Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

MC-25 INFORMATICA SYMPOSIUM

BY

J.W.DE BAKKER

G.A. BLAAUW

A.J.W. DUIJVESTIJN

E.W. DIJKSTRA

P.J. VAN DER HOUWEN

G.A.M. KAMSTEEG- KEMPER

F.E.J. KRUSEMAN ARETZ

W.L.VAN DER POEL

J.P. SCHAAP-KRUSEMAN

M.V. WILKES

G. ZOUTENDIJK

MATHEMATICAL CENTRE TRACTS 37

MATHEMATISCH CENTRUM AMSTERDAM 1971

Dedicated to Prof.Dr.Ir. A. van Wijngaarden
on the occasion of his twenty-fifth anniversary
at the Mathematical Centre

FOREWORD

The Mathematisch Centrum at Amsterdam, the MC as it is called by everybody, was founded in 1946 through the foresight of a few professors of mathematics, physics and astronomy. Until then in The Netherlands the practicing of mathematics as a profession had been in the hands of a handful of university professors (an average of three per university) and a very few people in industry; everyone else interested in mathematics practiced it in his spare time. Mathematical advice was hard to come by for scientific or commercial enterprises that could not afford a mathematics department of their own. The MC was founded to fill these gaps. It would enable young gifted people to spend all their time in mathematics; and expert advice would be obtainable by those interested in the applications of mathematics. A very far sighted initiative indeed in the difficult early post war years, when our country was lacking virtually everything after five years of enemy occupation.

From a few people in 1946 the MC has grown to an organization of far over a hundred, and although each of the universities now has a mathematical institute of several dozens of people, the MC still has a very important and unique position.

Many are the ways in which the MC has influenced the mathematical life in The Netherlands. Hundreds of courses have been given to such diverse groups as high-school teachers and practical operations analysts, ALGOL 60 courses have been given by the lot, many people have been trained in modern computing methods, and so on. Also, the MC has served as a nursery for university professors: a very sizable fraction of the Dutch professorship in mathematics and computing spent a shorter or longer period of their lives in the MC. And last but not least, the MC played an important part in the dissemination of computer use and -science throughout this country (and far beyond).

In the latter field in particular the MC has had many firsts or almost-firsts, not only on a national scale but also in world-wide perspective. As early as 1952 it had its own automatic computer operational, the ARRA, working with relays and drum storage. In 1956 its first electronic computer with core memory, the ARMAC, was put into use. At about the

same time a computer (the X1) with autonomous peripheral operations on interrupt basis was conceived, which became operational in 1958, another world-wide novelty. In 1960 the MC already had an ALGOL 60 compiler, and one of the few ALGOL 60 compilers in which own dynamic arrays ever have been implemented came from the MC.

We could continue this way, but what really should be stressed is that many of the activities mentioned above are a direct consequence of another first of the MC: the MC first had Prof. Van Wijngaarden. He has been with the MC virtually from the beginning, and it is really impossible to think of one without thinking of the other.

In the limited space that is at my disposal it is impossible to relate adequately what Prof. Van Wijngaarden has done for the promotion of the knowledge of computing in The Netherlands and in the world. Many and profound are his contributions to numerical analysis and programming languages, the latter culminating in the design of ALGOL 68. But important and influential as his results are in themselves, at least as important and inspiring is the quest for elegance reflected by his work. This is particularly noteworthy in a field where microseconds count and tend to obscure all other reasoning. From his work it is apparent that in computing elegance is not incompatible with economics, but that rather the contrary is true.

Also on the organizational scene we owe a lot to Prof. Van Wijngaarden. Just to mention a few of his activities in this area: he has been director of the MC since 1961, he was one of the founding fathers of the Dutch Computer Society (NRMG), which actually was an outgrowth of the MC colloquium "Modern Computing Machines", and he was the first NRMG president. Also he stood at the cradle of IFIP, which he served ten years as general assembly member and as trustee of the council.

On behalf of the Dutch computing community it is my privilege to congratulate the MC and Prof. Van Wijngaarden with this quarter century of activities, and to thank them profoundly for their accomplishments. It has been a great privilege for us to have this extraordinary man and this extraordinary organization in our country. For many years to come we wish them continuous flourishing.

A. van der Sluis
president NRMG

PREFACE

The Mathematical Centre, founded February 11, 1946, has organised the MC-25 Informatica Symposium on the occasion of its twenty-fifth anniversary and in the honour of Prof.Dr.Ir. A. van Wijngaarden, who has been associated with the Mathematical Centre during twenty-five years, viz. since January 1, 1947 as head of its Computation Department, and since 1961 also as its director.

In the symposium, to be held in Amsterdam, January 6 and 7, 1972, nine invited speakers will present lectures. This tract contains a summary of the lecture by M.V. Wilkes and the elaborated texts of the eight other lectures.

Two speakers, M.V. Wilkes and W.L. van der Poel, were asked to give talks of an historical nature, in relation to the first and second halves, respectively, of the past twenty-five years. The other speakers deal with various topics in the field of computer science, namely G.A. Blaauw (hardware design), A.J.W. Duijvestijn et al., and F.E.J. Kruseman Aretz (software design), E.W. Dijkstra (methodology of design), J.W. de Bakker (mathematical foundations), P.J. van der Houwen (numerical mathematics) and G. Zoutendijk (mathematical programming).

It may be of interest to observe that all speakers have had some relation to the scientific activities of the Mathematical Centre, albeit in varying degrees: each of them appears at least once as an author in the list of "MC-Publications 1946-1971", issued this year by the Mathematical Centre on the occasion of its twenty-fifth anniversary.

We have had much help from Mrs. S.J. Kuipers-Hoekstra and Mr. J. Hillebrand in organizing the symposium and preparing this tract. The typing of the manuscripts was supervised by Mrs. S.M.T. Hillebrand-Snijders and performed by Miss A. Fasen and Miss O.P. de Jong. The reproduction of the texts was done by Mr. J. Schipper, Mr. J. Suiker and Mr. D. Zwarst.

Amsterdam, December 31, 1971
The organizing committee
J.W. de Bakker
Th.J. Dekker
R.P. van de Riet

CONTENTS

RECURSION, INDUCTION AND SYMBOL MANIPULATION

J.W. de Bakker *)

1. INTRODUCTION

In our monograph "Recursive procedures" [1], we have described a mathematical theory of recursive procedures. It consists of four parts:

a. An analysis of the fundamental properties of recursion.
b. A formal system based on the results of part a, with as its main proof rule an induction rule which generalizes McCarthy's recursion induction [2].
c. A large number of examples of applying the formal system.
d. A further investigation of the formal system for a restricted type of procedures.

The present paper may be considered as a companion to [1]. However, we have tried to make it self contained, so that it can also be read independently of [1].
The paper is divided into three sections:

a. *Recursion* (section 2). We characterize recursive procedures as minimal fixed points of monotonic and continuous transformations. This section is complementary to the corresponding one in [1]: A number of results which are derived there only on an intuitive basis are now treated in a precise framework. Conversely, some proofs are omitted here, but given in full in [1].
b. *Induction* (section 3). The analysis of the preceding section is applied in the development of a formal system centering around a general induction rule. Only an outline of the system is given; more details and many examples are to be found in [1].
c. *Symbol manipulation* (section 4). The induction rule is illustrated by an investigation of a number of properties of symbol manipulation. Primitive operations, such as: deletion of the first element of a sequence of

*) Mathematical Centre, Amsterdam

symbols, addition of a given symbol on the left, and a few others, are introduced. Other operations are then expressed in terms of the primitive ones, and a selection of their properties is proved. Finally, from the example a general rule is abstracted, which relates recursive definitions to the inductive properties of the data structure one is dealing with in a specific area of application.

The main concepts and results of sections 2 and 3 are based upon unpublished work by Scott [3,4], partly describing joint work with the present author.

An extensive list of references to related work on mathematical properties of recursive procedures is contained in [1].

2. RECURSION

In the framework of a simple programming language, a definition is given of the semantics of recursive procedures, and from this definition their mathematical characterization as *minimal fixed points* of *monotonic* and *continuous* transformations is derived.

2.1. NOTATION

Let X be any set, and let $F: X \to X$ be the class of all *partial* functions from X to X. Let $f_1, f_2 \in F$. We define a partial ordering "$\subseteq$" between f_1 and f_2 as follows:

$$f_1 \subseteq f_2 \text{ iff for all } x \in X, \text{ if } f_1(x) = y \text{ then } f_2(x) = y.$$

In other words, $f_1 \subseteq f_2$ iff the graph of f_1 is a subset of the graph of f_2. That "$\subseteq$" is indeed a partial ordering is clear. In particular, we have $f_1 = f_2$ iff both $f_1 \subseteq f_2$ and $f_2 \subseteq f_1$.

With respect to the partial ordering "$\subseteq$" we have the usual notions of greatest lower bound (g.l.b.) and least upper bound (l.u.b.). For any subset F_0 of F, we denote its g.l.b. by $\cap\{f: f \in F_0\}$ and its l.u.b. by $\cup\{f: f \in F_0\}$. Note that the l.u.b. does not necessarily exist: If f_1, f_2 are such that for some x, $f_1(x) = y$, $f_2(x) = z$, and $y \neq z$, then there exists no function f such that $f_1 \subseteq f$ and $f_2 \subseteq f$. Note also that each

chain $f_0 \subseteq f_1 \subseteq \ldots \subseteq f_i \subseteq \ldots$ has a l.u.b. $\bigcup_{i=0}^{\infty} f_i$.

2.2. PROGRAM SCHEMES AND PROGRAMS

First we introduce a class of structures called *program schemes*. We then show how to *interpret* a program scheme in order to obtain a *program*. The meaning of a program is then defined in terms of *computation sequences*.

In the composition of program schemes, the following classes of symbols are used:

a. $\mathcal{F}$: The class of *function* symbols, with elements
   $F, F_1, F_2, \ldots$.
b. $\mathcal{B}$: The class of *predicate* symbols, with elements
   $p, p_1, q, r, \ldots$.
c. $\mathcal{P}$: The class of *procedure* symbols, with elements
   $P, P_1, P_2, \ldots$.
d. $\mathcal{C}$: The class of *constant* symbols, with the two elements
   E, Ω.

Anticipating the rules for interpreting program schemes, to be given below, we may already indicate the intended correspondence between these classes and their counterparts in an ALGOL-like language: The elements of $\mathcal{F}$ correspond to certain basic statements, the elements of $\mathcal{B}$ to boolean expressions, and the elements of $\mathcal{P}$ to procedures. Moreover, E corresponds to the dummy statement, and Ω to the undefined statement (L: goto L, say).

Next, we give the definition of the class of statement schemes $\mathcal{S}$.

a. Each element of $\mathcal{F}$, $\mathcal{P}$ or $\mathcal{C}$ is a statement scheme.
b. If S_1, S_2 are statement schemes, and $p \in \mathcal{B}$, then $S_1;S_2$ and $(p \rightarrow S_1, S_2)$ are statement schemes.

The notation $(p \rightarrow S_1, S_2)$ corresponds to the ALGOL notation <u>if</u> p <u>then</u> S_1 <u>else</u> S_2.

$S, S_1, \ldots, T, T_1, \ldots$ will stand for arbitrary elements of $\mathcal{S}$.

When we want to indicate that T possibly contains one or more occurrences of S, we write $T = T(S)$. The result of substituting S_1 for all occurrences of S in T is then written as $T(S_1)$.

Besides the statement schemes we have *declaration schemes*: A declaration scheme is a pair (P,T(P)), with $P \in \mathcal{P}$, and T(P) a statement scheme. Such a pair will usually be denoted in the sequel by the more familiar notation procedure P;T(P).

A *program scheme* is again a pair (D,T), where D is a finite set of declaration schemes, and T is a statement scheme.
Examples of program schemes are:

(procedure P; (p → F;P,E)
P)

(procedure P_1; (p → F_1;P_1,E)
procedure P_2; (q → F_2;P_2,(r → F_3;P_1,F_4))
P_1; (r → P_2,Ω);F_5)

For the reader who is more accustomed to a functional notation, we give a definition scheme corresponding to the procedures P_1 and P_2 of the second example. Using the notation "⟸" for "is recursively defined by", we have

$$P_1(x) \Longleftarrow (p(x) \rightarrow P_1(F_1(x)),x)$$
$$P_2(x) \Longleftarrow (q(x) \rightarrow P_2(F_2(x)),(r(x) \rightarrow P_1(F_3(x)),F_4(x))).$$

Program schemes cannot be "executed" as such. Firstly, a rule has to be given to attribute a meaning to the function symbols, predicate symbols, and constant symbols occurring in it. Secondly, we then must define how to provide a meaning for the various constructs in the program scheme, in terms of the meaning of their constituent symbols.

An *interpretation* I of a program scheme (D,T) consists of a pair (X,h), where

a. X is any domain (i.e., an arbitrary set).
b. h maps symbols to partial functions as follows:
   b1. To each function symbol $F \in \mathcal{F}$ occurring in (D,T) (i.e., either in T, or in one of the elements of D) is assigned a partial function F^h: X → X.
   b2: To each predicate symbol $p \in \mathcal{B}$ occurring in (D,T) is assigned a partial function p^h: X → {0,1}.
   b3: To the constant symbol $E \in \mathcal{C}$ is assigned the identity function

E^h: $X \to X$ (for all $x \in X$, $E^h(x) = x$) and to $\Omega \in C$ is assigned the nowhere defined function Ω^h: $X \to X$ (for no $x \in X$ there exists y such that $\Omega^h(x) = y$).

Given an interpretation $I = (X,h)$ of the symbols in a program scheme (D,T), we next define how to obtain a partial function $(D,T)^I$: $X \to X$. For this definition, we need the notion of a "*computation sequence* with respect to D and I":
Let $x_i \in X$, $1 \leq i \leq n+1$, and let S_i, $1 \leq i \leq n$, be statement schemes. A computation sequence is a finite sequence

$$x_1 S_1 x_2 S_2 \cdots x_n S_n x_{n+1}$$

such that, for each i, $1 \leq i \leq n$,

a. If $S_i = E$, then $i = n$ and $x_{n+1} = x_n$ $(= E^h(x_n))$

b. If $S_i = F;S$ then $\begin{cases} x_{i+1} = F^h(x_i) \\ S_{i+1} = S \end{cases}$

c. If $S_i = (p \to S',S");S$ then $\begin{cases} x_{i+1} = x_i \\ S_{i+1} = S';S, \text{ if } p^h(x_i) = 1 \\ S_{i+1} = S";S, \text{ if } p^h(x_i) = 0 \end{cases}$

d. If $S_i = P;S$ then $\begin{cases} x_{i+1} = x_i \\ S_{i+1} = T(P);S \text{ where } (P,T(P)) \in D. \end{cases}$

Note that

1. Each computation sequence has E as its last statement scheme. This is a convention which simplifies the definition.
2. There is no S_i such that $S_i = \Omega$ or $S_i = \Omega;S$.
3. Clause c corresponds to the usual meaning of conditionals, with 1(0) corresponding to <u>true</u> (<u>false</u>).
4. Clause d gives the usual copy rule for procedures: In order to elaborate a procedure call P, replace P by its body T(P), and elaborate the result.

