EUR 192.d,e

EUROPEAN ATOMIC ENERGY COMMUNITY - EURATOM

THREE WORKINGPAPERS ON THE THEORY OF COMPUTATION

by

L. A. LOMBARDI

1963

Joint Nuclear Research Center
Ispra Establishment - Italy

European Scientific Information Processing Center - CETIS
(CETIS Report No 49)

LEGAL NOTICE

This report can be obtained, at the price of Belgian Francs 60, from : PRESSES ACADEMIQUES EUROPEENNES - 98, Chaussée de Charleroi, Brussels 6.

Please remit payments :

— to BANQUE DE LA SOCIETE GENERALE (Agence Ma Campagne) — Brussels — account N° 964.558,

— to BELGIAN AMERICAN BANK and TRUST COMPANY — New York — account N° 121.86,

— to LLOYDS BANK (Foreign) Ltd. — 10 Moorgate, London E.C. 2,

giving to reference : « EUR 192 . d, e — Three working-papers on the theory of computation ».

Printed by L. VANMELLE S.A. GHENT
Brussels, January 1963.

EUR 192.d,e

EUROPEAN ATOMIC ENERGY COMMUNITY - EURATOM

THREE WORKINGPAPERS ON THE THEORY OF COMPUTATION

by

L. A. LOMBARDI

1963

Joint Nuclear Research Center
Ispra Establishment - Italy

European Scientific Information Processing Center - CETIS
(CETIS Report No 49)

ON TABLE OPERATING ALGORITHMS *

1. In this paper we give examples which show how table operating algorithms can be represented in declarative or nonprocedural form. The aim of this study is to make a preliminary test on the potential applications of declarative techniques to the design of language elements having a particular but concrete purpose, in order to collect indications for the design of general purpose data systems and languages. The algorithms exhibited in this paper are only samples, and no attempt to design the best possible algorithm for each operation or to achieve completeness has been made. The methods adopted are those discussed in section 2 (Tables) of [5]. Throughout this paper italics are used loosely as descriptive elements of publication language. The representations of algorithms in the system language does not contain any italics.

The following apodictic (or primitive) functions will be utilized:

i) All usual arithmetic and comparison functions, denoted by the usual connectives. The absolute value of a will be denoted ||a.

ii) The substitute functions S_r and S_f of one argument, whose value is the record or field, respectively, located in the address specified by the argument.

iii) The include functions I_r and I_f of three arguments, which include the record or field, respectively, given as third argument into the set denoted by the first one by placing it into the

* Paper presented at the IFIP Congress, Munich, Germany, August 27 - September 1, 1962

address given as second argument, and has the first argument as value.

To each table, say $\underline{b}$, we shall associate a list called $L(\underline{b})$ which consists of:

I. A sequence of words, whose addresses are referred to as $M_{\underline{r}}(\underline{b})$ and whose contents, referred to as $N_{\underline{r}}(\underline{b})$, are the following:

$N_1(\underline{b})$ = Initial address of $\underline{b}$.

$N_2(\underline{b})$ = Final address of $\underline{b}$.

$N_3(\underline{b})$ = Length of the records of $\underline{b}$.

$N_4(\underline{b})$ = Name of the predicate in the address defining the last exclusive record.

$N_5(\underline{b})$ = Internal address of the keyfield(s) within the records.

$N_6(\underline{b})$ = Name of the function representing the integral of the normalized probability distribution of the key.

$N_7(\underline{b})$ = Lower bound of the value of the key.

$N_8(\underline{b})$ = Upper bound of the value of the key.

$N_9(\underline{b})$ = Internal address of the link field (for chained tables).

$N_{10}(\underline{b})$ = Name of the functions to be used as table scan algorithm.

$N_{11}(\underline{b})$ = Name of the functions to be used as "add record to table" algorithm,

plus others which are not directly utilized in this paper.

II. A sequence of alphabetic phrases, whose addresses and values are referred to as $P(\underline{b},\underline{c})$ and $Q(\underline{b},\underline{c})$, respectively, such that $Q(\underline{b},\underline{c})$ denotes the mode of the field $\underline{c}$ of table $\underline{b}$.

New functions of $\underline{m}$ variables can be defined by using the notation:

$f_{\underline{i}}$, $\underline{\text{name}}$, $\underline{n}$, $\underline{a}_1$, ... , $\underline{a}_{\underline{n}} = \underline{F}$

where i is an integer, name a descriptive lable of the function, n the number of components of the value of the function, a_i denotes the space (integers, rationals, addresses, truth values, etc) in which the i-th component ranges, and F is a form in the m dummy arguments b_1 , ... , b_m and their components b_1^j , ... , b_n^j.

A function value and its j-th component are denoted by writing the function letters f_i or f_i^j , respectively, followed by a sequence of m arguments. If an argument has several components, the forms representing their values are enclosed in brackets and separated by semicolons. In particular, these rules apply when recursion occurs in F.

If an argument is a form in n dummy (i.e., λ-bound) variables, these scopes will be denoted c_1, ... , c_n and their components c_1^j , ... , c_n^j . (For the discussion of the corresponding question in the case of command structured languages see [4]).

Remark. Unlike most conventional notations, this one does not assign a fixed name to each variable. Here, all free and bound variables are denoted by the letter b and c, respectively, with an integer subscript. The denotation of such identifiers depends upon the position in the representation of an algorithm where they are used.

Components of the components of function values, arguments or scopes are denoted $f_i^{j,l}$, $b_i^{j,l}$ or $c_i^{j,l}$, respectively, and so on for higher level decomposition.

2. Addres calculation functions generally have 3-component values. The first is an address and, in case the function is computable,

gives the address searched for. The second is truth valued, and is T if and only if the search is succesful, that is, if and only if the function is computable. The third is an integer and gives the number of tests which are involved by the search.

Consider a table, named table 1, with records having fixed length, sequentially allocated, which is to be searched sequentially for a record satisfying a predicate pred in its address. In order to do so, we shall use the functions

f_1, TABLESCAN 1, 3, ADDRESS ; TRUTH ; INTEGER

$\equiv \text{cond}(N_4(b_1)(b_2^1) \longrightarrow b_2 \ , \ b_3(b_2^1) \longrightarrow [b_2^1 \ ; \ T \ ; \ b_2^3] \ , \ T \longrightarrow f_1(b_1)$

$[b_2^1 + N_3(b_1) \ ; \ F \ ; \ b_2^3 + 1] \ , \ b_3)$

and

f_2, TABLE SCAN 2, 3, ADDRESS ; TRUTH ; INTEGER

$\equiv f_1(b_1, [N_1(b_1) \ ; \ F \ ; \ 1] \ , \ b_2]$

That is, this address is expressed by

f_2^1 (table 1 , pred)

