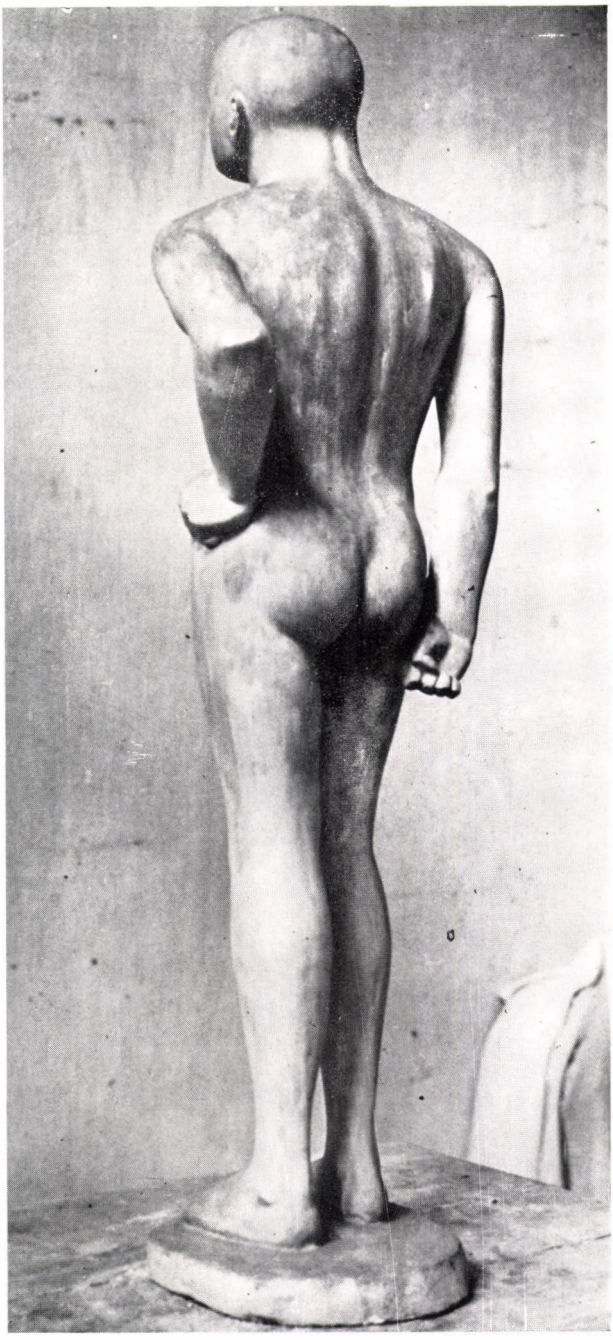


25. Újvári Péter, 1927

26. Kalapos önportré, 1927



27. Fiú akt, 1928



28. Fiú akt, 1928

29. Don Quijote, 1929

30. Don Quijote, 1929

31. Don Quijote-fej (részlet)

32. Don Quijote-fej (részlet)

33. Birman Izsóné, 1929

34. Gegesi Kiss Pál, 1930

35. Szőllősi Endre, 1930

36. Masaryk, 1930

37. Gádor Istvánné, 1931

38. „Teremtés", 1932

39. Schultheisz Baba, 1932

40. Futók, 1933

41. Scheiber Hugó, 1933

42. Madame Sans Gêne, 1934

43. Köszöntő, 1935

44. Pán Imréné, 1935

45. Bíró Henrik, 1936

46. Önportré, 1939

47. Napbanéző bányász (változat), 1941

48. Napban ező bányász (változat), 1941

49. Álló női akt

50. Álló női akt

51. És vidd magaddal . . ., 1940

52. József Attila, 1942

53. Tékozló fiú megtérése, 1941

54. Tékozló fiú megtérése, 1941

55. Világ proletárjai egyesüljetek, 1941

56. Független Magyarországért, 1942

57. Kubikos, 1941

58. Aszfaltozó, 1943

59. Tomi, 1942

60. Rokkant katona, 1944

61. Ruth és Noémi, 1944

62. Ruth és Noémi, 1944

63. Ruth és Noémi, 1944

64. Glück Marianne, 1945

65. Dózsa György, 1946

66. Duna-völgyi népek kórusa, 1946

67. Bocskoros paraszt, 1948