The notion of computation sequence is used as follows: Given a program scheme (D,T) and an interpretation $I = (X,h)$ of the symbols in D and T, we define the partial function $(D,T)^I$: $X \to X$ by:

For each $x \in X$,

$(D,T)^I(x) = y$ if there exists a computation sequence with respect to D and I

$x_1 S_1 x_2 S_2 \cdots x_n S_n x_{n+1}$

with $x_1 = x$, $S_1 = T$, and $x_{n+1} = y$;

is undefined, otherwise.

In the sequel, we shall omit explicit mentioning of the set of declarations D, if no confusion can arise. We then write simply T^I instead of $(D,T)^I$.

Some simple consequences of the definition are:

a. $(T_1;T_2)^I(x) = y$ iff there exists $z \in X$ such that $T_1^I(x) = z$ and $T_2^I(z) = y$. Hence, the function $(T_1;T_2)^I$ is the *composition* of the two functions T_1^I and T_2^I.

b. $(p \rightarrow T_1, T_2)^I(x) = y$ iff either $p^h(x) = 1$, and $T_1^I(x) = y$, or $p^h(x) = 0$ and $T_2^I(x) = y$.

The definitions of section 2.1 apply to the partial functions T^I. Thus, we may have, for given I, that $T_1^I \subseteq T_2^I$, $T_1^I = T_2^I$, etc. If for all I, $T_1^I \subseteq T_2^I$, we write $T_1 \subseteq T_2$. Similarly, equivalence of two program schemes T_1 and T_2 under all interpretations is denoted by $T_1 = T_2$.

The notions introduced so far will be used in an analysis of recursive procedures in the next section.

2.3. RECURSIVE PROCEDURES AS MINIMAL FIXED POINTS

The first result to be noted about procedures is the fixed point property:

(1) $P = T(P)$.

The proof is an immediate consequence of the "copy rule". We have to show that, for all I, $P^I = T(P)^I$.

a. $P^I \subseteq T(P)^I$.

Suppose $P^I(x) = y$. Then there exists a computation sequence with respect to I

$x_1 S_1 x_2 S_2 \cdots x_n S_n x_{n+1}$
such that $x_1 = x_2 = x$, $S_1 = P$, $S_2 = T(P)$ and $x_{n+1} = y$.
Then, clearly,
$x_2 S_2 x_3 \cdots x_n S_n x_{n+1}$
is a computation sequence with $x_2 = x$, $S_2 = T(P)$, and $x_{n+1} = y$. Therefore, $T(P)^I(x) = y$, and we have shown that, for all x, if $P^I(x) = y$ then $T(P)^I(x) = y$, i.e., we have shown that $P^I \subseteq T(P)^I$.

b. $T(P)^I \subseteq P^I$.

The proof of this is similar to case a and omitted.

In general, (1) does not determine the procedure uniquely. E.g., consider the procedure P_1 declared by procedure P_1; $(p \rightarrow P_1, E)$. Each S of the form $(p \rightarrow F, E)$, for arbitrary F, satisfies $S = (p \rightarrow S, E)$, as can easily be seen from the equivalence $(p \rightarrow (p \rightarrow F, E), E) = (p \rightarrow F, E)$. This fact can also be expressed by saying that the transformation $T(X) = (p \rightarrow X, E)$ has an infinity of fixed points, i.e., for all S of the form $(p \rightarrow F, E)$ we have $T(S) = S$.

As a second example, consider the factorial definition

(2) $$f(x) \Leftarrow (x=0 \rightarrow 1,\ x*f(x-1)).$$

Suppose we consider this definition for the domain of all integers. Let g(x) and h(x) be the following partial functions:

$$g(x) = x!, \quad \text{for } x \geq 0$$
$$\text{is undefined, for } x < 0$$
$$h(x) = x!, \quad \text{for } x \geq 0$$
$$= 0\ , \quad \text{for } x < 0.$$

Clearly, both g and h satisfy the functional equation for f

$$f(x) = (x=0 \rightarrow 1,\ x*f(x-1))$$

i.e., both g and h are fixed points of the transformation T(f), as given by the right hand side of (2).

These two examples illustrate the need for a further analysis of the semantics of recursive procedures, in order to allow us to determine which, among all possible fixed points, is the one we need. (Clearly in the first example we want $P = (p \rightarrow \Omega, E)$, and in the second example we want g(x) and

not h(x).)
The first step in this analysis is to observe the following monotonicity property of the schemes T:
For all interpretation I, if $S_1^I \subseteq S_2^I$, then $T(S_1)^I \subseteq T(S_2)^I$, i.e.,

(3) If $S_1 \subseteq S_2$ then $T(S_1) \subseteq T(S_2)$,

Verification of (3) amounts to a straighforward application of the notion of computation sequence, and is omitted here.

The second step needs some more argument. Suppose that, for some I, we know that a procedure P is defined for given x: $P^I(x) = y$. Thus, there is a computation sequence with respect to I

(4) $x_1 P x_2 T(P) \ldots x_n E x_{n+1}$

with $x_1 = x_2 = x$, $x_{n+1} = y$.
Clearly, in recursive situations, the computation sequence (4) may contain other occurrences of P. If the number of these other occurrences is k, (4) has the form

(5) $x_1 P x_2 T(P) \ldots x_{i_1} P;S' x_{i_1+1} T(P);S' \ldots$

$\ldots$

$x_{i_k} P;S'' x_{i_k+1} T(P);S'' \ldots x_n E x_{n+1}$.

We then have that

$x_1 P x_2 T(P) \ldots x_{i_1} P;S' x_{i_1+1} T(P)$

$\ldots$

$x_{i_k} P;S'' x_{i_k+1} T(\Omega);S'' \ldots x_n E x_{n+1}$

is also a computation sequence: The substitution of Ω for P in the final T(P) makes no difference, since P did not occur in the "tail" of (5) (that part of (5) between $x_{i_k} P;S'' x_{i_k+1} T(P);S''$ and $x_n E x_{n+1}$) anyway.

Next consider the segment

$$\ldots\ x_{i_{k-1}}P;S'''x_{i_{k-1}+1}T(P);S'''\ \ldots\ x_{i_k}P;S''x_{i_k+1}T(\Omega);S''\ \ldots$$

of the computation sequence (5). Clearly, we may replace this by

$$\ldots\ x_{i_{k-1}}P;S'''x_{i_{k-1}+1}T(T(\Omega));S'''\ \ldots\ x_{i_k}P;S''x_{i_k+1}T(\Omega);S''\ \ldots$$

Repeating the argument, we finally obtain that

$$x_1Px_2T^{k+1}(\Omega)\ \ldots\ x_nEx_{n+1}$$

is also a computation sequence. In other words, we have derived that, if $P^I(x) = y$, then there exists $k > 0$, depending on x, such that $T^{k+1}(\Omega)^I(x) = y$. This result amounts to the following familiar property of recursive procedures: If a recursive procedure terminates for a given argument, it terminates after a finite number of "inner calls" of the procedure. At the innermost level, P is not needed again, so we may replace it there by whatever statement we wish, without influencing the further execution. The reason for the choice of Ω for this statement will appear presently.

Consider the sequence

$$\Omega^I, T(\Omega)^I, T^2(\Omega)^I, \ldots\ . \tag{6}$$

By the monotonicity property (3), we see that we have

$$\Omega^I \subseteq T(\Omega)^I \subseteq T^2(\Omega)^I \subseteq \ldots\ .$$

Thus, (6) has a l.u.b. $\bigcup_{i=0}^{\infty} T^i(\Omega)^I$, and we have shown that, for each I,

$$P^I \subseteq \bigcup_{i=0}^{\infty} T^i(\Omega)^I$$

i.e., we have

$$P \subseteq \bigcup_{i=0}^{\infty} T^i(\Omega). \tag{7}$$

Next we show that

$$P \supseteq \bigcup_{i=0}^{\infty} T^i(\Omega).$$

We have, using (1) and monotonicity

$$\begin{aligned} \Omega &\subseteq P \\ T(\Omega) &\subseteq T(P) = P \\ &\vdots \\ T^i(\Omega) &\subseteq P \\ &\vdots \end{aligned}$$

Hence,

$$\bigcup_{i=0}^{\infty} T^i(\Omega) \subseteq P \tag{8}$$

follows. Thus, we have, from (7) and (8), that

$$\bigcup_{i=0}^{\infty} T^i(\Omega) = P. \tag{9}$$

This means that we have shown that P is the *minimal fixed point* of T: Let Q satisfy Q = T(Q). Then we show as above that $\bigcup_{i=0}^{\infty} T^i(\Omega) \subseteq Q$. In an alternative formulation, we have proved that

$$P = \bigcap\{X: X = T(X)\}.$$

Result (9) can be expressed by saying that we have obtained P by successive approximation, starting with the undefined function. E.g., if one wants to evaluate the factorial function f, one starts with the completely undefined function. Then one establishes that f(0) = 1; using this, one extends f and finds that f(1) = 1, next f(2) = 2, f(3) = 6 etc. are successively found.

Now that we have obtained the minimal fixed point property, consider again the example of the procedure P_1, declared by <u>procedure</u> P_1; ($p \rightarrow P_1$,E). We construct, for T(X) = ($p \rightarrow$X,E), the sequence of approximations Ω, $T(\Omega) = (p \rightarrow \Omega,E)$, $T^2(\Omega) = (p \rightarrow (p \rightarrow \Omega,E),E) = (p \rightarrow \Omega,E),\ldots,$ $T^i(\Omega) = (p \rightarrow \Omega,E),\ldots$. Thus, for this T, $P_1 = \bigcup_{i=0}^{\infty} T^i(\Omega) = (p \rightarrow \Omega,E)$.

We conclude this section with the introduction of another property of the transformations T, which generalizes their monotonicity. Assume that we have a chain

$$X_0 \subseteq X_1 \subseteq \ldots \subseteq X_i \subseteq \ldots .$$

Then

$$\bigcup_{i=0}^{\infty} T(X_i) = T(\bigcup_{i=0}^{\infty} X_i). \tag{10}$$

(10) asserts that the transformations T are *continuous*. That $\bigcup_{i=0}^{\infty} T(X_i) \subseteq T(\bigcup_{i=0}^{\infty} X_i)$ is a direct result from monotonicity: For each i, $T(X_i) \subseteq T(\bigcup_{i=0}^{\infty} X_i)$; hence, $\bigcup_{i=0}^{\infty} T(X_i) \subseteq T(\bigcup_{i=0}^{\infty} X_i)$. The proof of the reverse inclusion, which is omitted here, proceeds by an inductive argument on the complexity of the T concerned. E.g., if T(X) has the simple form T(X) = F;X, we have to verify whether $F; \bigcup_{i=0}^{\infty} X_i = \bigcup_{i=0}^{\infty} F;X_i$. That this is indeed the case follows easily from the definitions.
A consequence of (10) is the fact that for each S,T,

$$\bigcup_{i=0}^{\infty} S(T^i(\Omega)) = S(\bigcup_{i=0}^{\infty} T^i(\Omega)). \tag{11}$$

This fact will be used in the justification of our generalization of recursion induction, to be introduced in section 3.

2.4. SYSTEMS OF RECURSIVE PROCEDURES

For systems of mutually dependent recursive procedures, such as

$$\underline{\text{procedure}}\ P_1;\ T_1(P_1,P_2)$$
$$\underline{\text{procedure}}\ P_2;\ T_2(P_1,P_2)$$

the analysis of the preceding section has not yet been refined enough. From (9) we see that we have

$$\begin{aligned} P_1 &= \Omega \cup T_1(\Omega,P_2) \cup T_1(T_1(\Omega,P_2),P_2) \cup \ldots \\ P_2 &= \Omega \cup T_2(P_1,\Omega) \cup T_2(P_1,T_2(P_1,\Omega)) \cup \ldots . \end{aligned} \tag{12}$$

This is not yet sufficient. E.g., on the basis of these formulae we cannot show the following: If $Q_1 = T_1(Q_1,Q_2)$, $Q_2 = T_2(Q_1,Q_2)$, then $P_1 \subseteq Q_1$, $P_2 \subseteq Q_2$.
What we need is a pair of formulae like (12) which do not, however, contain occurrences of P_1 and P_2 in their right hand sides. This can be obtained as follows:

First we introduce a new notation:

$$(T_1,T_2)^0(X,Y) = (X,Y), \quad \text{and for } i > 0,$$

$$(T_1,T_2)^i(X,Y) = (T_1,T_2)^{i-1}(T_1(X,Y),T_2(X,Y)).$$

Using an argument on computation sequences similar to the one we used in order to derive $P = \bigcup_{i=0}^{\infty} T^i(\Omega)$, we can show that

$$(13) \qquad (P_1,P_2) = \bigcup_{i=0}^{\infty} (T_1,T_2)^i(\Omega,\Omega)$$

i.e., we have

$$P_1 = \Omega \cup T_1(\Omega,\Omega) \cup T_1(T_1(\Omega,\Omega),T_2(\Omega,\Omega)) \cup \ldots$$

$$P_2 = \Omega \cup T_2(\Omega,\Omega) \cup T_2(T_1(\Omega,\Omega),T_2(\Omega,\Omega)) \cup \ldots .$$

From (13) the desired minimal fixed point property of the pair (P_1,P_2) follows:

$$(P_1,P_2) = \cap\{(X,Y): X = T_1(X,Y),\ Y = T_2(X,Y)\}.$$

Thus, we may say that the pair (P_1,P_2) is obtained as a *simultaneous* minimal fixed point of the pair of transformations (T_1,T_2).