If there are doubts on the existence of the record wanted, we can utilize

f_3, ADDRESS FUNCTION 1, 1, ADDRESS

$\equiv \text{cond}(b_1^2 \longrightarrow b_1^1 \ , \ T \longrightarrow U)$

and

f_4, ADDRESS FUNCTION 2, 1, ADDRESS

$\equiv f_3(f_2(b_1, b_2))$

Consider now a chained table, table 2, and let us design a more efficient algorithm than is sequential scan to locate an item whose keyfield contains datum. The first tentative address to be computed by using

f_5, TENTATIVE, 1, ADDRESS

$\equiv (N_8(b_1)-N_7(b_1)/N_2(b_1)-N_1(b_1)) * N_6(b_1)(b_2)+N_1(b_1)$

The main algorithm is represented by

f_6, CHAIN 1, 3, ADDRESS ; TRUTH ; INTEGER

$\equiv \text{cond}(S_f(N_9(b_1)+b_2^1) = U \longrightarrow b_2)S_f(b_2^1 + N_5(b_1) = b_3 \longrightarrow [b_2^1;T;b_2^3]$,

$T \longrightarrow f_6(b_1, [S_f(b_2^1 + N_9(b_1),F,b_2^3 + 1] , b_3)$

and

f_7, CHAIN 2, 3, ADDRESS ; TRUTH ; INTEGER

$\equiv f_6(b_1, [f_5(b_1,b_2);F;1] ,b_2)$

Then, the address is expressed by

f_7(table 2, datum)

In order to consolidate these two algorithms (and other algorithms which apply to tables which are differently organized) we shall use

f_8 , TABLE FUNCTION 3, 1, ADDRESS

$\equiv f_3(N_{10}(b_1)(b_1,b_2))$

In this way, provided that the system includes a wide selection of

table scanning functions, all table functions are represented by f_8. The appropriate algorithm is part of the table layout description and is selected automatically by N_{10}, while the process description, which uses f_8, is invariant with respect to such algorithm.

As further example, consider a table, table 3, suitable for binary (logarithmic) search. In this case we shall use

f_9, BINARY SEARCH 1, 3, ADDRESS ; TRUTH ; INTEGER

$\equiv$ cond(b_4=0 $\longrightarrow$ b_2, $S_f(b_2^1 + N_5(b_1))$=b_3 $\longrightarrow$ $[b_2^1;T;b_1^3]$,T $\longrightarrow$ $f_9(b_1$,

$[(b_2^1$+cond($S_f(b_2^1$+$N_5(b_1)$ < b_3 $\longrightarrow$ 1,T $\longrightarrow$ -1) * b_4;F;b_2^3+1$]$,

$b_3(b_4/2)$+cond(b_4=0 $\longrightarrow$ 0,T $\longrightarrow$ 1))

and

f_{10}, BINARY SEARCH 2, 3, ADDRESS ; TRUTH ; INTEGER

$\equiv$ $f_9(b_1$, $[(N_2(b_1)+N_1(b_1))/2;F;1]$,$b_2(N_2(b_1)-N_1(b_1))/2)$.

Here again, if N_{10}(table 3) = f_{10}, the address will be represented in the form:

f_8(table 3, datum)

Lastly, consider the example of a table 4 where the records are allocated in buckets, with keys placed within the records, and these buckets are chained. In order to locate a record having datum as value of the key we should use

f_{11}, CHAIN ADDRESS 1, 3, ADDRESS ; TRUTH ; INTEGER

$\equiv \text{cond}(b_2^2 \longrightarrow b_2, S_f(b_2^1+N_9(b_1)=U \longrightarrow [U;F;b_1^3] , T \longrightarrow f_{11}(b_1, f_{12}$

$(b_1, S_f(c_1+N_5(b_1)=b_3, S_f(b_2^1 + N_9(b_1)), b_3)$

where

f_{12}, CHAIN AUXILIARY 1, 3, ADDRESS ; TRUTH ; INTEGER

$\equiv f_1(b_1, [b_3;F;1] , b_2)$

and

f_{13}, CHAIN ADDRESS 2, 3, ADDRESS ; TRUTH ; INTEGER

$\equiv f_{11}(b_1, f_{12}(b_1, S_f(c_1+M_5(b_1)=b_2, N_6(b_1)(b_2)), b_2)$

so that, if N_{10} (table 4) = f_{13}, the address required is again f_8(table 4, datum).

3. A table function (see [5] , section[2]and [3]), can be represented by using

f_{14}, TABLE FUNCTION 1, 1, $Q(b_1, b_3)$

$\equiv \text{cond}(b_2^2 \longrightarrow S_f(b_2^1+b_3), T \longrightarrow U)$

and

f_{15}, TABLE FUNCTION 2, 1, $Q(b_1, b_3)$

$\equiv f_{14}(b_1, f_8(b_1, b_2), b_3)$

so that the value of the field, field 1, of the record of table 5 whose keyfield contains datum, is

f_{15} (table 5, datum, field 1)

and the value of the field 2 of the record of table 6 whose key satisfies the predicate pred. 2 is

f_{15}(table 6, pred 2, field 2)

The design of table scanning algorithms, similar to f_4, f_7, f_{10} or f_{13}, able to operate on tables organized according to the standard RAMAC allocation pattern or on threaded lists are left to the reader as exercises.

Consider the problem of introducing a new item record into table 6 which is sequentially ordered with respect to its keyfield. This operation is described by means of

f_{16}, TABLE ADD 1, 2, ADDRESS ; TABLE

$\equiv \left[I_f(L(b_1),M_2(b_1),N_2(b_1)+1);\ f_{17}(b_1,N_2(b_1),b_2)\right]$

where

f_{17}, TABLE ADD 2, 2, ADDRESS ; TABLE

$\equiv \text{cond}(S_f(b_2+N_5(b_1)) < b_3) \vee (b_2 \neq M_1(b_1) \longrightarrow I_r(b_1,b_2+N_3(b_1),b_3),$

$T \longrightarrow f_{17}(I_r(b_1,b_2,S_r(b_2+N_3(b_1))),b_2-N_3(b_1),b_3)$

so that the above algorithm can be represented as

f_{16}(table 6, record)

Again, in order to make process descriptions invariant with respect to table algorithms, provided that a selection of algorithms able to operate like f_{16} on tables with different structures is available, we shall use

f_{17}, TABLE ADD 3, 2, ADDRESS ; TABLE

$\equiv N_{11}(b_1)(b_1,b_2)$

The function f_{17} has the same level of universality as f_8.

As problems for the reader we suggest to develop:

i) A function of a table and a predicate whose value is the first argument without the records satisfying the second.

ii) A function of a table, a predicate, an address and a literal whose value is the first argument where the value of the fields having the third argument as internal address of all records whose address satisfies the second is replaced by the fourth.

Another useful exercise consists of adapting all algorithms outlined so far to the case of tables with records whose variable length is specified in a field.