68. Kucsmás paraszt, 1948

69. Nő teknősbékával, 1947

70. Gáspár Endre, 1947

71. Vasmunkás (vázlat)

72. Vasmunkás, 1948

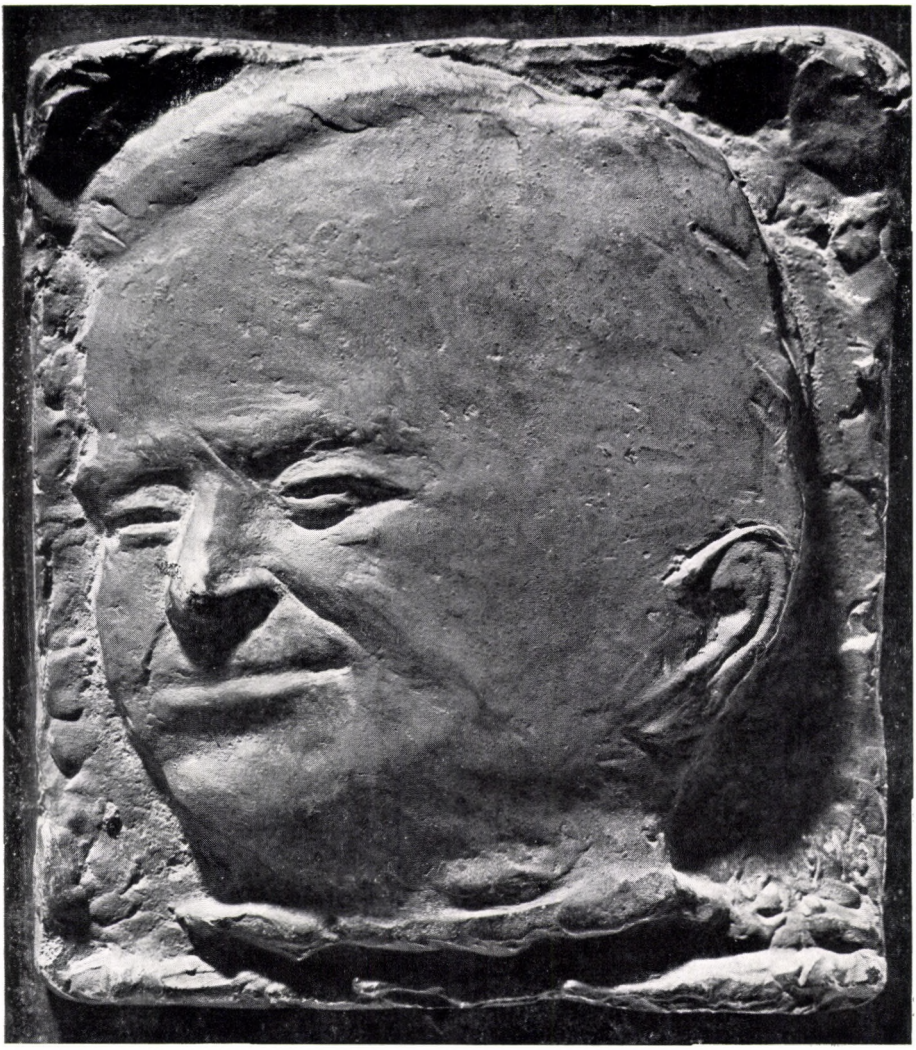
73. Bukor Béla, 1948

74. Bukor Béla, 1948

75. Téglahordó, 1949

76. Téglahordó, 1949

77. Ulysses, 1949



78. Sógorom, 1946

79. Mednyánszky, 1955

80. Halászfiú, 1955

81. Önportré, 1955

82. Cica, 1957

83. Munkásfiú, 1957

84. Vetkőző nő, 1957

85. Démoszthenész, 1957

86. Szenes Zsuzsa, 1959

87. Bartók Béla, 1960

88. Meditáló, 1960

89. Szputnyiknézők, 1962

90. Szputnyiknézők (részlet)

91. Kontyos női fej, 1962

92. Álló férfi, 1964

A kiadásért felelős az Akadémiai Kiadó igazgatója
Felelős szerkesztő: Dr. Szucsán Miklós Műszaki szerkesztő: Fülöp Antal
A burkoló- és kötésterv Kocsis Tibor munkája
Terjedelem: 18,25 A/5 ív + 8,05 ív melléklet
AK 7 k 7477
A szedés készült: Zrínyi Nyomda, Budapest (74.5079/9-2500)
Akadémiai Nyomda, Budapest. Felelős vezető: Bernát György
Printed in Hungary

Az Akadémiai Kiadó sorozata:

MŰVÉSZETTÖRTÉNETI
FÜZETEK

1. *Mojzer Miklós*
TORONY, KUPOLA, KOLONNÁD
78 oldal . Fűzve 21,— Ft

2. *Galavics Géza*
PROGRAM ÉS MŰALKOTÁS
A 18. SZÁZAD VÉGÉN
71 oldal . Fűzve 18,— Ft