By way of introduction to the next step, compare the following two ALGOL-like programs:

```
P1 is
begin procedure P1; T1(P1,P2);
      procedure P2; T2(P1,P2);
      P1
end
```

and P_2 is

```
begin procedure P1;
      begin procedure P2; T2(P1,P2);
            T1(P1,P2)
      end;
      P1
end
```

We know that $P_1 = P_2$. In our terminology, this can be expressed by:
If P_1 is defined by (13), and P_1' by

$$P_1' = \cap\{X: X=T_1(X,\cap\{Y: Y=T_2(X,Y)\})\} \tag{14}$$

then $P_1 = P_1'$. For the proof of this, we refer to [1]. (14) can be used to obtain an alternative for the formula (13). We use the following notation:

$$T(X)^0(\Omega) = \Omega$$

$$T(X)^i(\Omega) = T(X,T(X)^{i-1}(\Omega)).$$

Then from the results of the preceding section we have

$$\cap\{Y: Y=T_2(X,Y)\} = \bigcup_{i=0}^{\infty} T_2(X)^i(\Omega).$$

Let us write $T''(X) = \bigcup_{i=0}^{\infty} T_2(X)^i(\Omega)$. Then

$$P_1 = \cap\{X: X=T_1(X,T''(X))\}.$$

Let us write $T'(X) = T_1(X,T''(X))$. Then we have for P_1:

$$P_1 = \bigcup_{i=0}^{\infty} (T')^i(\Omega). \tag{15}$$

Thus, the simultaneous formula (13) can be replaced by two formulae: (15) for P_1 and its analogue for P_2.
Note that the introduction of T' and T'' has extended our program scheme language. Thus for this extended language we have to verify anew whether the monotonicity and continuity properties (3) and (10) still hold. For

the proof that this is indeed the case we refer again to [1].

3. INDUCTION

The insight obtained into the mathematical properties of recursive procedures is applied in the development of a formal system with axioms and rules of inference. The main rule of inference is the so-called μ-*induction rule*, which generalizes McCarthy's recursion induction [2].

3.1. THE μ-CALCULUS

In section 2 we have seen that a procedure P, declared by
procedure P; T(P), is the minimal fixed point of the transformation T, i.e.,

$$P = \bigcap\{X: X=T(X)\}.$$

We now introduce the following notation:

$$\bigcap\{X: X=T(X)\} = \mu X[T(X)]$$

i.e., μX[T(X)] is read as: the minimal fixed point of the transformation T. The main advantage of this notation is that it brings out that the μ-operator is a *variable-binding* operator, which implies the usual consequences for the distinction between free and bound variables. E.g., the rule of rewriting of bound variables is applicable, i.e., μX[T(X)] = μY[T(Y)], provided that Y does not occur free in T.
Also, for a system of two procedures procedure P_1; $T_1(P_1,P_2)$; procedure P_2; $T_2(P_1,P_2)$, we saw in section 2.4 that

$$P_1 = \bigcap\{X: X=T_1(X, \bigcap\{Y: Y=T_2(X,Y)\})\}.$$

Thus, using the μ-notation, we can write for P_1

$$P_1 = \mu X[T_1(X,\mu Y[T_2(X,Y)])]$$

and similarly for P_2.
Note that with this new notation we no longer need declarations. E.g., the program scheme

($\underline{\text{procedure}}$ P_1; $T_1(P_1,P_2)$
$\underline{\text{procedure}}$ P_2; $T_2(P_1,P_2)$
F_1; P_1; F_2; P_2; F_3)

is now written as

$$F_1;\ \mu X[T_1(X,\mu Y[T_2(X,Y)])];\ F_2;\ \mu X[T_2(\mu Y[T_1(Y,X)],X)].$$

Thus, the definition of *terms* in the formal language (corresponding to the program schemes of the previous section) can simply be given as:

a. Each function symbol or constant symbol is a term.
b. If S_1,S_2 are terms and p is a predicate symbol, then $S_1;S_2$ and $(p\rightarrow S_1,S_2)$ are terms.
c. If $T(X)$ is a term, then $\mu X[T(X)]$ is a term, where X is any function symbol.

In the sequel we shall usually omit the ";" symbol between terms.

Assertions about terms in this language are now introduced as follows:

a. An *atomic formula* is an expression of the form $S_1 \subseteq S_2$ or $S_1 = S_2$, where S_1,S_2 are terms.
b. A *formula* is a list of zero or more atomic formulae.
c. An *assertion* is an expression of the form $\Phi \vdash \psi$, with Φ,ψ formulae.

Examples of assertions are

$$X \subseteq Y,\ Y \subseteq Z \vdash X \subseteq Z$$

$$X \subseteq Y \vdash T(X) \subseteq T(Y)$$

$$\vdash \mu X[X] = \Omega$$

$$\vdash \mu X[(p\rightarrow FX,E)] = (p\rightarrow F\mu X[(p\rightarrow FX,E)],E).$$

A verbal explanation of these assertions may help in understanding the formalism: The first assertion states the transitivity of "$\subseteq$", the second states monotonicity of the terms in their free variables. The third assertion tells us that the procedure P, declared by $\underline{\text{procedure}}$ P; P terminates nowhere. As to the fourth assertion, it will be seen that the meaning of

$\mu X[(p \to EX, E)]$ corresponds to that of <u>while</u> p <u>do</u> F. Hence, this assertion can be read as: <u>while</u> p <u>do</u> F = $(p \to F;$ <u>while</u> p <u>do</u> F, E). Note that this is nothing but the fixed point property of the procedure $\mu X[(p \to FX, E)]$.

3.2. THE μ-INDUCTION RULE

We shall not give here the full system of axioms and rules for the μ-calculus. For more details, we refer to [1].
Briefly, we have

a. Axioms for composition
   Examples: $\vdash EX = X$, $\vdash \Omega X = \Omega$.

b. Axioms for inclusion
   Examples: $X \subseteq Y \vdash T(X) \subseteq T(Y)$, $\vdash \Omega \subseteq X$.

c. Axioms for conditionals
   Examples: $\vdash (p \to (p \to X, Y), Z) = (p \to X, Z)$, $\vdash (p \to X, Y)Z = (p \to XZ, YZ)$.

d. An axiom for the μ-operator
   $\vdash T(\mu X[T(X)]) \subseteq \mu X[T(X)]$.

e. A rule of inference for the μ-operator, which is called the μ-*induction rule*

$$\frac{\Phi \vdash \psi(\Omega) \qquad \Phi, \psi(X) \vdash \psi(T(X))}{\Phi \vdash \psi(\mu X[T(X)])}$$

   which holds provided that X does not occur free in Φ.

This last rule is the central one of the μ-calculus. That it has indeed the form of an induction rule may become more transparent if we write it as follows: Let α: $\Phi \vdash \psi$ be the assertion we want to prove. Then the rule tells us that if we have verified that

a. $\alpha(\Omega)$ holds,
b. if $\alpha(X)$ holds, then $\alpha(T(X))$ holds,

we may then infer that $\alpha(\mu X[T(X)])$ holds.

The justification of this rule relies on the continuity of our trans-

formations. E.g., if we want to show that, for given S_1, S_2,

$\vdash S_1(\mu X[T(X)]) \subseteq S_2$, we show

a. $\vdash S_1(\Omega) \subseteq S_2$

b. $S_1(X) \subseteq S_2 \vdash S_1(T(X)) \subseteq S_2$.

Starting from a, and repeatedly applying b, we obtain $S_1(\Omega) \subseteq S_2$, $S_1(T(\Omega)) \subseteq S_2$, ..., $S_1(T^i(\Omega)) \subseteq S_2$, ... Hence $\bigcup_{i=0}^{\infty} S_1(T^i(\Omega)) \subseteq S_2$. Then, using continuity: $\bigcup_{i=0}^{\infty} S_1(T^i(\Omega)) = S_1(\bigcup_{i=0}^{\infty} T^i(\Omega))$ (see (11)), and the fact that $\mu X[T(X)] = \bigcup_{i=0}^{\infty} T^i(\Omega)$, we see that we can infer that $S_1(\mu X[T(X)]) \subseteq S_2$, which is the desired result. The argument of this example can easily be extended to general assertions.

As a first application of the μ-induction rule, we discuss the procedure P_1 of section 2.3, declared by procedure P_1; $(p \to P_1, E)$, for which we derived that $P_1 = (p \to \Omega, E)$. In the μ-calculus, this can be formulated as

$$\vdash \mu X[(p \to X, E)] = (p \to \Omega, E)$$

which can be shown as follows:

a. $\vdash \mu X[(p \to X, E)] \subseteq (p \to \Omega, E)$.

Let $\Phi(X)$ be $X \subseteq (p \to \Omega, E)$ and let $T(X) = (p \to X, E)$. Then we have:

$\Phi(\Omega)$: $\Omega \subseteq (p \to \Omega, E)$

$\Phi(X)$: $X \subseteq (p \to \Omega, E)$

$\Phi(T(X))$: $(p \to X, E) \subseteq (p \to \Omega, E)$

$\Phi(\mu X[T(X)])$: $\mu X[(p \to X, E)] \subseteq (p \to \Omega, E)$

According to the μ-induction rule, in order to prove $\vdash \Phi(\mu X[T(X)])$ we must verify $\vdash \Phi(\Omega)$ and $\Phi(X) \vdash \Phi(T(X))$, i.e., we must establish that

(i) $\vdash \Omega \subseteq (p \to \Omega, E)$, which is clear, and

(ii) $X \subseteq (p \to \Omega, E) \vdash (p \to X, E) \subseteq (p \to \Omega, E)$

which follows from

$X \subseteq (p \to \Omega, E) \vdash (p \to X, E) \subseteq (p \to (p \to \Omega, E), E) = (p \to \Omega, E)$

b. $\vdash (p \to \Omega, E) \subseteq \mu X[(p \to X, E)]$.

By the μ-axiom, we have

$\vdash (p \to \mu X[(p \to X, E)], E) \subseteq \mu X[(p \to X, E)]$.

Since, by monotonicity,

$\vdash (p \to \Omega, E) \subseteq (p \to \mu X[(p \to X, E)], E)$

the desired result follows from the transitivity of "$\subseteq$".

As a second application, we show how to use the μ-induction rule in order to justify McCarthy's recursion induction. This can, in our notation, be read as: If P_1 is declared by procedure P_1; $T(P_1)$, and if P_2 and P_3 satisfy $P_2 = T(P_2)$, $P_3 = T(P_3)$, then $P_2 = P_3$ for all arguments for which P_1 is defined. It will be sufficient to show that $P_1 \subseteq P_2$ and $P_1 \subseteq P_3$, since, then, for all arguments x for which P_1 is defined, we have $P_1(x) = P_2(x) = P_3(x)$. We show that $P_1 \subseteq P_2$. In the μ-calculus, this reads

$$Y = T(Y) \vdash \mu X[T(X)] \subseteq Y.$$

We have to verify the two steps

a. $$Y \subseteq T(Y) \vdash \Omega \subseteq Y$$

which is clear, and

b. $$Y = T(Y),\ X \subseteq Y \vdash T(X) \subseteq Y$$

which easily follows from monotonicity.

It is also simple to prove with the μ-induction rule that

$$\vdash \mu X[T(X)] \subseteq T(\mu X[T(X)]).$$

Together with the μ-axiom, this yields the fixed point property of procedures, i.e.,

$$\vdash \mu X[T(X)] = T(\mu X[T(X)]).$$

Many other examples of proofs based on the μ-induction rule are given in

[1]. In the next section, we discuss yet another example, in which we apply the μ-calculus to an investigation of symbol manipulation.

4. SYMBOL MANIPULATION

4.1. AN APPLIED μ-CALCULUS FOR SYMBOL MANIPULATION

We consider a finite set of symbols $\Sigma = \{x,y,\ldots,z\}$. Arbitrary elements of Σ are denoted by $\xi,\eta,\ldots$. Σ^* denotes the set of all finite sequences of elements of Σ, including the empty sequence, which is denoted by ε.

The following primitive operations and predicates on elements of Σ^* are introduced:

a. For each $\xi \in \Sigma$, T_ξ is the operation which puts the symbol ξ in front of its argument, i.e.,
$T_\xi(\sigma) = \xi\sigma$.

b. D deletes the first symbol of its argument, if this is non empty; otherwise, D is undefined, i.e.,
$D(\xi\sigma) = \sigma$,
$D(\varepsilon)$ is undefined.

c. A_ε sets its argument to the empty sequence, i.e.,
$A_\varepsilon(\sigma) = \varepsilon$.

d. For each $\xi \in \Sigma$, p_ξ is a predicate which is true if its argument starts with the symbol ξ, otherwise p_ξ is false, i.e.,
$p_\xi(\sigma) = 1$, if there exists σ' such that $\sigma = \xi\sigma'$,
$= 0$, otherwise.

We shall indicate how a number of other operations can be expressed in terms of these primitives and we shall prove a selection of their properties.

We need an "applied" μ-calculus, i.e., to the general formalism of section 3 we add a number of special axioms, characterizing our primitives. Before stating these axioms, we first introduce a new notation. We shall often be dealing with terms of the following structure.

$$(16) \qquad (p_x \to 0_x, (p_y \to 0_y, \ldots, (p_z \to 0_z, 0')\ldots))$$

where $0_x,0_y,\ldots,0_z$ stand for arbitrary terms which may contain symbols indexed by x,y,...,z. As an instance of (16) we have

$$(p_x \to DT_x,(p_y \to DT_y,\ldots,(p_z \to DT_z,E)\ldots)).$$

For terms of the form (16) we use the abbreviated notation

$$(p_\xi \to 0^\xi,0')$$

or, also

$$(p_\eta \to 0^\eta,0')$$

etc. I.e., each pair of identical lower-upper indices used in this way indicates a short hand for the full term of the form (16). As a further example of this notation, consider the following equivalences:

$$\vdash (p_x \to 0_x,(p_y \to 0_y,\ldots,(p_z \to 0_z,(p_\eta \to 0',0''))\ldots)) =$$

$$(p_x \to 0_x,(p_y \to 0_y,\ldots,(p_z \to 0_z,0'')\ldots))$$

which holds for each $\eta \in \Sigma$, and

$$\vdash (p_x \to 0_x,(p_y \to 0_y,\ldots,(p_z \to 0_z,(p_x \to 0'_x,(p_y \to 0'_y,\ldots,(p_z \to 0'_z,0'')\ldots)))\ldots)) =$$

$$(p_x \to 0_x,(p_y \to 0_y,\ldots,(p_z \to 0_z,0'')\ldots)).$$

The proofs of these equivalence follow by some manipulations with the conditional axioms, and are omitted here. Using the new notation, they are expressed in lemma 1:

LEMMA 1

$$\vdash (p_\xi \to 0^\xi,(p_\eta \to 0',0'')) = (p_\xi \to 0^\xi,0'')$$

$$\vdash (p_\xi \to 0^\xi,(p_\eta \to 0'^\eta,0'')) = (p_\xi \to 0^\xi,0'').$$

We now give five axioms characterizing our primitives:

A_1: $T_\xi D = E$, for each $\xi \in \Sigma$

A_2: $A_\varepsilon D = \Omega$,

A_3: $T_\xi A_\varepsilon = A_\varepsilon$, for each $\xi \in \Sigma$

A_4: $T_\xi(p_\eta \to X,Y) = T_\xi X$, for each $\xi,\eta \in \Sigma$ with $\xi = \eta$

$= T_\xi Y$, for each $\xi,\eta \in \Sigma$ with $\xi \neq \eta$

A_5: $\mu X[(p_\xi \to (DXT)^\xi, A_\varepsilon)] = E$,

Cf. the axioms for integers in [1], which are in turn based on a set of axioms in unpublished work by Scott.