4. The implementation of the language elements described above obviously requires the availability of a real or simulated computer able to evaluate all apodictic functions described in i - iv (section 1) and to link defined functions with provision for assigning both arguments and scopes. The design of such a computer [2] is the main problem of modern data system science and its discussion cannot be encompassedwithin the scope of the present communication. Besides this major point, we should mention that the implementation of these language elements requires an assembler which would not differ substantially from standard assemblers for procedural languages, which should reduce forms to the standard Lukasiewicz format, allocate all forms of which the description

of a process consists, call and allocate all system forms from the external storage devices and replace or supplement the corresponding function letters with their addresses, replace all symbols with their value derived from the layout tables, construct the layout tables L(b) from layout descriptions, etc. The only point which is worth mentioning is that the assembler should not replace the M_r and N_r with constants: instead, it should replace them with appropriate table functions, because the contents of the layout tables is subject to changes during the evaluation of the process description.

During evaluation the tables L(b) should be handled like any other table. We should draw the attention to one of the main difficulties of recursive programming, which has been pointed out by Dr. Mauro Pacelli in 1961 and which comes up through this set of examples: that is, the problem of optimizing the evaluation of sets of forms having common subforms which should be evaluated for the same arguments. When using command structured languages, it is often easy to extract subprocedures from code sections in order to minimize the time of execution: that is, programs are optimized by means of a handtayloring of the flow of the control. In the case of declarative languages without recursion, it is easy to have optimisation performed by the machine by means of appropriate precedence analysis. But when recursion occurs, then the question of common evaluation comes up in connection with a priori unbounded sets of incarnations of forms, and to the best of our knowledge there is no universal optimisation technique available. One solution to this problem, that we have implicitly proposed in this paper by adopting it in many instances, consists of extensively utilizing Cartesian products of

of values of functions, and conversely, projections of such products, that is, functions with multiple-component values and single components of such values. This yields the need for multiple-component arguments and scopes of functions.

However, we do not know of any method, but hand-tayloring, of consolidating form evaluations in order to minimize evaluation time. On the other hand, the corresponding problem for procedural languages, that is, the problem of the design of analysis codes, is also practically unsolved. From these indications we can infer that probably one of the basic skills of the future declarative computer designers and programmers will be the familiarity with these set-generating and projecting techniques.

Another technique of optimizing, which applies when a subform, say sf, occurs twice or more often within a form f in t arguments, consists of designing a new form nf in t arguments, such that

$$nf \equiv mf(b_1,\ldots,b_r,sf(b_1,\ldots,b_r))$$

where mf is the form in t + 1 arguments which is obtained from f by replacing grafically all occurrences of sf with $b_t + 1$, and using nf instead of f. A simple instance of this technique can be found in the design of ADDRESS FUNCTION 1, section 2.

References.

(1) McCarthy John, A basis for a mathematical theory of computation, Proc. Western Joint Computer Conference 1961 (Los Angeles, Cal., 1961).

(2) Gilmore Paul C., An abstract computer with a Lisp-like machine language without a label operator, Proc. Symp. on the relations between formal systems and computer languages, Blaricum (Holland) October 1961.

(3) Lombardi, Lionello. Theory of Files, Proceedings 1960 Eastern Joint Computer Conference, paper 3.3 (New York, N.Y., 1960).

(4) Lombardi, Lionello. System handling of functional operators. J. Assoc. Comput. Mach. 8 , 2 , 168-185 (1961).

(5) Lombardi, Lionello, Mathematical Structure of Nonarithmetic Data Processing Procedures, J. Assoc. Comput. Mach., 9 1 136-159 (1962).

(6) Lombardi, Lionello. Inexpensive punched card equipment. J. Machine Accounting 12 8 12-18 (August 1961).

(7) Lombardi, Lionello. Logic of automation of system Communications, J. Machine Accounting. (13, 4 (1962), 18-29.

THE DECLARATIVE CONTROL OF THE DATA FLOW BY MEANS OF RECURSIVE FUNCTIONS *

1. Introduction

This note is devoted to the discussion of algorithms for file processing which are represented in terms of a language without command structures, that is, a declarative language.

So far, most of the effort in the area of declarative languages has been in the field of the representation of processes whose results, or output, are numbers or small, well defined sets of numbers. Consequently, such effort has been mainly directed towards the development of morphologies and syntaxes for writing and semantics for evaluating forms whose values range over a set of numbers. File processes do not belong to this category, since their result, or output, are large sets of records distributed in sets of output files, and the main difficulty in achieving this result is the proper organization of such files. Consequently, since declarative languages consist of methods of representing processes as functions of the data whose values are the results, declarative techniques to represent file processes must be based on forms whose values range over spaces of sets of files. Due to the level of repetitivity involved by file processes, such forms

---

* Paper presented at the Symposium on Symbolic Languages in data Processing, International Computation Centre, Roma, Italy, March 26-31, 1962

should be designed in a way to allow for recursion.

While the application of recursive functions to forms able to represent operations on random access files or tables is discussed in [13] , this note is concerned with coordinated or collinear file processes [16], that is, with the file processes which take place in magnetic tape processors. The results of the theory of files [7 , 16] are extensively used, with special regard to the standardization and optimization of the input and the predicate-controlled output [7, section 3] . In fact, without the tool of the filetheoretical approach, the problem stated at the end of the preceding paragraph would be very hard to solve.

Since this is a report on preliminary investigations aiming to show the possibility and the basic elements of a declarative approach to the representation of file processes, rather than a manual for a declarative file processing language or for a form-evaluating file processing computer, the schemes discussed in [16] have been considerably simplified. In particular, the stream and bundle level have not been considered, thus implying, among other things, the need for each logical file flowing through an individual input-output unit, which is possible only if there is a large number of such units available. Furthermore, among the file indicators, only the existence and the left and right derivatives of the input files are considered, while the input-output and validity indicators, as well as all indicators of the output files, are provisionally dropped. In fact, these useful features can be added later without any need for major theoretical advances,

while their treatment here would make the discussion considerably more complicated.

The file processing algorithms considered here provide for coordinating the input by means of the function $f_{input\ 2}$ and computing the contents of the output records and coordinating the output by means of the function $f_{output\ 2}$. These two functions are cascaded, in the sense that the second one, which gives the timing to the whole system, calls the first one. One of the most important things that are brought up in connection with this, is that, when two functions are cascaded, the first one should have a value with several components, some of which are the data that it was supposed to produce and pass to the second one, and others have no other purpose than to enable the second one to assign parameters to the first at the time of its following incarnation. In other words, the link between these two particular functions is a prototype of the nature of communication which should exist between independent moduli — or submachines — of a computer with declarative logic. Input and output buffering is not discussed here; the entering and filing of records is performed by not better specified functions $f_{physical\ input}$ of one variable and $f_{physical\ output}$ of two variables, whose value are the next-coming record of the input unit specified by the variable and the contents of the output unit specified by the first variable supplemented with the second one, respectively. These two functions are somewhat eterogeneous with respect to the remainder of the system, because, as it will appear, in order to be consistent, the contents of I-O units should be represented as lists and operated with composition and decomposition operations. This will be done in the near future

probably without major difficulties, while drafting a complete recursive file processor, based on six cascaded levels: physical output, output buffering, output computation and coordination, input coordination, input buffering, physical input. However, in this first essay we have bound our investigation to the two central levels, which give rise to the main difficulties, and to the link between them, which is the most critical. No use is made here of the set inclusion operator of order three, which has been widely used in [13], and is essentially a particular interpretation of Gilmore's dynamic load operator of order two [4] . It has been possible here to replace this problemraising operator with composition operations each time the opportunity of using it come up.