3. *Szabó Júlia*
A MAGYAR AKTIVIZMUS
TÖRTÉNETE
83 oldal . Fűzve 22,— Ft

4. *Gervers-Molnár Vera*
A KÖZÉPKORI MAGYARORSZÁG
ROTUNDÁI
93 oldal . Fűzve 28,— Ft

5. *Vayerné Zibolen Ágnes*
KISFALUDY KÁROLY
A művészeti romantika kezdetei
Magyarországon
72 oldal . Fűzve 20,— Ft

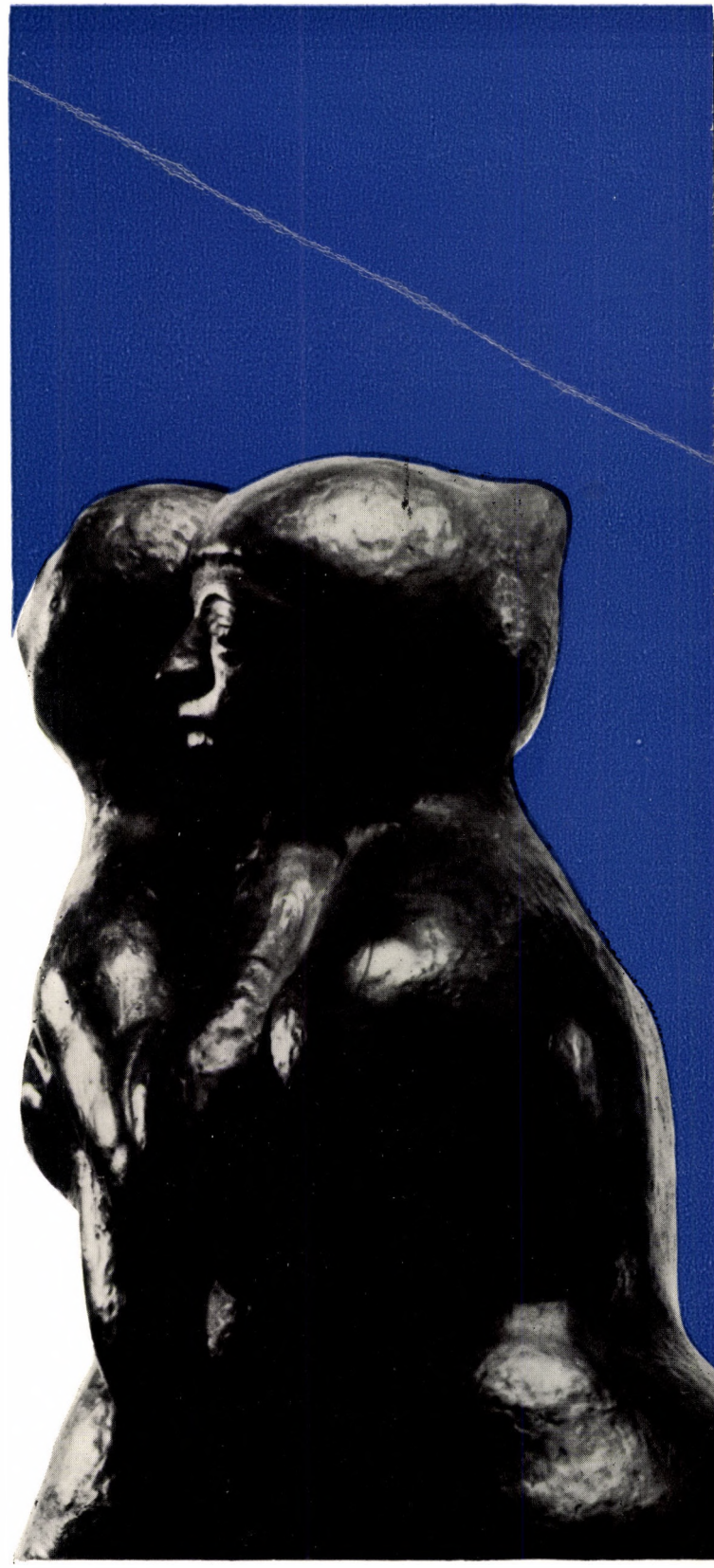
6. *Sz. Koroknay Éva*
MAGYAR RENESZÁNSZ
KÖNYVKÖTÉSEK
125 oldal . Fűzve 34,— Ft

AKADÉMIAI KIADÓ · BUDAPEST

Ára: 62,— Ft

BOKROS BIRMAN DEZSŐ

ÖNÉLETRAJZA, LEVELEZÉSE, MŰVEI



AKADÉMIAI KIADÓ, BUDAPEST