Axioms A_1 to A_4 are readily understood by considering the associated equivalences over Σ . E.g., for A_1 we have, for each $\xi \in \Sigma$: $(T_\xi D)(\sigma) = D(T_\xi(\sigma)) = D(\xi\sigma) = \sigma = E(\sigma)$.

Axiom A_5 may take some more trouble. It may be helpful to consider its equivalent formulation in a functional notation: Let the function f be recursively defined by

$$f(\sigma) \Leftarrow (p_x(\sigma) \to T_x(f(D(\sigma))), (p_y(\sigma) \to T_y(f(D(\sigma))), \ldots$$
$$(p_z(\sigma) \to T_z(f(D(\sigma))), A_\varepsilon(\sigma)) \ldots)).$$

Then, for each σ, we have

$$f(\sigma) = \sigma.$$

This can be shown by induction on the length of the argument sequence σ: If $\sigma = \varepsilon$, then none of $p_x(\sigma), p_y(\sigma), \ldots, p_z(\sigma)$ are true. Hence, for this σ, $f(\sigma) = A_\varepsilon(\sigma) = \varepsilon$.
Let the assertion be true for all σ_1 of length $< n$. Let σ be of length n. Then there exists σ_1 and ξ such that $\sigma = \xi\sigma_1$. Clearly, for this σ, $p_\xi(\sigma) = 1$ and, therefore,

$$f(\sigma) = T_\xi(f(D(\sigma))) = T_\xi(f(D(\xi\sigma_1))) = T_\xi(f(\sigma_1)) =$$
$$T_\xi(\sigma_1) = \xi\sigma_1 = \sigma.$$

This argument by induction on the length of the sequences involved can

be shown to have as its counterpart in our applied μ-calculus the following: Suppose we want to show $\vdash O_1 \subseteq O_2$, for some terms O_1, O_2. We first show that

(17) $\qquad \vdash A_\varepsilon O_1 \subseteq A_\varepsilon O_2$

and next we show that, for each $\xi \in \Sigma$

(18) $\qquad XO_1 \subseteq XO_2 \vdash XT_\xi O_1 \subseteq XT_\xi O_2.$

From (17) and (18) we then infer $\vdash O_1 \subseteq O_2$ as follows: By A_5, $\vdash O_1 \subseteq O_2$ may replaced by

$$\vdash \mu X[(p_\xi \to (DXT)^\xi, A_\varepsilon)]O_1 \subseteq \mu X[(p_\xi \to (DXT)^\xi, A_\varepsilon)]O_2.$$

By the μ-induction rule, it is sufficient to show that

$$XO_1 \subseteq XO_2 \vdash (p_\xi \to (DXT)^\xi, A_\varepsilon)O_1 \subseteq (p_\xi \to (DXT)^\xi, A_\varepsilon)O_2$$

which easily follows from (17) and (18). For later use, we call this inductive argument rule I.

Before we investigate some examples of more complex operations which can be expressed in terms of our primitives, we first list a number of properties which are direct consequences of the axioms:

LEMMA 2

a. $\vdash T_\xi(p_\eta \to O^\eta, O') = T_\xi O_\xi$

b1. $\vdash (p_\xi \to (DT)^\xi, A_\varepsilon) = E$

b2. $\vdash (p_\xi \to (DT)^\xi, E) = E$

c. $\vdash (p_\xi \to DT_\xi, E) = E$

d. $\vdash (p_\xi \to (p_\eta \to X, Y), Z) = (p_\xi \to X, Z), \qquad \xi, \eta \in \Sigma,\ \xi = \eta,$

$\qquad\qquad = (p_\xi \to Y, Z), \qquad \xi, \eta \in \Sigma,\ \xi \neq \eta.$

e. $\vdash (p_\xi \to E, E) = E$

PROOF

a. Follows by repeated application of axiom A_4.

b1. Follows by A_5 and the fixed point property.

b2. From b1 and lemma 1, since

$$\vdash E = (p_\xi \to (DT)^\xi, A_\varepsilon) = (p_\xi \to (DT)^\xi, (p_\eta \to (DT)^\eta, A_\varepsilon)) = (p_\xi \to (DT)^\xi, E).$$

c. $\vdash (p_\xi \to DT_\xi, E) =$ (part b2)

$(p_\eta \to (DT)^\eta, E)(p_\xi \to DT_\xi, E) =$

$(p_\eta \to (DT)^\eta (p_\xi \to DT_\xi, E), (p_\xi \to DT_\xi, E)) =$ (lemma 1)

$(p_\eta \to (DT)^\eta (p_\xi \to DT_\xi, E), E).$

Also, if $\xi = \eta$, then

$$\vdash T_\eta (p_\xi \to DT_\xi, E) = T_\eta DT_\eta = T_\eta,$$

if $\xi \neq \eta$, then

$$\vdash T_\eta (p_\xi \to DT_\xi, E) = T_\eta.$$

Thus,

$$\vdash (p_\eta \to (DT)^\eta (p_\xi \to DT_\xi, E), E) = (p_\eta \to (DT)^\eta, E) = E.$$

d. Using part c, we have

$$\vdash (p_\xi \to (p_\eta \to X, Y), Z) = (p_\xi \to (p_\xi \to DT_\xi, E)(p_\eta \to X, Y), Z).$$

The proof then follows easily from A_4.

e. Follows from parts c and d.

Let us define

$$(p_\varepsilon \to X, Y) = (p_\xi \to (Y)^\xi, X)$$

i.e., a sequence is empty iff it begins with none of the symbols in Σ. We expect that $\vdash A_\varepsilon (p_\varepsilon \to X, Y) = A_\varepsilon X$. In order to prove this, we need

LEMMA 3

$$\vdash A_\varepsilon = \mu X[(p_\xi \to (DX)^\xi, E)].$$

PROOF. Call the right hand side P.

a. $\vdash A_\varepsilon \subseteq P$. Using A_5, lemma 2.b1, and the μ-induction rule, we have to show

$XA_\varepsilon \subseteq P \vdash (p_\eta \to (DXT)^\eta, E)A_\varepsilon \subseteq P.$

Using A_2 and lemma 2.b1, we obtain

$\vdash (p_\eta \to (DXT)^\eta, E)A_\varepsilon =$

$(p_\eta \to (DXTA_\varepsilon)^\eta, A_\varepsilon) =$

$(p_\eta \to (DXA_\varepsilon)^\eta, E).$

By the fixed point property, $\vdash P = (p_\xi \to (DP)^\xi, E)$, and the result follows, since

$XA_\varepsilon \subseteq P \vdash (p_\eta \to (DXA_\varepsilon)^\eta, E) \subseteq (p_\xi \to (DP)^\xi, E).$

b. $\vdash P \subseteq A_\varepsilon$. The proof of this is left to the reader.

LEMMA 4

a. $\vdash A_\varepsilon(p_\varepsilon \to X, Y) = A_\varepsilon X.$

b. $\vdash A_\varepsilon A_\varepsilon = A_\varepsilon.$

PROOF

a. Follows easily from lemma 3.

b. Direct from part a and lemma 2.b1.

4.2. SOME NON-PRIMITIVE OPERATIONS

We give a number of examples how to express certain operations in terms of the primitive ones.
We consider the following operations, which are listed together with their intended meaning:

a. L_ξ : $L_\xi(\sigma) = \sigma\xi.$

b. Inv: $\mathrm{Inv}(\eta_1\ \eta_2 \cdots \eta_{n-1}\ \eta_n) = \eta_n\ \eta_{n-1} \cdots \eta_2\ \eta_1$

$\mathrm{Inv}(\varepsilon) = \varepsilon.$

c. R : $R(\xi\sigma) = \sigma\xi$,

$R(\varepsilon) = \varepsilon$.

d. S : $S(\sigma\xi) = \xi\sigma$,

$S(\varepsilon) = \varepsilon$.

These operations are definable by

a. $L_\xi = \mu X[(p_\eta \to (DXT)^\eta, T_\xi)]$.

b. $Inv = \mu X[(p_\xi \to (DXL)^\xi, E)]$.

c. $R = (p_\xi \to (DL)^\xi, E)$.

d. $S = Inv\ R\ Inv$

The set of operations L , Inv, R, S is meant only as a representative sample. Many other examples could be given.
Properties of the four operations are listed in lemma 5.

LEMMA 5

a1. $\vdash A_\varepsilon L_\xi = A_\varepsilon T_\xi$

2. $\vdash T_\xi L_\eta = L_\eta T_\xi$

b1. $\vdash A_\varepsilon Inv = A_\varepsilon$

2. $\vdash T_\xi Inv = Inv\ L_\xi$

3. $\vdash L_\xi Inv = Inv\ T_\xi$

4. $\vdash Inv\ Inv = E$

c1. $\vdash A_\varepsilon R = A_\varepsilon$

2. $\vdash T_\xi R = L_\xi$

d1. $\vdash A_\varepsilon S = A_\varepsilon$

2. $\vdash T_\xi S = S(p_\eta \to (DT_\xi T)^\eta, T_\xi)$

e. $\vdash RS = SR = E$

PROOF. We prove only a selection.

a2. $\vdash T_\xi L_\eta = T_\xi(p_\zeta{\to}(DL_\eta T)^\zeta, T_\eta)$

$= T_\xi DL_\eta T_\xi = L_\eta T_\xi$

b3. $\vdash L_\xi \text{ Inv} = \text{Inv } T_\xi$

(i) $\subseteq$. We apply the μ-induction rule to L_ξ and show that

$Z \text{ Inv} \subseteq \text{Inv } T_\xi \vdash (p_\eta{\to}(DZT)^\eta, T_\xi) \text{ Inv} \subseteq \text{Inv } T_\xi$.

We have

$\vdash (p_\eta{\to}(DZT)^\eta, T_\xi) \text{ Inv} =$

$(p_\eta{\to}(DZT \text{ Inv})^\eta, T_\xi \text{ Inv}) =$

$(p_\eta{\to}(DZ \text{ Inv } L)^\eta, \text{ Inv } L_\xi) =$ (b2)

$(p_\eta{\to}(DZ \text{ Inv } L)^\eta, (p_\zeta{\to}(D \text{ Inv } L)^\zeta, E)L_\xi) =$

$(p_\eta{\to}(DZ \text{ Inv } L)^\eta, L_\xi) =$

$(p_\eta{\to}(DZ \text{ Inv } L)^\eta, T_\xi) =$

and

$\vdash \text{Inv } T_\xi =$

$(p_\eta{\to}(D \text{ Inv } L)^\eta, E) \; T_\xi =$

$(p_\eta{\to}(D \text{ Inv } L \; T_\xi)^\eta, T_\xi) =$ (a2)

$(p_\eta{\to}(D \text{ Inv } T_\xi L)^\eta, T_\xi)$.

The desired result then follows from

$Z \text{ Inv} \subseteq \text{Inv } T_\xi \vdash (p_\eta{\to}(DZ \text{ Inv } L)^\eta, T_\xi) \subseteq (p_\eta{\to}(D \text{ Inv } T_\xi L)^\eta, T_\xi)$.

(ii) $\supseteq$. This proof is similar to part (i) and omitted.

b4. We show only that $\vdash E \subseteq \text{Inv Inv}$:

$\vdash \text{Inv Inv} =$

$(p_\xi{\to}(D \text{ Inv } L)^\xi, E) \text{ Inv} =$

$(p_\xi{\to}(D \text{ Inv } L \text{ Inv})^\xi, \text{Inv}) =$ (b3)

$(p_\xi{\to}(D \text{ Inv Inv } T)^\xi, E)$.

The result then follows by A_5.

d2. We show

$\vdash T_\xi S = S(p_\eta \to (DT_\xi T)^\eta, T_\xi)$.

Using the definition of S, b2 and b4, we reduce this to

$\vdash T_\xi \text{ Inv } R \text{ Inv} = \text{Inv } R \text{ Inv}(p_\eta \to (DT_\xi T)^\eta, T_\xi)$, or

$\vdash \text{Inv } L_\xi R \text{ Inv} = \text{Inv } R \text{ Inv}(p_\eta \to (DT_\xi T)^\eta, T_\xi)$, or

$\vdash L_\xi R \text{ Inv} = R \text{ Inv}(p_\eta \to (DT_\xi T)^\eta, T_\xi)$.

We have

$\vdash L_\xi R \text{ Inv} =$

$(p_\eta \to (DL_\xi T)^\eta, T_\xi) R \text{ Inv} =$

$(p_\eta \to (DL_\xi TR \text{ Inv})^\eta, T_\xi R \text{ Inv}) =$ (c2)

$(p_\eta \to (DL_\xi L \text{ Inv})^\eta, L_\xi \text{ Inv}) =$

$(p_\eta \to (DL_\xi \text{ Inv } T)^\eta, \text{Inv } T_\xi) =$

$(p_\eta \to (DL_\xi \text{ Inv } T)^\eta, T_\xi)$

and

$\vdash R \text{ Inv}(p_\eta \to (DT_\xi T)^\eta, T_\xi) =$

$(p_\zeta \to (DL)^\zeta, E) \text{ Inv}(p_\eta \to (DT_\xi T)^\eta, T_\xi) =$

$(p_\zeta \to (DL \text{ Inv})^\zeta, E) (p_\eta \to (DT_\xi T)^\eta, T_\xi) =$

$(p_\zeta \to (D \text{ Inv } T)^\zeta, E) (p_\eta \to (DT_\xi T)^\eta, T_\xi) =$

$(p_\zeta \to (D \text{ Inv } T)^\zeta (p_\eta \to (DT_\xi T)^\eta, T_\xi), T_\xi) =$

$(p_\zeta \to (D \text{ Inv } TDT_\xi T)^\zeta, T_\xi) =$

$(p_\zeta \to (D \text{ Inv } T_\xi T)^\zeta, T_\xi)$

$(p_\zeta \to (DL_\xi \text{ Inv } T)^\zeta, T_\xi)$.

From these two results, the proof of d2 follows.

The results of lemma 5 suggest the following rule: For arbitrary 0, if we know that

a. $A_\varepsilon 0 = 0'$,

b. $T_\xi 0 = 00''_\xi$, for each $\xi \in \Sigma$,

then

$$0 = \mu X[(p_\xi \rightarrow (DX0'')^\xi, 0')].$$

The proof of this, i.e., of

$$(19) \qquad \begin{array}{l} A_\varepsilon 0 = 0',\ T_x 0 = 00''_x,\ \ldots,\ T_z 0 = 00''_z \vdash \\ 0 = \mu X[(p_\xi \rightarrow (DX0'')^\xi, 0')] \end{array}$$

is easily given by means of the derived rule of inference I of section 4.1. (Use is made of the fact that $A_\varepsilon 0 = 0' \vdash A_\varepsilon 0 = A_\varepsilon 0'$, which follows from lemma 4b.)