Compositions and decompositions of lists are used extensively in order to build composite values and parameters of functions. Such techniques are derived from those developed by McCarthy [3] , though they are used here in a different way and for a different purpose. An example of a composite value form built in this way is the value of $f_{input\ 2}$, the fifth component of which is the whole machine status, while the other eight components contain information sufficient to control its subsequent incarnation. This technique is also used by $f_{output\ 2}$ in order to decode the value of $f_{input\ 2}$ that it receives through $f_{input\ 1}$.

The problem of developing semantical elements for decomposition and composition operators is not discussed here. The solution of this problem will be a particular case of a semantics able to define elements of new spaces and operations thereon in terms of the previously available spaces and operations, in a way such that

operation letters (or function letters) are morphologically independent of the structure of the operands, while an appropriate meaning is assigned to them for each allowable configuration of such structure. A considerable effort is currently being devoted to the solution to this last problem, which is the key question of the theory of computation. This author will soon report on the results of his investigation, based on extensive development and theorization of some features of the B5000 and KDF9 computers. Care has been exercized here to use only the particular class of recursive functions called external recursive [13,15] , because a much easier syntax to evaluate them, based on the so-called discharge stack, is possible, as shown in the second part of [15].

2. NOTATION

Only very short explanations are given here on the notation used, since the first part of [15] is devoted to its complete discussion. It should be noticed that practical considerations have suggested some very slight deviations from the notations used in the report [13] , written four months earlier than the present one.

All function letters are denoted by the letter f with a subscript. Besides specifying such functions, these subscripts may be mnemonic and connote the practical use of the function letter. However, functions which are usually denoted with special infixes, such as + or - , are here denoted as usual, for the sake of readability.

The definition of a function of order $\underline{n}$ has the layout $f_A = \underline{b}$, where f_A is the function letter involved and $\underline{b}$ is a form in the $\underline{n}$ variables $x_1, x_2, \ldots, x_{\underline{n}}$, where x_i always denotes the i-th argument (free variable) of the currently defined function. The form $\underline{b}$ can contain calls for other functions, the one currently defined inclusive. These calls have the layout $f_B(d_1, d_2, \ldots, d_{\underline{m}})$, where f_B is the function letter of the function called, $\underline{m}$ its order, and the d_j are forms in the $\underline{n}$ variables x_i, which have the meaning specified above. Whenever the value of any of the d_j for given $\{x_i\}$ is a function of a finite sequence of (λ-bound) variables, the h-th of them is denoted y_h. Unlike most languages for representing algorithms, where each variable, free or bound, has a fixed name assigned to it, in this language the names of free and bound variables are always x and y, respectively, with

an integer subscript, and the denotation of such names depends on the position where they are used. The reason for the adoption of this method is its orientation towards address-free evaluation logics based on stack operation, [5 , 18], while the conventional approach, by means of the standard gimmick of associating storage areas with names, is oriented towards the execution of commands on addresses [1].

McCarthy's conditional function f_{cond} of three variables [3], whose value is x_2 or x_3, depending on whether or not x_1 is true, is apodictic in this system and referred to sometimes with the redundant notation $f_{cond}(d_1 \rightarrow d_2, T \rightarrow d_3)$, for improving readability. A list of n elements is represented by the integer n followed by the n elements, all separated by commas. The value of the apodictic function f_{comp} of order $n+1$ is the list which its arguments represent, while the value of the apodictic function of order one f_{dec} , whose argument is a list (we should rather say "has the dimensions of a list"), is the set of the elements of this list. The value of the apodictic function of order 2 f_{el}, whose first argument x_1 is a positive integer and whose second argument x_2 is a list, is the x_1-th element of x_2. By using recursively these three basic functions, called composition, decomposition and element, respectively, it is easy to build two further handy list operating functions of order two, f_{first} and f_{last} , whose value is the set of the first or last x_1 elements of the list x_2, respectively. The value of the apodictic function f_{numb} of order one will be the number of elements composing its argument, which is a list.

Recursion is allowed in the function definitions, that is, if such a definition has the form $f_A \equiv \underline{b}$ calls for f_A can be contained in the form $\underline{b}$.

If we establish a weak precedence relation in the space of the function letters involved by an algorithm , such that $f_0 \leqslant f_m$ if there is a sequence $f_0, f_1, \ldots, f_m$ where f_i is used in the definition of f_{i-1} then, if this relation yields a partial weak order, we shall say that the algorithm is based on a class of individual recursive functions (the case of the strong order would imply the absence of recursion). As it was remarked by Pacelli, this way of defining algorithms is possibly less powerful than McCarthy's [3], who explicitly introduces sets of recursive functions defined through systems. However, since the logical problem of comparing the two classes of recursive functions to which this difference has given rise is open and goes far beyond the specific scope of this essay, care has been exercized here to base the file processing algorithm on a class of individual recursive functions.

A handy gimmick which makes it easier to read the definitions of recursive functions consists of replacing with a dash (possibly in parentheses to avoid confusion with the minus mark) some appearances of x_j in d_j in all function definitions of the type

$$f_A \equiv f_B\ (d_1, \ldots, d_m) \tag{2.1}$$

or

$$f_A \equiv f_{cond}(d_A \rightarrow d_B, T \rightarrow f_B(d_1, \ldots, d_m)) \tag{2.2}$$

The letter z with a subscript (possibly mnemonic or descriptive) is called shorthand, that is, it replaces graphically the occurrence of an arbitrary writing, which is associated with it by

means of a shorthand definition having the layout

$$z \equiv d$$

where z is the shorthand and d is the writing that z stands for. Such shorthand definitions have been already introduced in [8] and [17], where they were called symbols.

Despite the confusion between function and shorthand definitions that the common layout and the common use of the mark " $\equiv$ " might yield, one should keep in mind the deep semantic difference between them: they denote functional equivalence and graphical identity, respectively, and the occurrence of $\equiv$ is related to the McCarthy's label operator [3] in the first case, while it denotes plain identity or replacement in the second. The limited purpose of shorthands is to simplify the design of algorithms and to cut the size of the (real or simulated) computers which carry them out.

The letter v with a descriptive subscript denotes an off-algorithm datum or a datum which for some reason is left unspecified. In this example, this notation will be used to denote the elements of the programs which the algorithm or abstract machine should execute (see section 4).

The letter w with a subscript is used loosely in this essay in order to build explanatory examples.

In the sequel, Ω will denote an empty set.