Clearly, the form of rule (19) directly corresponds to the structure of axiom A_5, which characterizes the inductive properties of the data structure we are dealing with. Thus, similar rules can be given for other data structures, provided that they have a counterpart of axiom A_5.

Let us consider an example from Dijkstra (unpublished) from this point of view:

The domain consists of pairs of non-negative integers. Let $p_>(x,y)$ be true iff $x > y$, and similar for $p_=$ and $p_<$. Let

$$A_1(x,y) = (x+y,y)$$

$$A_2(x,y) = (x,y+x)$$

$$B_1(x,y) = (x-y,y)$$

$$B_2(x,y) = (x,y-x)$$

$$A_0(x,y) = (x,x).$$

Then we have

$$\vdash \mu X[(p_> \rightarrow B_1 X A_1, (p_< \rightarrow B_2 X A_2, A_0))] = E$$

as counterpart of A_5.

We want to define the greatest common divisor, say G. We know that

$$A_1G = G$$

$$A_2G = G$$

$$A_0G = A_0.$$

Then, by the analogue of rule (19), with $O' = A_0$, $O'' = E$, we have

$$\vdash G = \mu X[(p_{>} \rightarrow B_1X, (p_{<} \rightarrow B_2X, A_0))].$$

Various alternatives for G can now be expressed using properties of while statements, of which many examples are given in [1].

REFERENCES

[1] De Bakker, J.W., Recursive procedures, Mathematical Centre Tracts 24, Mathematical Centre, Amsterdam (1971).

[2] McCarthy, J., A basis for a mathematical theory of computation, in Computer Programming and Formal Systems, 33-70 (eds. P. Braffort and D. Hirschberg), Amsterdam, North-Holland (1963).

[3] Scott, D. & J.W. de Bakker, A theory of programs, unpublished notes, IBM Seminar, Vienna (1969).

[4] Scott, D., in Minutes of the fourth meeting of the IFIP Working Group 2.2 on Formal Language Description Languages. Essex University (1969).

THE USE OF APL IN COMPUTER DESIGN

G.A. Blaauw *)

ABSTRACT

This paper describes the use of the terminal implementation of APL for the specification and verification of a hardware architecture, the description of an initial implementation algorithm, its verification and its controlled transformation to a logical design, which in turn can be translated into an automation of design language. This computer aided design process is illustrated by the architecture and implementation of a divide operation, with remainder and overflow determination.

1. INTRODUCTION

In the design of a computer, or in fact any apparatus, three phases can be recognized (Blaauw [3]). The first is the architecture, which specifies what functions the computer can perform. The second is the implementation, which states how these functions can be performed by a logical structure. The third is the realisation, which concerns the physical structure which embodies the logic. In particular the components which are selected and the locations where they are placed are of concern here.

The terms architecture, implementation and realisation can apply to various levels and segments of design, such as an operating system, or a peripheral device. They will be used here with reference to computer hardware as seen by the system programmer.

Documentation of realisation and implementation

Already in the design of the earliest computers much attention was given to proper documentation as an aid in assuring correctness. Two major

---

*) Technological University, Twente

methods of documentation can be recognized right from the start. One is the use of diagrams, showing components in a symbolic way and representing interconnections by lines. This representation had its origin in electric wiring diagrams, most notably those used in electromechanical tabulating machines and electronic circuits. It eventually developed into diagrams consisting of a multitude of blocks. The other method of representation had its source in the algebra. It basically consisted of a number of logical statements representing the circuits. Names were used to indicate signals and hence implied the interconnection.

The classical example of an algebraic description is the 1957 paper on the LGP-30 (Frankel [10]). However, already the Harvard Mark IV, in 1951, (Blaauw [1]) and the electronic ARRA of the Mathematical Centre, in 1953, (van Wijngaarden [17], and Blaauw [2]) used simple logical statements to document the machine design. In these cases the design used 'and-or' statements with true and inverse outputs. These statements provided a complete description of the implementation. This information was combined with realization information. Thus the wiring lists of the output signals were given. These lists were used in the construction of the machine. They also provided a cross check for the proper occurrence of inputs to the 'and-or' circuits. Other realisation details which were included were pin-numbers, locations, circuit types and commentary. In later machines the processing of the realisation was more and more automated, resulting in the modern design automation programs.

Documentation of Architecture.

As architecture became distinguished from implementation the desire for a concise statement thereof also manifested itself. A milestone in this development was the formal description of the IBM System/360 in 1964 (Falkhoff [9]). The description was completed concurrent with the system development and not actually used by the designers. Nevertheless, it showed that a complex system could be described in all its details by a programming language. The particular language used was APL (Iverson [12]) as developed up to that point. The representation had a purely descriptive purpose.

Transition from architecture to implementation

The design of an implementation for a given architecture can be considered as the transformation of one description into another. In his 1962 book Iverson [12] had already shown that APL could be used to describe implementation logic. The ALERT system (Friedman [11]), based upon APL, demonstrated that the transformation from an architecture to such an implementation could in part be performed in an automated way, while the resulting implementation in turn could be used as the point of departure for the design automation of the realisation.

Simulation of the design

Several other languages have been proposed for the description of machine architecture and implementation. Some of these were implemented such that they could simulate the design which was described (Stabler [16]). In each case the language was developed or modified for the specific purpose of machine design.

Use of terminal implementation of APL

In 1968 APL was implemented (Breed [5]) as a terminal language (Pakin [14]). This implementation has had a beneficial effect upon the readability and generality of APL. The symbolism has been restricted and the absence of sub- and superscripts has not only simplified representation but at the same time extended the rank of the arrays to be treated.

In this paper the use of the general purpose language APL and its terminal implementation in the design of computer hardware will be described. In particular the architectural specification, its transformation by the designer into an implemention and the verification of the correctness of the design will be discussed.

It is assumed that the reader is familiar with APL. A summary of part of the language is found in appendix A. As an illustration of the design process the divide operation has been chosen. This example has sufficient complexity to illustrate various design decisions. Obviously several simplifications have been made.

The interaction with a terminal system results in a large number of

programs. The average reader is at a disadvantage in not being able to use the terminal digesting this information. However, he should not feel obliged to work through these programs more than the text suggests or his curiosity requires.

2. ARCHITECTURE

The architectural definition is the starting point for the design of the implementation. A written version of this definition is desirable to reach a large audience. In particular the indirectly, but vitally involved people in maintenance, machine operation, education, sales and management should be able to understand the functioning of the equipment. On the other hand written descriptions tend to become abundant in words and stylized in expression. An algorithmic description in a language like APL, has the advantage of exactness and conciseness. Its disadvantages are the unfamiliarity of the symbols and the catastrophic effect of each error. The problem of unfamiliarity can be reduced by the use of a general purpose language. The widespread use of APL as an interactive language makes it in particular attractive. Unfamiliarity can also be reduced by using the written and the algorithmic description side by side. Possible misconceptions in the text can be eliminated by the expressions, while the expressions in turn are explained by the text. Each description however should be completed in itself. The error problem is reduced considerably by a terminal implementation of the algorithmic language. Such an implementation can verify the description for syntactical correctness and reduce the number of transcription errors by proper editing functions.

Architecture in words

A typical description of hardware division could be that "the quotient QT and remainder RT are formed, which result, when the dividend DT is divided by the divisor DR". This description however is typically deficient in omitting critical detail. First, the overflow requires the clause "provided the quotient is defined and can be represented in which case the signal OVDL is zero, otherwise the signal OVDL is one and quotient and remainder

are arbitrary". Secondly, the concept of representation requires elaboration. Somewhere it should be stated how numbers are represented. In the current example 2-complement binary integer representation is assumed.

Architecture in APL

An APL description of the architecture is shown in program 1.

```
     ∇ ARCHDIV
[1]    ⍝ VALUE QUOTIENT
[2]    QNT←(¯1*DT[0]≠DR[0])×⌊|(TWEC DT)÷(TWEC DR)+0=∨/DR
[3]    ⍝ VALUE REMAINDER
[4]    RST←(TWEC DT)-QNT×TWEC DR
[5]    ⍝ QUOTIENT
[6]    QT←,(((⍴DT)-⍴DR)⍴2)⊤QNT
[7]    ⍝ REMAINDER
[8]    RT←,((⍴DR)⍴2)⊤RST
[9]    ⍝ OVERFLOW
[10]   OVDL←(QNT≠TWEC QT)∨0=∨/DR
     ∇

     ∇ Z←TWEC Y
[1]    Z←(2⊥Y)-Y[0]×2*⍴Y
     ∇
```

program 1. Constructive architecture division.

Statements 2 and 4 of ARCHDIV determine the value of quotient QNT and remainder RST. Next, lines 6 and 8 encode these quantities. The last line determines the value of the overflow indicator OVDL. The function ARCHDIV uses the function TWEC, which is also listed in program 1. TWEC determines the value of a number by interpreting its representation as 2-complement notation.

Characteristics of APL

A few of the characteristics of APL make it attractive in general (McCracken [13], and in particular for specifying a computer design, may be noted.

a. The language is concise. Thus, statement 2 of ARCHDIV determines the quotient as an integer, truncated towards zero, while a zero divisor is

replaced by the value 1 to avoid an indeterminate operation.

b. The language is systematic. All statements are executed right to left, no operators may be elided. Arithmetic and logical operators can be included in the same statement, as is the case in line 2. Also, an operator like absolute value requires only the single simbol and no longer introduces the concept of priority as expressed by brackets. Finally, negative number are identified by a minus sign, as in line 2, and not by a subtraction operator, as appears in line 6. Other points could be added.

c. The language generalises useful algebraic concepts. Thus reduction ⊕/ as a generalization of Σ and Π applies to any dyadic operation ⊕. In statement 2, for instance, the "or" reduction, ∨/, is used to determine if any bit of DR is 1. Similar generalisations apply to the vector product, while at the same time array operations are systematically performed element by element.

d. The language is array oriented. Many loops are avoided by treating arrays as an intity. Thus the variables which appear in a computer design, like DR and DT, are treated as logical vectors and as a rule can be handled in a single statement. Useful array operators like the dyadic and monadic ρ, the decode ⊥ and encode ⊤ and the index generator ∫ are available. For example in statement 6 the number QNT is encoded into a binary representation whose length is the difference of dividend and divisor length.

Verification of architecture

The architectural definition is independent of the length of the operands. This independence is valuable in verifying the correctness of algorithms. Thus in table 2 all 64 cases of a 4 bit dividend and 2 bit divisor are tabulated. The number to the left of the equal sign is the quotient; the number to the right of the letter R is the remainder. The asterisk indicates an overflow. The interpretation of dividend and divisor is also shown. Due to the unsymmetric nature of the 2-complement representation, the table is far from symmetric.

```
00 = 0000 ÷ 00 R 00 *   0 ÷  0      00 = 1000 ÷ 00 R 00 *  -8 ÷  0
00 = 0000 ÷ 01 R 00     0 ÷  1      00 = 1000 ÷ 01 R 00 *  -8 ÷  1
00 = 0000 ÷ 10 R 00     0 ÷ -2      00 = 1000 ÷ 10 R 00 *  -8 ÷ -2
00 = 0000 ÷ 11 R 00     0 ÷ -1      00 = 1000 ÷ 11 R 00 *  -8 ÷ -1
01 = 0001 ÷ 00 R 01 *   1 ÷  0      01 = 1001 ÷ 00 R 01 *  -7 ÷  0
01 = 0001 ÷ 01 R 00     1 ÷  1      01 = 1001 ÷ 01 R 00 *  -7 ÷  1
00 = 0001 ÷ 10 R 01     1 ÷ -2      11 = 1001 ÷ 10 R 11 *  -7 ÷ -2
11 = 0001 ÷ 11 R 00     1 ÷ -1      11 = 1001 ÷ 11 R 00 *  -7 ÷ -1
10 = 0010 ÷ 00 R 10 *   2 ÷  0      10 = 1010 ÷ 00 R 10 *  -6 ÷  0
10 = 0010 ÷ 01 R 00 *   2 ÷  1      10 = 1010 ÷ 01 R 00 *  -6 ÷  1
11 = 0010 ÷ 10 R 00     2 ÷ -2      11 = 1010 ÷ 10 R 00 *  -6 ÷ -2
10 = 0010 ÷ 11 R 00     2 ÷ -1      10 = 1010 ÷ 11 R 00 *  -6 ÷ -1
11 = 0011 ÷ 00 R 11 *   3 ÷  0      11 = 1011 ÷ 00 R 11 *  -5 ÷  0
11 = 0011 ÷ 01 R 00 *   3 ÷  1      11 = 1011 ÷ 01 R 00 *  -5 ÷  1
11 = 0011 ÷ 10 R 01     3 ÷ -2      10 = 1011 ÷ 10 R 11 *  -5 ÷ -2
01 = 0011 ÷ 11 R 00 *   3 ÷ -1      01 = 1011 ÷ 11 R 00 *  -5 ÷ -1
00 = 0100 ÷ 00 R 00 *   4 ÷  0      00 = 1100 ÷ 00 R 00 *  -4 ÷  0
00 = 0100 ÷ 01 R 00 *   4 ÷  1      00 = 1100 ÷ 01 R 00 *  -4 ÷  1
10 = 0100 ÷ 10 R 00     4 ÷ -2      10 = 1100 ÷ 10 R 00 *  -4 ÷ -2
00 = 0100 ÷ 11 R 00 *   4 ÷ -1      00 = 1100 ÷ 11 R 00 *  -4 ÷ -1
01 = 0101 ÷ 00 R 01 *   5 ÷  0      01 = 1101 ÷ 00 R 01 *  -3 ÷  0
01 = 0101 ÷ 01 R 00 *   5 ÷  1      01 = 1101 ÷ 01 R 00 *  -3 ÷  1
10 = 0101 ÷ 10 R 01     5 ÷ -2      01 = 1101 ÷ 10 R 11    -3 ÷ -2
11 = 0101 ÷ 11 R 00 *   5 ÷ -1      11 = 1101 ÷ 11 R 00 *  -3 ÷ -1
10 = 0110 ÷ 00 R 10 *   6 ÷  0      10 = 1110 ÷ 00 R 10 *  -2 ÷  0
10 = 0110 ÷ 01 R 00 *   6 ÷  1      10 = 1110 ÷ 01 R 00    -2 ÷  1
01 = 0110 ÷ 10 R 00 *   6 ÷ -2      01 = 1110 ÷ 10 R 00    -2 ÷ -2
10 = 0110 ÷ 11 R 00 *   6 ÷ -1      10 = 1110 ÷ 11 R 00 *  -2 ÷ -1
11 = 0111 ÷ 00 R 11 *   7 ÷  0      11 = 1111 ÷ 00 R 11 *  -1 ÷  0
11 = 0111 ÷ 01 R 00 *   7 ÷  1      11 = 1111 ÷ 01 R 00    -1 ÷  1
01 = 0111 ÷ 10 R 01 *   7 ÷ -2      00 = 1111 ÷ 10 R 11    -1 ÷ -2
01 = 0111 ÷ 11 R 00 *   7 ÷ -1      01 = 1111 ÷ 11 R 00    -1 ÷ -1
```

table 2. Exhaustive printout of 4-bit dividend divided by 2-bit divisor.