3. The file processing algorithm

Each currently available record is presented in the system as a list of seven items, namely

$$f_{comp}(7, w_{record}, w_E, w_L, w_R, w_{PK}, w_{CK}, w_{FK})$$

where w_{record} is the record represented as list of the contents of its fields, w_E, w_L and w_R are values of the existence indicator and the left and right derivative at this record of the file to which it belongs, respectively, while w_{PK}, w_{CK} and w_{FK} are the keys of the preceding, current and following record if the file involved, respectively. In the sequel, the key constructing function f_{key} of one argument (a record) will be supposed unique for all files, for simplicity. We shall also assume that the last exclusive records of all files will consist of the literal EOF.

Let

$$f_{min} = f_{min\ 1}(-,-,1,x_1(f_{el}(1,x_2)),f_{num}(x_2))$$

and

$$f_{min\ 1} = f_{cond}((x_3=x_5) \longrightarrow x_4, T \longrightarrow f_{min\ 1}(-,-,(-)+1,f_{cond}$$

$$((x_4 < x_1(f_{el}(x_3,x_2))) \longrightarrow x_4, T \longrightarrow f_{el}(x_3,x_2)))$$

Then, f_{min} is a function of order two, which computes the minimum value of the function x_1 of the elements of the list x_2.

Let

$$f_{input\ 1} \equiv f_{input\ 2}(-,-,(f_{min}(f_{key}(y_1),x_3) \neq x_5),1,\Omega\, x_3,x_4,$$

$$(f_{key}(f_{el}(1,x_3)) \neq x_1) \vee (f_{el}(2,f_{el}(1,x_2))),-)$$

With reference to [17] (section 3), x_1 will denote the contents of the current key register (CKR), x_2 the complete set of the records available at the previous pulse (the dimensions of x_2 thus being the ones of a list of lists of lists of literals of unspecified dimensions), x_3 and x_4 denote the contents of the lower and upper logical one-record buffers (required to compute the right derivatives), x_5 is the value that the current key register had at the previous pulse and x_6 is the list of all input files.

The function $f_{input\ 2}$ called above is defined as

$$f_{input\ 2} = f_{cond}((x_4 > f_{num}(x_{10}) \rightarrow f_{comp}(10,x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,x_9,x_{10}),$$

$$T \rightarrow f_{input\ 2}(-,-,-,(-)+1,f_{comp}(x_4,f_{dec}(x_5),f_{cond}(z_2 \rightarrow f_{comp}$$

$$(7,stop,T,U,U,U,U,U),T \rightarrow f_{cond}(x_9 \rightarrow z_1,T \rightarrow f_{comp}(7,f_{el}(x_4,x_6),$$

$$T, \rightarrow f_{el}(2,f_{el}(x_4,x_2)),x_1=f_{key}(f_{el}(x_4,x_7),f_{el}(6,f_{el}(x_4,x_2),$$

$$f_{el}(7,f_{el}(x_4,x_2),f_{key}(f_{el}(x_4,x_7))))),f_{cond}(x_9 \rightarrow f_{cond}(f_{el}(x_4,x_7)$$

$$= EOF \rightarrow EOF,T \rightarrow f_{el}(x_4,x_7),T \rightarrow -,f_{cond}(x_9 \rightarrow f_{cond}(f_{el}(x_4,x_7)$$

$$\rightarrow -,T \rightarrow f_{physical\ input}(f_{el}(x_4,x_{10})),f_{cond}((f_{key}(f_{el}(x_4,x_6))$$

$$=x_1) \wedge f_{el}(2,f_{el},(x_4,x_2)) \rightarrow F,T \rightarrow -,(f_{key}(f_{el}(x_4,x_6)) \neq x_1) \vee$$

$$((f_{el}(2,f_{el}(x_4+1,x_2))) \wedge (\neg x_8)),-)$$

where

$$z_1 \equiv f_{cond}(x_3 \rightarrow f_{el}(x_4,x_2,T \rightarrow f_{comp}(7,f_{first}(1,f_{el}(x_4,x_2)),F,f_{last}(5,f_{el}(x_4,x_2)))))$$

and

$$z_2 \equiv (f_{el}(x_4,x_6) = EOF)$$

Here, x_1 is the contents of CKR, x_2 is the set of the available records at the time of the preceding pulse, x_3 is a signal which is on during the first pulse of each phase (see [7],[17]) x_4 is a standard recursion count, x_5 is the part of the machine status elaborated until the previous incarnation of $f_{input\ 2}$, inclusive, x_6 and x_7 are the contents of the lower and upper buffer, respectively, x_8 is a signal which triggers the entrance of a record which clobbers another one belonging to the same file and having the same key (see [17] , pages 152-153), x_9 is a bit which summarizes the conditions under which a new record should not be entered, and x_{10} denotes the set of all input files.

The value of both $f_{input\ 2}$ and $f_{input\ 1}$ is a list of ten elements, of which the fifth is the set of records available to the pulse of which the evaluation of $f_{input\ 1}$ is the start [17, section 3]. We shall utilize the auxiliary function f_{stop} defined by

$$f_{stop} \equiv f_{cond}(\neg f_{el}(x_2,x_1) \rightarrow F,T \rightarrow f_{cond}((x_2 > f_{numb}(x_1)) \rightarrow T,T \rightarrow f_{stop}(x,x_{2+1})))$$

where x_1 is a list and x_2 an internal count. Furthermore, we shall

need

$$f_{\text{list 1}} \equiv f_{\text{list 2}}(-,-,1,x_2(1))$$

where x_1 is an integer, x_2 a function of an integer variable, and

$$f_{\text{list 2}} \equiv f_{\text{cond}}((x_3 > x_1) \rightarrow f_{\text{dec}}(x_4), T \rightarrow f_{\text{list 2}}(-,-,(-)+1, f_{\text{comp}}(x_3, f_{\text{dec}}(x_4), x_2(x_3))))$$

where x_1 denotes the number of recursions, x_2 is a function of an integer, x_3 an internal count and x_4 the issue of the preceding incarnation, and

$$f_{\text{file 1}} \equiv f_{\text{file 2}}(x_1, 1, f_{\text{cond}}(f_{\text{el}}(1, f_{\text{el}}(1, x_1))(x_3, x_4, x_5)) \rightarrow f_{\text{physical file}}(x_2, f_{\text{el}}(2, f_{\text{el}}(1, x_1))(x_3, x_4, x_5)), T \rightarrow x_2), x_2, x_3, x_4, x_5)$$

where x_1 is the group of FCE related to the current incarnation of f_1, x_2 is the output file involved, x_3 is the list of available records, x_4 the list of internal variables, x_5 the stop indicator [17] and

$$f_{\text{file 2}} \equiv f_{\text{cond}}((x_2 > f_{\text{numb}}(x_1)) \rightarrow x_4, T \rightarrow f_{\text{file 2}}(-,(-)+1, f_{\text{cond}}(f_{\text{el}}(1, f_{\text{el}}(x_2, x_1))(x_5, x_6, x_7)) \rightarrow f_{\text{physical file}}(x_4, f_{\text{el}}(2, f_{\text{el}}(x_2, x_1)), (x_5, x_6), T \rightarrow x_4), -, -, -, -)$$

where x_1 has the same connotation as for f_{file1}, x_2 is a count, x_3 is the main issue of the preceding incarnation, x_4 the output file involved, and x_5, x_6 and x_7 have the same meaning that x_3, x_4 and x_5,

respectively, had in the definition of $f_{file\ 1}$.
The definition of the last two functions requires that the file control predicates [17] are organized in the form of a list of as many elements as are the output files, where the i-th element is the list of all the file control predicates related to the i-th output file. Furthermore, each such predicate should be presented as list of two elements, of which the first is the flow control predicate as defined in [17] and the second is a form whose value is the record to be filed, represented as list of the contents of its fields.