Such a tabulation of course does not constitute a proof of correctness. However, since it includes a large number of special cases, it is valuable in giving a first order indication of the implications of the algorithm. Often, in fact, this approach is the only way to ascertain that the architecture as given by the algorithm really expresses the intention of the designer, since a generally agreed definition, as exists for division, may not be available.

Constructive and descriptive architecture

The architectural definition of ARCHDIV makes use of the algebraic divide operator, ÷. As a result the architecture can be constructive. That is, for a given divendend and divisor, the quotient, remainder and overflow indication can be determined. An architecture, however, need not be constructive, it can be merely descriptive. Thus in program 3 the function CHECKDIV also constitutes an architecture.

```
    ∇ CHECKDIV
[1]   ⍝ REMAINDER
[2]   CR←OVDL∨(~∨/RT)∨(DT[0]=RT[0])∧(|TWEC DR)>|TWEC RT
[3]   ⍝ QUOTIENT
[4]   CQ←OVDL∨(TWEC DT)=(TWEC RT)+(TWEC DR)×TWEC QT
[5]   ⍝ OVERFLOW
[6]   CO←OVDL=(|TWEC DT)≥(|TWEC DR)×(DT[0]≠DR[0])+2*¯1+(⍴DT)-⍴DR
[7]   TRUE←CO∧CR∧CQ
    ∇
```

program 3. Descriptive architecture division.

Here statement 2 verifies that the remainder is absolute smaller than the divisor and has the same sign as the dividend, unless it is zero or an overflow occurs. Similarly, statement 4 requires that quotient times divisor plus remainder equals dividend. Finally, statement 6 ascertains that an overflow occurs when the largest possible quotient and remainder are too small for the given dividend and divisor. The descriptive architecture describes correctly the arbitrary value of quotient and remainder in case of overflow. The constructive architecture constructs an arbitrary value as required. However, the designer must be aware of this, since the algorithm can not state the fact that this value is arbitrary.

3. IMPLEMENTATION

The architecture of CHECKDIV corresponds closely to the algebraic definition of division, which also is descriptive rather than constructive. Since ARCHDIV depends upon algebraic division, the implementer is given little help in constructing an algorithm which can perform division. In

contrast, the algebraic definition of multiplication is constructive by referring to repeated addition. Most multiplication algorithms are based upon this definition and merely reduce the number of additions. Division algorithms on the other hand, typically depend upon trial and error as is illustrated by the simple restoring division algorithm for positive operands POSDIV, program 4 and figure 1. POSDIV corresponds to the traditional longhand division, only the trial subtraction is more explicit.

```
     ∇ POSDIV
[1]    ⍝ PROLOGUE
[2]    P←(⍴DR)↑DT
[3]    Q←(⍴DR)↓DT
[4]    R←DR
[5]    ⍝ DIVISION
[6]    J←⍴Q
[7]    ⍝ REDUCTION
[8]   RED:A←P,Q[0]
[9]    B←~R[0],R
[10]   CIN←1
[11]   ARCHADD
[12]   P←1↓S
[13]   Q←1↓Q,COUT
[14]   →COUT/TEL
[15]   ⍝ RESTORATION
[16]   A←P
[17]   B←R
[18]   CIN←0
[19]   ARCHADD
[20]   P←S
[21]  TEL:J←J-1
[22]   →(J≠0)/RED
[23]   ⍝ EPILOGUE
[24]   RT←P
[25]   QT←Q
[26]   OVDL←Q[0]
     ∇
```

program 4. Restoring division for positive operands.

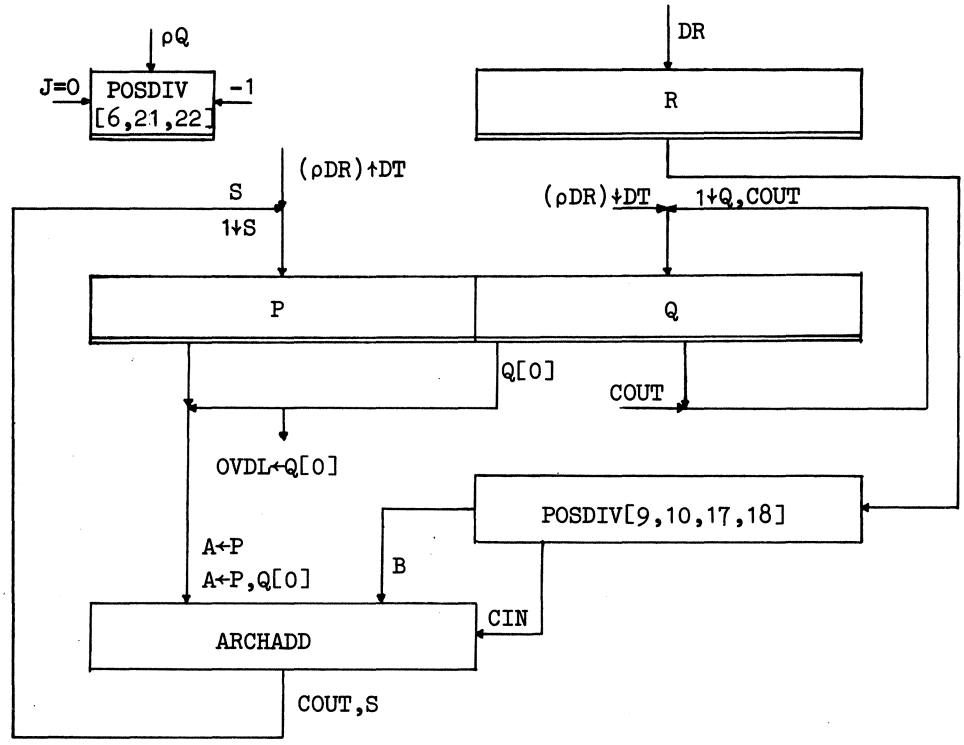


figure 1. Block diagram restoring division for positive operands.

Initial implementation algorithm

POSDIV is a constructive algorithm whose operators can be build in principle. As such it represents a first step towards a complete implementation. The algorithm has a prologue and epilogue which relate the contents of the register P, Q and R to the operands and results of the division.

Deferred substitution

Within the division proper, lines 6-22 of POSDIV, advantage is still taken of algebraic operators. Thus, the subtraction of the count in line 21 eventually should be performed by logical operators. When the designer

is confident that such a counter poses no serious problems, he can defer the introduction of this detail. Note however, that this deferred substitution does not make the algorithm less exact. The adder similarly is introduced in lines 11 and 19 by its architecture ARCHADD, program 5.

```
     ∇ ARCHADD
[1]    ⍝ SUM
[2]    S←((⍴A)⍴2)⊤CIN+(2⊥A)+2⊥B
[3]    ⍝ CARRY OUT
[4]    COUT←(2⊥S)≠CIN+(2⊥A)+2⊥B
     ∇
```

program 5. Architecture addition.

An example of an adder implementation which the designer may choose to substitute later in his design for ARCHADD is the simple ripple adder of program 6.

```
     ∇ RIPPLE
[1]    G←A∧B
[2]    T←A≠B
[3]    J←⍴A
[4]    C←(J⍴0),CIN
[5]    J←J-1
[6]    C[J]←G[J]∨T[J]∧C[J+1]
[7]    →(J≠0)/5
[8]    COUT←C[0]
[9]    S←T≠1↓C
     ∇
```

program 6. Ripple adder.

Another item of deferred substitution is the initial count, which is entered in line 6 of POSDIV. Entering the quotient length as the initial count is of course a simple matter. By using the current form of statement 6 the algorithm remains independent of operand sizes.

Administrative detail

The rippel adder of program 6 shows the administrative detail which at this stage of the design may still be included in the algorithm. The carry circuit of statement 6 is expressed in terms of the index J. The circuit should be repeated for every bit position, as is ensured by statement 3, 5 and 7. These statements therefore do not correspond to logical circuits. They will be removed when the operand sizes are known and the algorithm is transformed for use in the realisation.

Verification of initial algorithm

Terminal functions, like tracing, and auxiliary functions, which the designer may construct himself, can establish quickly the approximate correctness of an algorithm. Also, the independence of operand sizes again permits exhaustive verification for small operands. Thus program 7 shows a test program TESTDIV which compares POSDIV, or any other implementation, against its architecture.

```
    ∇ TESTDIV
[1]   ⍝ SET IN ADVANCE: N←DIVIDEND LENGTH, M←DIVISOR LENGTH, X←COUNT
[2]   ⍝ GENERATION OF DIVIDEND DT AND DIVISOR DR
[3]   H←((N+M)⍴2)⊤X
[4]   DT←N↑H
[5]   DR←N↓H
[6]   ⍝ ONLY POSITIVE OPERANDS IF POS IS SET TO 1
[7]   →(~POS)/7
[8]   ⍝ MAKE OPERANDS POSITIVE
[9]   DT← 0 0 ,DT
[10]  DR←0,DR
[11]  ⍝ ALGORITHM TO BE TESTED
[12]  POSDIV
[13]  ⍝ TESTALGORITHM
[14]  CHECKDIV
[15]  →TRUE/NEXT
[16]  ⍝ PRINT TROUBLE REPORT
[17]  '01 ←÷R'[QT, 2 3 2 ,DT, 2 4 2 ,DR, 2 5 2 ,RT];' ⍕'[0,OVDL];' WRONG'
[18]  ARCHDIV
[19]  '01 ←÷R'[QT, 2 3 2 ,DT, 2 4 2 ,DR, 2 5 2 ,RT];' ⍕'[0,OVDL];' RIGHT'
[20] NEXT:X←X-1
[21]  →(X≥0)/1
    ∇
```

program 7. Division test routine.

Fast verification of an implementation against a large number of test cases is attractive for the logical designer, even if not convincing to the mathematician. However, the algorithmic statement is also suitable for algebraic verification, by inductive methods, by the recognition of invariance properties and algebraic substitution (Du Croix, Duijvestijn). In turn the designer may not feel sufficiently secure about his algebraic reasoning to rely entirely on such an approach.

Block diagram of algorithm

Figure 1 shows the equipment, which the designer may have postulated for POSDIV, in the form of a block diagram. The diagram reflects the initial nature of the algorithm. Signals which share the adder are just combined, without reference to gates or controls. Nevertheless the diagram is useful to visualise the flow of information and the amount of equipment involved.

Use of APL in block diagram

The illustrative character of a block diagram has an inner contradiction. The more is shown, the less clear the drawing becomes. Therefore, groups of signals are represented by a single line and not all lines are drawn in full length, as for the signal COUT. Also, it is not practical to work out in detail the relative size and location of lines and rectangles. These omissions eventually lead to incompleteness. However, with a language like APL a block diagram can be made complete again. To that end each unit is identified by an expression, or, if this is too space consuming, by a name referring to another diagram or a series of expressions.

Improved initial algorithm

The initial algorithm may require further manipulation to extend its scope and improve its performance. Thus program 8, NEGDIV, extends restoring division to negative operands, while program 9, NONRES, illustrated in figure 2, improves performance by using the wellknown nonrestoring division algorithm.

```
     ∇ NEGDIV
[1]    ⍝ PROLOGUE
[2]    P←(⍴DR)↑DT
[3]    Q←(⍴DR)↓DT
[4]    R←DR
[5]    ⍝ DIVISION
[6]    J←⍴Q
[7]    U←P[0]
[8]    W←P[0]
[9]    ⍝ REDUCTION
[10] RED:A←P,Q[0]
[11]   B←(W=R[0])≠R[0],R
[12]   CIN←W=R[0]
[13]   ARCHADD
[14]   P←1↓S
[15]   Q←1↓Q,COUT≠R[0]
[16]   W←~COUT
[17]   →((W=U)∨~∨/W,P)/TEL
[18]   ⍝ RESTORATION
[19]   A←P
[20]   B←(W=R[0])≠R
[21]   CIN←W=R[0]
[22]   ARCHADD
[23]   P←S
[24]   W←~COUT
[25] TEL:J←J-1
[26]   →(J=0)/QTC
[27]   ⍝ ZEROTEST
[28]   →(∨/W,P,Q[0])/RED
[29]   ⍝ SHIFT
[30]   Q←1↓Q,R[0]
[31]   →TEL
[32]   ⍝ QUOTIENT CORRECTION
[33] QTC:A←Q
[34]   B←0
[35]   CIN←W≠R[0]
[36]   ARCHADD
[37]   Q←S
[38]   W←~COUT
[39]   ⍝ OVERFLOW
[40]   OVLP←(Q[0]∨~W)≠U≠R[0]
[41]   ⍝ EPILOGUE
[42]   RT←P
[43]   QT←Q
[44]   OVDL←OVLP
     ∇
```

program 8. Restoring division for positive and negative operands.

```
     ∇ NONRES
[1]    ⍝ PROLOGUE
[2]    P←(⍴DR)↑DT
[3]    Q←(⍴DR)↓DT
[4]    R←DR
[5]    ⍝ DIVISION
[6]    J←⍴Q
[7]    U←P[0]
[8]    W←P[0]
[9]    ⍝ REDUCTION
[10] RED:A←P,Q[0]
[11]   B←(W=R[0])≠R[0],R
[12]   CIN←W=R[0]
[13]   ARCHADD
[14]   P←1↓S
[15]   Q←1↓Q,COUT≠R[0]
[16]   W←~COUT
[17] TEL:J←J-1
[18]   →(J=0)/RTC
[19]   ⍝ ZEROTEST
[20]   →(∨/W,P,Q[0])/RED
[21]   ⍝ SHIFT
[22]   Q←1↓Q,R[0]
[23]   →TEL
[24] RTC:→((W=U)∨~∨/W,P)/QTC
[25]   ⍝ REMAINDER CORRECTION
[26]   A←P
[27]   B←(W=R[0])≠R
[28]   CIN←W=R[0]
[29]   ARCHADD
[30]   P←S
[31]   W←~COUT
[32]   ⍝ QUOTIENT CORRECTION
[33] QTC:A←Q
[34]   B←0
[35]   CIN←W≠R[0]
[36]   ARCHADD
[37]   Q←S
[38]   W←~COUT
[39]   ⍝ OVERFLOW
[40]   OVLP←(Q[0]∨~W)≠U≠R[0]
[41]   ⍝ EPILOGUE
[42]   RT←P
[43]   QT←Q
[44]   OVDL←OVLP
     ∇
```

program 9. Nonrestoring division.