The basic function used to compute the contents of the records and of the output files, that is, to control the output, is

$$f_{output\ 2} \equiv f_{cond}(x_4 \rightarrow x_5, T \rightarrow f_{output\ 2}(f_{input\ 1}(f_{min}(f_{key}(y_1), el(6,x_1)),$$

$$f_{el}(5,x_1), f_{el}(6,x_1) f_{el}(7,x_1) x_8),$$

$$f_{comp}(f_{list\ 1}(f_{numb}(x_6), f_{el}(y_1,x_6)),$$

$$f_{min}(f_{key}(y_1), f_{el}(6,x_1)),$$

$$f_{stop}(f_{el}(5,x_1), 1),$$

$$f_{comp}(f_{numb}(x_5), f_{list}(f_{numb}(x_5),$$

$$f_{file\ 1}(f_{el}(y_1,x_7), f_{el}(y_1,x_5) f_{el}(5,x_1),$$

$$x_2,x_4)), x_6, x_7, x_8)$$

where x_1 is the value of $f_{input\ 1}$, x_2 is the list of the values of the internal variables at the previous incarnation, x_3 is (CKR), x_4 an internal signal which denotes the occurrence of the last pulse, x_5 is the list of the output files at the previous incarnation, x_6 is the list of forms which assign values to the temporary variables, x_7 is the list of the flow control predicates [17] and x_8 is the list of the input files.

In [17] the concept of pulse as atomic file processing action was presented and discussed. With the present algorithm we can give a simple formal definition of this basic concept of file processing:

a pulse is an incarnation of $f_{output\ 2}$.

The general controlling function of this algorithm, which, if we had adopted a tree structure rather than a machine-oriented linear notation, would be at the root, is

$$f_{output\ 1} \equiv f_{output\ 2}(z_3, x_3, 0, F, f_{list}(f_{el}(1, x_2) f_{physical\ file}(f_{el}(y_1, x_2),$$

$$f_{el}(y_1, x_6))), x_2, x_4, x_5, x_1)$$

where x_1 and x_2 are lists of input and output files, respectively, x_3 and x_4 are the lists of the initial values and of the forms defining the temporary variables, respectively, x_5 is the list of the flow control predicates with the corresponding record value associated (see above in this section), x_6 is the list of the contents of the labels of the output files (which, in this algorithm, are assumed independent of the input), and

$$z_3 \equiv f_{list}(f_{el}(1, x_1), f_{comp}(7, U, F, U, U, U, 0, 0))$$

is the initial machine status.

Remark. The usage of $f_{output\ 1}$ made here is to a certain extent similar to the one of a special purpose supervisory control or monitor in systems programming for conventional, command-structured algorithms or computers. In the case of declarative representation there is no difference between monitors and other functions, since each function monitors the incarnation of those on the basis of which it is defined, and each function which is used as monitor by being at the root or lowest level in an algorithm can be on a branch of other algorithms.

4. The programming language

The implementation of the above algorithm on a computer with wired or simulated stack logic [5] able to evaluate on either wired or programmed basis all apodictic functions mentioned in section 2 consists of writing the functions and shorthands defined in section 3 into its memory or of preparing microprogramming plans consisting of their definitions. A program for this new computer consists of a function definition of the type

$$f_{\text{some process}} \equiv f_{\text{output 1}}(v_1, v_2, v_3, v_4, v_5, v_6) \qquad (4.1)$$

while its operation consists of letting it evaluate such function. The meanings of the v_i are the following:

v_1 : list of the names of the input files

v_2 : list of the names of the output files

v_3 : list of the initial values of the intermediate variables

v_4 : list of the forms which assign values to the intermediate variables

v_5 : list of lists of flow control predicates and record evaluation forms

v_6 : list of labels of the output files

The programmer of this algorithm is supposed to write the v_i according to the morphological rules of the system. For example, if he is programming a process involving five intermediate variables, the initial values of the first four of them being 1, $f_{\text{some function}}$, T and BUMBLEBEE, respectively, while the last one is initially not defined, then, at the place of v_3 he should write

$$f_{\text{comp}}(5, 1, f_{\text{some function}}, T, \text{BUMBLEBEE}, U)$$

For the design of v_1, v_2, v_4, v_5 and v_6 analogous rules should be followed. Regarding the lists v_4 and v_5, these items stand for forms in three bound variables, namely the list of the available records, the list of internal variables, and the STOP indicator. They will be denoted y_1, y_2 and y_3, respectively. Reference to elements of such variables while preparing the lists v_4 and v_5 should be made considering this. For example, the contents of the third field of the fifth input file will be denoted $f_{el}(3,f_{el}(1,f_{el}(5,y_1)))$, while the right derivative of the same file is denoted $f_{el}(4,f_{el}(5,y_1))$. The sixth internal variable is denoted $f_{el}(6,y_2)$.

The notation rules of the preceding paragraph hold for machine language programming. However, it is easy to write a program for mnemonic translation, in order to enable the programmer to use a notation for fields, variables and indicators [7, 16] , similar to the one of the Algebraic Data System Language. It would be an interesting exercise to write such a program in a declarative language similar to the one used in this paper.

In the program (4.1), the function $f_{\text{some process}}$ has order zero. Its function letter plays a role rather similar to the one played by program-identification cards in conventional programming.

Conclusion

The main direct advantage of this approach to file processing is the extreme simplicity of the programming language, which depends on the fact that the representation of any file process is independent of the procedure by which it is carried out.

Despite the fact that the main fields of application of file processing techniques, which are machine accounting and linguistics, involve little mathematics, the design of this algorithm or similar ones requires some applications of formal logic. In other words, the alleged non-mathematical nature of file processes is shown to depend only on the conventional procedural approach to them, while this method, based on the analysis of the inherent logical structure of such processes, allows for fruitful applications of mathematics to their representation.

While conventional algorithm and computer design involves the application of nothing beyond than propositional logic, the declarative approach based on recursion requires first order functional calculus. So far, this author has not yet met an algorithm design problem where second order functional logic is needed.

The need for specialized mathematical tools yielded by this approach affects only the design of algorithms. In contrast, no knowledge beyond elementary Boolean operations is required in production programming, where the level of skill that the programmer should have is considerably lower than the one of a conventional business programmer, such as, for example, a COBOL programmer.

Bibliography

[1] Goldstine, H.H. and von Neumann, J. - Planning and coding of problems for an electronic computing instrument, Institute for Advanced Study, Princeton, N.J., 1947.

[2] McCarthy, J. - A basis for a mathematical theory of computation, paper 5.3, Proc. 1961 W.J.C.C., Los Angeles, Cal., (1961)

[3] McCarthy, J. - Recursive functions of symbolic expressions and their computation by machine, Comm. Assoc. Comput. Mach., 3 , 4 (1960), 184-195.

[4] Gilmore, P.C. - An abstract computer with a LISP-like machine language without the label operator, IBM J. Res. Devel., (to appear).

[5] Pacelli, M. - Tecniche di traduzione automatica, Atti del convegno sui linguaggi simbolici, Pisa, January 1962 (in press)

[6] Church, A. - Introduction to mathematical logic, Princeton Univ. Press, Princeton, N.J., (1956).

[7] Lombardi, L.A. - Theory of files, Proc. 1960 W.J.C.C., paper 3.3., New York, N.Y., 1960.

[8] Lombardi, L.A. - System Handling of Functional Operators, J. Assoc. Comput. Mach., 8 , 2 (1961), 168-185.

[9] Lombardi, L.A. - Inexpensive punched card equipment, J. Machine Accounting, 12 , 8 (1961), 11 - 18.

[10] Lombardi, L.A. - Logic of automation of system communications, J. Machine Accounting, 13 , 4 (1962) , 18 - 29.

11 Lombardi, L.A. - On a problem of punched tape-to-card conversion, J. Machine Accounting, 13 , 5 (1962).

12 Lombardi, L.A. - Nonprocedural data system languages, (invited paper), Proc. 16th. National Conference of A.C.M., Los Angeles, Calif., (1961).

13 Lombardi, L.A. - On table operating algorithms, Proc. 2nd. IFIPS Congress, München, Germany, August 1962 (to appear)

15 Lombardi, L.A. - Zwei Beiträge zur Morphologie und Syntax deklarativer Systemsprachen, DMV-GAMM Tagung, Bonn, Germany, April 1962.

16 Lombardi, L.A. - Mathematical Structure of Nonarithmetic Data Processing Procedures, J. Assoc. Comput. Mach., 9 , 1 (1962), 136 - 159.

ZWEI BEITRAGE ZUR MORPHOLOGIE UND SYNTAX DEKLARATIVER SYSTEMSPRACHEN *

Vom Verfasser wird eine auf die Darstellung der Algorithmen in deklarativer (d.h. befehlloser) Form gegründete allgemeine Theorie der Rechnung entwickelt. Zwei ausgewählte Aspekte davon werden in diesem Vortrag in vereinfachter Form kurz besprochen.

1.) Der erste Beitrag betrifft die Bezeichnung der Variablen. In gewöhnlichen Sprachen ist jeder Variablen ein Name als feste Bezeichnung zugeordnet. In unserem Fall hängt die Bezeichnung vom Ausdruck, in welchem sie vorkommt, ab. Die Beschreibung eines beliebigen Algorithmus in unserem System besteht aus einer Reihe von Funktionsdefinitionen, von welchen jede die Form

$$f^{n}_{name} \equiv w \qquad (1.1)$$

hat, wobei name die Bezeichnung der Funktion, n ihre Ordnung (d.h., Anzahl der freien Variablen) und w einen Ausdruck in m Namen $x_1, x_2 \ldots\ldots x_m$ von Variablen bedeuten; f^{n}_{name} wird als Funktionsbuchstabe von (1.1) bezeichnet.

Eine Erwähnung einer Funktion, deren Funktionsbuchstabe f^{t}_{ir} ist, im Ausdruck w von (1.1), wird

$$f^{t}_{ir}(w_1, w_2, \ldots\ldots, w_q) \qquad (1.2)$$

---

* Unterlagen vorgelegt bei der DMV - GAMM Tagung in Bonn, Deutschland, vom 24-28 April 1962

geschrieben, wobei $q \leqslant t$. Jedes w_i kann hier entweder der Buchstabe η oder ein Ausdruck sein, welcher zur i-ten Variablen x_i von f_{ir}^t in der Verkörperung, die durch diese Erwähnung entsteht, als Wert gegeben wird. Jeder solchen Variablen x_j, bei der $j > q$ oder $w_j = \eta$ ist, wird kein Wert zugeordnet, d.h. sie wird eine Variable des Wertes des Ausdruckes (1.2). Während der Auswertung bekommt sie eine neue, geeignete Bezeichnung.

Zum Beispiel, wenn

$$f_a^2 \equiv x_1 + 2x_2$$

dann bekommt man durch Auswertung von $f_a^2(\eta, 3)$ den Wert $x_1 + 6$, durch die Auswertung von $f_a^2(10, \eta)$ oder $f_a^2(10)$ den Wert $10 + 2x_1$.

Der Vorteil dieser Notation vom Standpunkt der Darstellung der Algorithmen besteht darin, daß ein kleiner Variablen-Wortschatz ausreichend ist.

Vom Standpunkt der Auswertung mit einer gewöhnlichen "Stapel-Logik" (stack-Logik) besteht der wichtigste Vorteil darin, daß der Wert jeder beliebigen Variablen, die x_i bezeichnet wird, sich immer im Stapel (stack) an der i-ten (logischen) Stelle unter dem Parameteranzeigeregister (parameter point+ recorder) befindet.

Als Beispiele werden hier wie in [2] Algorithmen für die angenäherte Berechnung des einfachen Integrals von f(x) über das Intervall [a,b] mit der Formel $I = (\sum_{i=1}^{100} f(a+i.\delta)).\delta$, wo $\delta = (b-a)/100$ und doppelter Integrale durch zweifache Integrierung dargelegt.

Das einfache Integral $\int_{x_1}^{x_2} x_3(x_4)\,dx_4$ wird als Funktion dreier Variablen in der Form

$$f_1^3 \equiv ((x_2-x_1)/100)\circ f_2^5((x_2-x_1)/100, x_2, x_3, x_1, 0) \qquad (1.3)$$

dargestellt, wo

$$f_2^5 \equiv (x_2=x_4) \longrightarrow x_5, T \longrightarrow f_2^5(x_1, x_2, x_3, x_4+x_1, x_5+x_3(x_4)) \qquad (1.4)$$

eine rekursive Hilfsfunktion in der in [1] erklärten Form ist.

Das Integral $\int_{x_1(x_4)}^{x_2(x_4)} x_3(x_5, x_4)\,dx_5$

wird

$$f_3^4 \equiv f_1^3(x_1(x_4), x_3(x_5, x_4)) \qquad (1.5)$$

ausgedrückt, während das doppelte Integral

$$\int_{x_1}^{x_2} \left(\int_{x_3(x_7)}^{x_4(x_7)} x_5(x_6, x_7)\,dx_6 \right) dx_7$$

in der Form

$$f_5^5 \equiv f_1^3(x_1, x_2, f_3^4(x_3(x_7), x_4(x_7), x_5(x_6, x_7), x_7)) \qquad (1.6)$$

als Reduktion zu einfachen Integralen dargestellt wird.