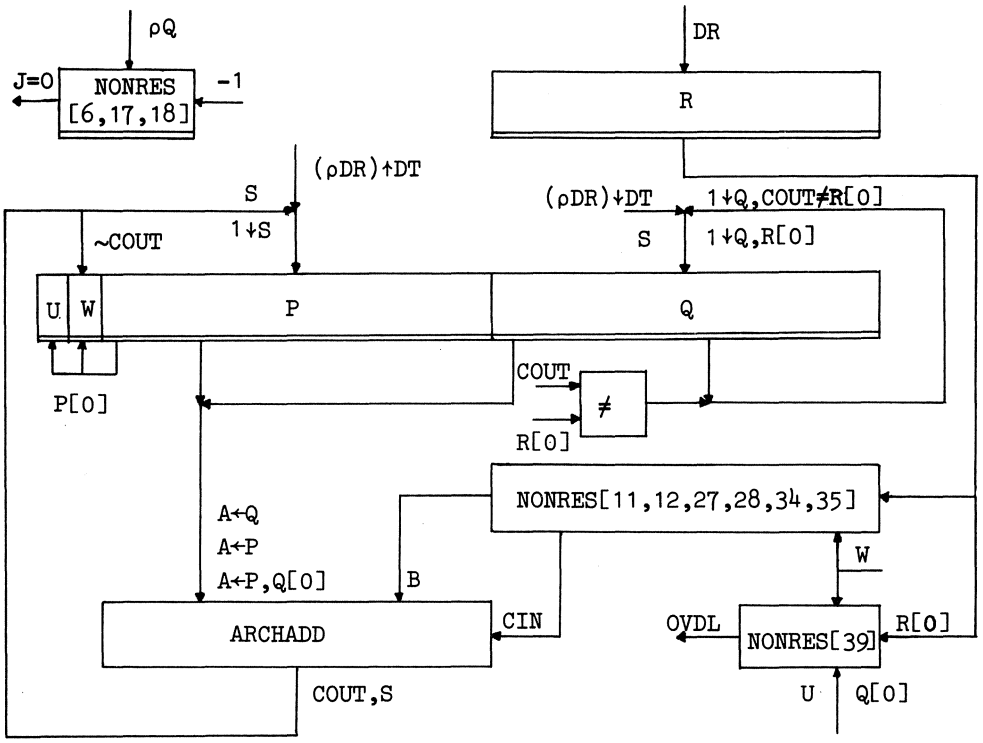


figure 2. Block diagram nonrestoring division.

A comparison of the two programs shows that the restoration which in POSDIV or NEGDIV may occur during any cycle, is deferred to the last cycle in NONRES.

The use of negative operands introduces several complexities. The quotient is originally recorded in 1-complement rather than 2-complement and hence requires a final correction in lines 33-37. A partial dividend which is zero must be shifted rather then reduced, to secure the case of a zero remainder for a negative dividend. This is shown in lines 27-30 of NEGDIV and 19-22 of NONRES. Finally, the correct overflow indication should be obtained. This is done in line 40 by verifying the quotient sign against dividend and divisor sign, taking a zero quotient into account. The algorithms are carefully designed to yield the wrong quotient sign in all cases of overflow.

The reader is not expected to comprehend these programs immediately. They are used here as an illustration of the various algorithms which a designer may evaluate and modify to suit his precise needs. Thus the remainder correction in lines 25-31 of NONRES is skipped when not required, while the quotient correction in lines 32-38 is always performed, even if in line 35 the carry-in is zero and no change occurs. Another choice clearly could have been made. However, a negative dividend yielding a zero quotient always requires a quotient of all ones to be corrected to all zeros, which results in an overflow out of the adder. This overflow sets W in line 38 and subsequently can be used in the simple overflow circuit of line 40. This procedure employs similar statements, in lines 16,31 and 38, and avoids a zero test for the quotient. These considerations are a typical part of the design process. Obviously many alternative solutions could have been selected. The simple condition for remainder correction has only recently been published (Rhyne [15]). The algorithm NONRES was developed independent of this publication.

NONRES is a suitable point of departure for even faster algorithms using multiples of the divisor and shifting over several bits at a time, yet yielding exact remainder and overflow. For our present purpose however it will be assumed that NONRES is satisfactory as an initial algorithm and the subsequent designs steps will be considered next.

Separation of time and space

When the designer is satisfied with his initial algorithm, he should become more specific in the use of time and space. As a first step he should ensure that those statements which were intended to be performed by a single piece of equipment are suited to that purpose. Thus the additions implied in lines 13, 29 and 36 of NONRES can all be performed by the same adder. Indeed, this must have been the intention of the designer since otherwise he would have used a counter instead of an adder in statement 36. The use of a common adder requires that the operands have uniform length. Therefore the difference between the operand length in line 13 as opposed to the length in lines 29 and 36 should be eliminated. Another requirement

for the time sharing of the adder is the selection of the inputs. Hence gate signals must be introduced which select the various operands.

As a second step in the separation of time and space the designer should declare himself concerning sequential or combinational circuits. This step also can be taken gradually, leaving for later specification the exact nature of registers, latch back circuits, or even complex sequential circuits, rare as the latter may be in practical machine design. As a minimum, the input signals of a register should be distinguished in name from its output signals, with a simple statement indicating the transfer of information. Several languages introduce special functions to describe registers. In APL this was not found necessary.

Finally, the various test conditions must be identified as combinational circuits feeding the sequence control.

Modified implementation algorithm

Program 10, IMPDIV, shows the result of these modification to NONRES. (See page 2.19). Labels are used to indicate elementary time periods. All expressions between successive labels are valid during such a period. The sequence of these expressions indicates the flow of information through the logic. As a rule a variable which is specified in an expression does not appear in a preceding expression of this elementary period. As a result the expressions need only be executed once during simulation. At times a combinational circuit can be described more compactly by deviating from this rule. In such a case administrative expressions must be incorporated which repeat the sequence till the circuit has come to rest. Although this procedure is feasable, it is better avoided. In fact, fast simulation and simple verification, is so valuable, that the designer is justified to exclude any circuits, like the above mentioned sequential circuits, which do not permit this.

IMPDIV can be executed by setting all gates to 1. Thus equivalence with the architecture can be established again. However, it is also possible to verify equivalence of small subsequences of NONRES and IMPDIV. For instance it can be shown algebraically and by simulation that lines 26-31 of NONRES and 44-51 of IMPDIV are equivalent if the gates ROS, DOS, SOP and COW are all one.

```
     ∇ IMPDIV
[1]    ⍝ PROLOGUE
[2]    P←(ρDR)↑DT
[3]    Q←(ρDR)↓DT
[4]    R←DR
[5]    ⍝ DIVISION
[6]   NH0:IU←DOR∧P[0]
[7]    IW←DOR∧P[0]
[8]    IJ←DOR×ρQ
[9]   NH1:U←IU
[10]   W←IW
[11]   J←IJ
[12]   ⍝ REDUCTION
[13]  NH2:A←POS∧P,Q[0]
[14]   B←DOS∧(W=R[0])≠R[0],R
[15]   CIN←DOS∧W=R[0]
[16]   ARCHADD
[17]   IP←SOP∧1↓S
[18]   IQ←KOK∧1↓Q,COUT≠R[0]
[19]   IW←COW∧~COUT
[20]   IJ←TOK×J-1
[21]  NH3:P←IP
[22]   Q←IQ
[23]   W←IW
[24]   J←IJ
[25]   JNUL←J=0
[26]   →JNUL/NH9
[27]   ⍝ ZEROTEST
[28]  NH5:NULT←∨/W,P,Q[0]
[29]   →NULT/NH2
[30]   ⍝ SHIFT
[31]  NH6:A←0
[32]   B←0
[33]   CIN←0
[34]   ARCHADD
[35]   IQ←KOK∧1↓Q,COUT≠R[0]
[36]   IJ←TOK×J-1
[37]  NH7:Q←IQ
[38]   J←IJ
[39]   JNUL←J=0
[40]   →(~JNUL)/NH5
[41]  NH9:RCOR←(W=U)∨~∨/W,P
[42]   →RCOR/NH12
[43]   ⍝ REMAINDER CORRECTION
[44]  NH10:A←ROS∧P[0],P
[45]   B←DOS∧(W=R[0])≠R[0],R
[46]   CIN←DOS∧W=R[0]
[47]   ARCHADD
[48]   IP←SOP∧1↓S
[49]   IW←COW∧~COUT
[50]  NH11:P←IP
[51]   W←IW
[52]   ⍝ QUOTIENT CORRECTION
[53]  NH12:A←SOK∧Q[0],Q
[54]   B←0
[55]   CIN←SOK∧W≠R[0]
[56]   ARCHADD
[57]   IQ←SOK∧1↓S
[58]   IW←COW∧~COUT
[59]  NH13:Q←IQ
[60]   W←IW
[61]   ⍝ OVERFLOW
[62]   OVLP←(Q[0]∨~W)≠U≠R[0]
[63]   ⍝ EPILOGUE
[64]   RT←P
[65]   QT←Q
[66]   OVDL←OVLP
       ∇
```

program 10. Implementation form of nonrestoring division.

Program block diagram dataflow

In IMPDIV the function ARCHADD is used four times: in lines 16, 34, 47 and 56, even though the designer intends to build the corresponding adder only once. Therefore, as a next step IMPDIV can be transformed to correspond directly to the proposed dataflow. The result is the function BLOKDIV

of program 11, while the use of this dataflow by a nonrestoring division is specified in GATEDIV, GATE and CODE of program 12. Thus the separation of time and space has become explicit in the programs.

```
     ∇ BLOKDIV
[1]    ⍝ ARITHMETIC
[2]    A←(POS∧P,Q[0])∨(ROS∧P[0],P)∨SOK∧Q[0],Q
[3]    B←DOS∧(W=R[0])≠R[0],R
[4]    CIN←(DOS∧W=R[0])∨SOK∧W≠R[0]
[5]    ARCHADD
[6]    IP←SOP∧1↓S
[7]    IQ←(KOK∧1↓Q,COUT≠R[0])∨SOK∧1↓S
[8]    IW←(DOR∧P[0])∨COW∧~COUT
[9]    IU←DOR∧P[0]
[10]   IJ←(DOR×⍴Q)+TOK×J-1
[11]   ⍝ REGISTERS
[12]   P←(TIP∧IP)∨(~TIP)∧P
[13]   Q←(TIQ∧IQ)∨(~TIQ)∧Q
[14]   W←(TIW∧IW)∨(~TIW)∧W
[15]   U←(TIU∧IU)∨(~TIU)∧U
[16]   J←(TIJ×IJ)+(~TIJ)×J
[17]   ⍝ TESTCONDITIONS
[18]   JNUL←J=0
[19]   NULT←∨/W,P,Q[0]
[20]   RCOR←(W=U)∨~∨/W,P
[21]   OVLP←(Q[0]∨~W)≠U≠R[0]
     ∇
```

program 11. Dataflow for division.

```
     ∇ GATEDIV
[1]    ⍝ PROLOGUE
[2]    P←(⍴DR)↑DT
[3]    Q←(⍴DR)↓DT
[4]    R←DR
[5]    ⍝ DIVISION
[6]    GATE 22 27 30 31
[7]    ⍝ REDUCTION
[8]   PD2:GATE 16 19 23 25 21 24 28 29 30 31
[9]    →JNUL/PD9
[10]   ⍝ ZEROTEST
[11]  PD5:→NULT/PD2
[12]   ⍝ SHIFT
[13]   GATE 24 25 29 31
[14]   →(~JNUL)/PD5
[15]  PD9:→RCOR/PD12
[16]   ⍝ REMAINDER CORRECTION
[17]   GATE 18 19 23 21 28 30
[18]   ⍝ QUOTIENT CORRECTION
[19]  PD12:GATE 17 21 29 30
[20]   ⍝ EPILOGUE
[21]   RT←P
[22]   QT←Q
[23]   OVDL←OVLP
     ∇
```

```
     ∇ GATE X
[1]    OP←(⍳36)∈X
[2]    CODE
[3]    BLOKDIV
     ∇
```

```
     ∇ CODE
[1]    POS←OP[16]
[2]    SOK←OP[17]
[3]    ROS←OP[18]
[4]    DOS←OP[19]
[5]    MOR←OP[20]
[6]    COW←OP[21]
[7]    DOR←OP[22]
[8]    SOP←OP[23]
[9]    TOK←OP[24]
[10]   KOK←OP[25]
[11]   VOS←OP[26]
[12]   TIU←OP[27]
[13]   TIP←OP[28]
[14]   TIQ←OP[29]
[15]   TIW←OP[30]
[16]   TIJ←OP[31]
[17]   BEI←OP[32]
[18]   ABS←OP[33]
[19]   MIN←OP[34]
     ∇
```

program 12. Gatecontrol for division.

BLOKDIV can be obtained from IMPDIV by simply combining all different specifications of a variable with an 'or' operator. Thus line 2 of BLOKDIV is obtained from lines 13, 31, 44 and 53 of IMPDIV. For line 3 it proves that lines 14, 32, 45 and 54 all collapse to a single 'and' condition. Clearly the designer already had this in mind, as shown by his choice of gate signals and the details of the algorithm. This particular transformation of course could be automated. This would be an aid in verifying the correctness of the transformation, rather than a means of speeding the design process. Actually the editing functions of APL make it simple to perform this transformation in a controlled manner.

GATEDIV represents the sequential part of IMPDIV. The program is reduced to the control of gates and the decisions based upon test signals.