Wenn wir zum Beispiel die vom Parameter r abhängige Funktion $(\sin(y_1 \cdot y_2))/(y_1 + r \cdot y_2)$ auf dem ebenen Gebiet $1/2 \leq y_1 \leq 1$, $0 \leq y_2 \leq \sqrt{1-(y_1)^2}$ integrieren wollen, dürften wir die damit erzeugte Funktion einer einzigen Variablen

$$f_6^1 \equiv f_5^5(5, 1, 0, f_{qdrwrz}(1-x_3 \uparrow 2), f_{sin}(x_2, x_3)/(x_2+(x_1 \cdot x_3)))$$

schreiben.

In unserer allgemeinen Theorie der Rechnung wird die Ordnung der Funktionen nicht bei der Definition, sondern nur bei der Erwähnung angegeben, so daß es nicht unbedingt in allen Verkörperungen die gleiche ist.

2.) Der zweite Beitrag betrifft eine syntaktische Lösung eines der Probleme bei der Auswertung von Ausdrücken durch die Stapellogik. Es ist das Problem der Beschränkung der Länge des Stapels.

Die Definition einer beliebtigen Funktion f^t_{name} kann immer in der Form

$$f^t_{name} \equiv p_1 \rightarrow w_1,\ p_2 \rightarrow w_2,\ \ldots\ldots\ldots\ p_u \rightarrow w_u \qquad (2.1)$$

dargestellt werden. Wenn irgendeiner der w_i von (2.1) mit einem Funktionsbuchstaben beginnt, wird die Erwähnung der entsprechenden Funktion als äußerliche Erwähnung bezeichnet. Wenn in der rechten Seite von (2.1) f^t_{name} entweder nie oder ausschließlich in äußerlichen Erwähnungen vorkommt, wird die Funktion f^t_{name} als äußerlich definiert bezeichnet. Die Elemente eines Systems von Funktionen, die alle von gewissen Urfunktionen und voneinander äußerlich definiert sind, heißen äußerlich rekursive Funktionen.

In diesem Kurzvortrag sowie in [4] und [5], wo die in [3,6] beschriebenen deklarativen Algorithmen rekursiv dargestellt werden, werden ausschließlich äußerlich rekursive Funktionen gebraucht.
Man kann leicht den folgenden Satz der Reduktion zur äußerlichen Form beweisen:

Satz: "Hinreichende Bedingung dafür, daß einem System S_o von Funktionen, in welchem es allgemeine (den universellen Turing-Maschinen entsprechende) Auswertungsfunktionen gibt, ein anderes System S_1 von äußerlich rekursiven Funktionen entspricht, so daß jedes Element von S_o mindestens einem von S_1 funktionell identisch ist, ist, daß S_o ein Untersystem S_2 von äußerlich rekursiven Funktionen enthält, in welchem es eine allgemeine Auswertungsfunktion gibt".

Wenn ein Algorithmus durch Stapellogik ausgewertet wird, entspricht der Erscheinung jeder Erwähnung die Eingabe in den Stapel des Wertes der Eingabeparameter der Erwähnung und der Verbindungsdaten. Die Eingabeparameter der vorhergehenden Erwähnung werden normalerweise im Stapel gelassen. Aber, wenn die Erwähnung äußerlich ist, werden diese nicht mehr gebraucht werden, so daß man sie einfach eliminieren kann (Entladung). In diesem Fall werden die neuen Verbindungsdaten nicht eingegeben, weil die der vorigen Erwähnung an ihrer Stelle gebraucht werden sollen. Diese erweiterte Logik für den Stapel wollen wir als Entladungslogik bezeichnen. Ihre Vorteile, die im Falle der Auswertung rekursiver Funktionen besonders wichtig sind, bestehen darin, daß man die Aufbewahrung im Stapel der Eingabeparameter und Verbindungsdaten von schon verbrauchten Verkörperungen von Funktionen vermeidet.

Als Beispiel werden hier zwei Algorithmen angegeben, die beide die Fakultät einer ganzen Zahl darstellen, von denen nur der zweite aus äußerlich rekursiven Funktionen besteht.

$$f^1_{fkt1} \equiv x_1 = 0 \longrightarrow 1,\ T \longrightarrow x_1 \cdot (f_{fkt1}(x_1-1)) \qquad (2.2)$$

$$\begin{cases} \mathfrak{f}^1_{fkt2} \equiv f_{fkt3}(x_1;0,1) \\ \mathfrak{f}^3_{fkt3} \equiv x_1 = x_2 \rightarrow x_3,\ T \rightarrow f_{fkt3}(x_1,x_2+1,x_3(x_2+1)) \end{cases} \qquad (2.3)$$

Die Auswertung von (2.2) mit einer Stapellogik (mit oder ohne Entladung) bringt in den Stapel eine Reihe von Zahlen, deren Länge zu x_1 proportional ist; zudem werden überflüssige Zählungen gemacht. Im Gegensatz dazu, wird im Fall (2.3) der Stapel höchstens vier Zahlen enthalten, und keine einzige überflüssige Rechnung ist nötig.

In unserer Theorie der Rechnung werden nur äußerlich rekursive Funktionen gebraucht. Die Möglichkeit des ausschließlichen Gebrauches solcher Funktionen ist vom Satz der Reduktion zur äußerlichen Form gesichert. Es kann aber vorkommen, daß die Vorteile, die man durch die Entladungslogik vom Gebrauch der äußerlich rekursiven Funktionen bekommt, nur scheinbar sind: es kann vorkommen, daß, obwohl die Anzahl der logischen Elemente im Stapel dadurch beschränkt wird, die Länge der einzelnen Elemente sich vergrößert; im ungünstigsten Fall kann sich sogar ein solches Element wie ein Stapel betragen. Deshalb ist es notwendig, bei der Definition der Algorithmen dies zu beachten.

Der Begriff der Äußerlichkeit kann ohne weiteres zu rekursiven Funktionen erweitert werden, die durch Systeme [1] (anstatt einzeln) definiert sind.

L I T E R A T U R

[1] J.McCARTHY.

A basis for a mathematical theory of computation, Proc. Western Joint Comp. Conf. (1961), Vortrag 5.3.

[2] L.A. LOMBARDI.

System Handling of Funtional Operations.

J. Assoc. Comput. Mach., (8) 2 (1961), S. 168-185.

[3] L.A. LOMBARDI.

Mathematical Structure of Nonarithmetic Data Processing Procedures, J. Assoc. Comput. Mach., (9) 1 (1962), S. 136-159.

[4] L.A. LOMBARDI.

On the declarative Control of the flow of data by means of recursive functions

Proc. Symp. on "Symbolic Languages in Data Processing", Rome März 1962

[5] L.A. LOMBARDI.

On the Representation of Table Operating Algorithms by means of Recursive Functions (zur Veröffentlichung).

[6] L.A. LOMBARDI.

Theory of Files.

Proc. Eastern Joint Comp. Conf. (1960), Vortrag 3.3.