Amplifying the dataflow

At this time the designer may wish to augment the dataflow for division with the requirements of other operations. As an example program 13, BLOKPAR, shows a parallel dataflow which, besides division, can perform addition, sign control and cumulative multiplication. Some of the corresponding gate programs are shown in program 14.

```
     ∇ BLOKPAR
[1]    ⍝ ARITHMETIC
[2]    A←(POS∧P[0],P,Q[0])∨(ROS∧P[0 0],P)∨SOK∧Q[0 0],Q
[3]    V1←R COMPAR ¯3↑Q,V
[4]    B←V1[0],V1
[5]    ARCHADD
[6]    IP←(SOP∧2↓S)∨VOS∧¯2↓S
[7]    IQ←(KOK∧1↓Q,COUT≠R[0])∨(SOK∧2↓S)∨VOS∧(¯2↑S),¯2↓Q
[8]    IV←VOS∧1↑¯2↑Q
[9]    IW←(DOR∧P[0])∨COW∧~COUT
[10]   IU←DOR∧P[0]
[11]   IJ←(DOR×⍴Q)+(MOR×(⍴Q)÷2)+TOK×J-1
[12]   ⍝ REGISTERS
[13]   P←(TIP∧IP)∨(~TIP)∧P
[14]   Q←(TIQ∧IQ)∨(~TIQ)∧Q
[15]   W←(TIW∧IW)∨(~TIW)∧W
[16]   U←(TIU∧IU)∨(~TIU)∧U
[17]   J←(TIJ×IJ)+(~TIJ)×J
[18]   V←IV
[19]   ⍝ TESTCONDITIONS
[20]   JNUL←J=0
[21]   NULT←∨/W,P,Q[0]
[22]   RCOR←(W=U)∨~∨/W,P
[23]   OVLP←(Q[0]∨~W)≠U≠R[0]
     ∇

     ∇ VV←R COMPAR F
[1]    ONE←DOS∨BEI∨VOS∧F[1]≠F[2]
[2]    TWO←VOS∧((~F[0])∧F[1]∧F[2])∨F[0]∧(~F[1])∧~F[2]
[3]    INV←(DOS∧W=R[0])∨(VOS∧F[0])∨BEI∧MIN≠ABS∧R[0]
[4]    COM←INV≠R
[5]    VV←(ONE∧COM[0],COM)∨TWO∧COM,INV
[6]    CIN←(SOK∧W≠R[0])∨INV∧ONE∨TWO
     ∇
```

program 13. Parallel dataflow.

```
     ∇ GATEPOS
[1]    ⍝ PROLOGUE
[2]    P←(ρA)ρ0
[3]    Q←(ρA)ρ0
[4]    R←A
[5]    ⍝ ABSOLUTE VALUE
[6]    GATE 23 28 32 33
[7]    ⍝ EPILOGUE
[8]    B←P
     ∇

     ∇ GATEMIN
[1]    ⍝ PROLOGUE
[2]    P←A
[3]    Q←(ρA)ρ0
[4]    R←B
[5]    ⍝ SUBTRACTION
[6]    GATE 18 23 28 32 34
[7]    ⍝ EPILOGUE
[8]    B←P
     ∇


     ∇ GATEMUL
[1]    ⍝ PROLOGUE
[2]    P←BW
[3]    Q←VR
[4]    R←VT
[5]    ⍝ MULTIPLICATION
[6]    GATE 20 31
[7]    ⍝ STROKE
[8]   PV2:GATE 18 24 26 28 29 31
[9]    →(~JNUL)/PV2
[10]   ⍝ EPILOGUE
[11]   PD←P,Q
     ∇
```

program 14. Gate control absolute value, subtraction and multiplication.

The program GATEDIV still applies, except that the reference in GATE is now to BLOKPAR. Again these amplifications can be performed in a controlled manner, assuring the correctness of the design. The designer furthermore may wish to explore more parallel or more serial operation. He also may wish to make adjustments to optimise performance of a given instruction mix.

Thus, in this example, multiplication is twice as fast as division, on the assumption that the former is more frequent than the latter.

Complete block diagram

Figure 3 shows the blockdiagram corresponding to BLOKPAR.

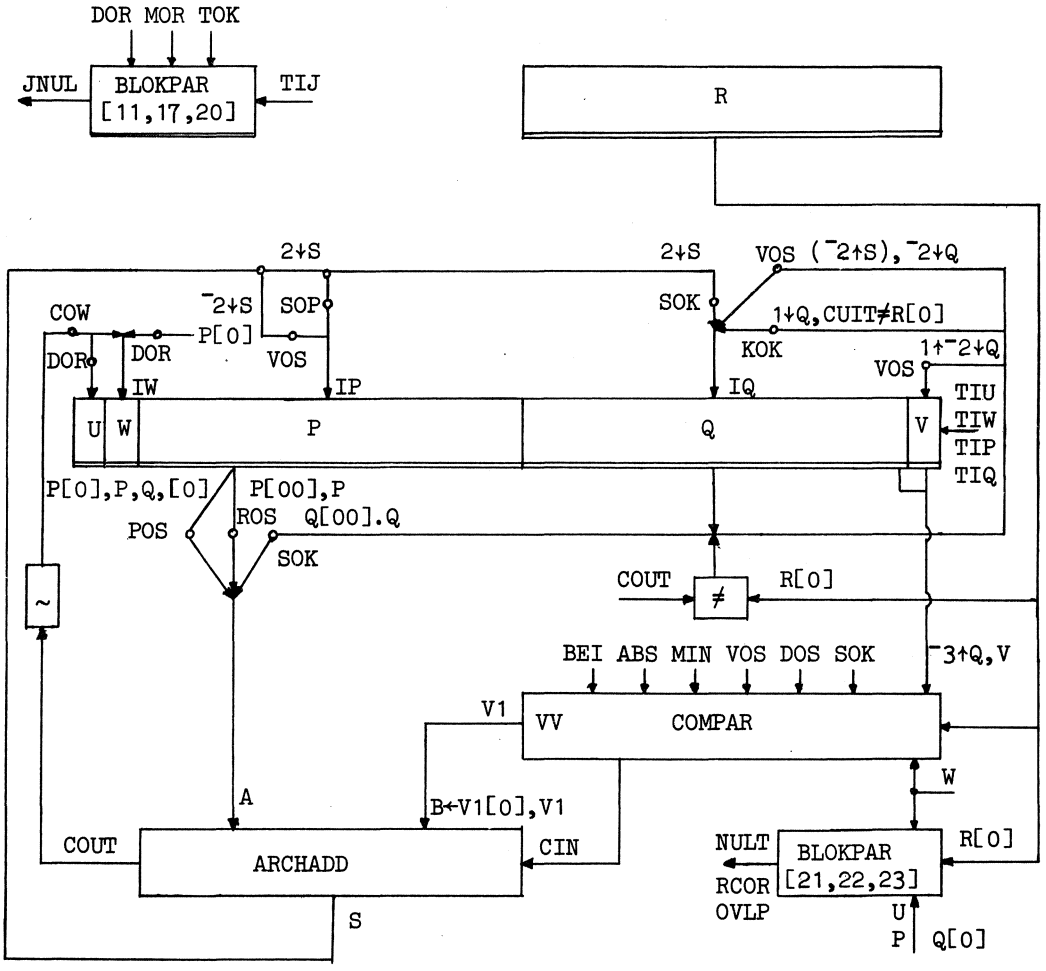


figure 3. Parallel dataflow.

The diagram reflects the later state of the design effort. The gates and control signals are now shown. The joining of wires implies an 'of' circuit. All information is available for a complete logical design. The figure still uses the pictorial implications of lines and arrows. For instance, a signal entering a register is only identified as such. Instead, a full assignment statement could be shown. In that case removing the lines and rectangles would leave a series of expressions which constitute in effect the program BLOKPAR. This illustrates that the drawing is in essence superfluous and serves only for clarification. However, the added clarification as a rule is fully worth its space. The purpose of using APL in the block diagram is not to eliminate the diagram, but to make it more complete, hence more satisfactory.

Control implementation

As a final implementation step a method for the control should be selected. Again a multitude of solutions present themselves. The division operation could be given its own control as is sketched in figure 4 and described in program 15.

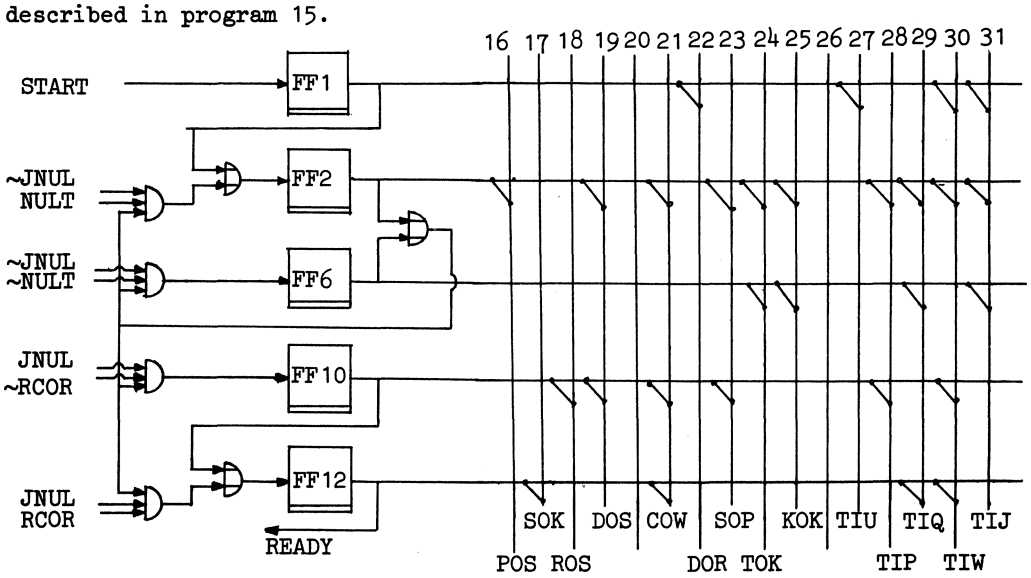


figure 4. Specialized control for division.

```
     ∇ SPECDIV
[1]    ⍝ PROLOGUE
[2]    P←(⍴DR)↑DT
[3]    Q←(⍴DR)↓DT
[4]    R←DR
[5]    ⍝ CALL SPECIALISED CONTROL
[6]    CONDIV
[7]    ⍝ EPILOGUE
[8]    RT←P
[9]    QT←Q
[10]   OVDL←OVLP
     ∇

     ∇ CONDIV
[1]    ⍝ START
[2]    FF←13⍴0
[3]    IFF← 0 1 ,11⍴0
[4]    →6
[5]   PROCEED:IFF←13⍴0
[6]    ⍝ BRANCH CONDITIONS
[7]    OR1←FF[2]∨FF[6]
[8]    IFF[2]←FF[1]∨OR1∧NULT∧~JNUL
[9]    IFF[6]←OR1∧(~NULT)∧~JNUL
[10]   IFF[10]←OR1∧JNUL∧~RCOR
[11]   IFF[12]←FF[10]∨OR1∧JNUL∧RCOR
[12]   ⍝ SET GATES
[13]   FF←IFF
[14]   POS←FF[2]
[15]   SOK←FF[12]
[16]   ROS←FF[10]
[17]   DOS←FF[2]∨FF[10]
[18]   COW←FF[2]∨FF[10]∨FF[12]
[19]   DOR←FF[1]
[20]   SOP←FF[2]∨FF[10]
[21]   TOK←FF[2]∨FF[6]
[22]   KOK←FF[2]∨FF[6]
[23]   TIU←FF[1]
[24]   TIP←FF[2]∨FF[10]
[25]   TIQ←FF[2]∨FF[6]∨FF[12]
[26]   TIW←FF[1]∨FF[2]∨FF[10]∨FF[12]
[27]   TIJ←FF[1]∨FF[2]∨FF[6]
[28]   ⍝ DATAFLOW ACTION
[29]   BLOKPAR
[30]   ⍝ END CONDITION
[31]   →(~FF[12])/PROCEED
     ∇
```

program 15. Specialized division control.

Another alternative is the use of microcoding. This is represented by the control equipment shown in figure 5 and represented by program 16.

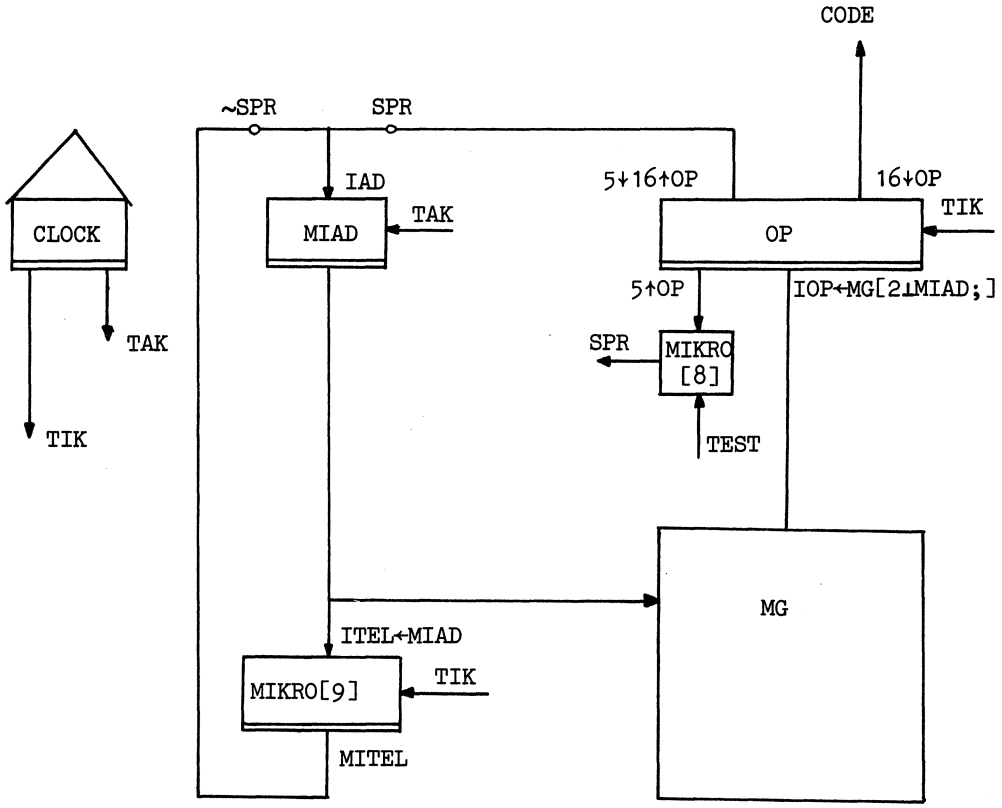


figure 5. Microcode control.

```
     ∇ MIKRO
[1]    ⍝ STORAGE FETCH
[2]    IOP←MG[2⊥MIAD;]
[3]    ITEL←MIAD
[4]    →(~TIK)/3
[5]    ⍝ SET GATES
[6]    OP←IOP
[7]    CODE
[8]    ⍝ DATAFLOW ACTION
[9]    BLOKPAR
[10]   ⍝ BRANCH CONDITIONS
[11]   TEST←1,JNUL,NULT,RCOR,OVDL,11⍴0
[12]   SPR←OP[0]=TEST[2⊥1↓5↑OP]
[13]   ⍝ NEXT ADDRESS
[14]   MITEL←(11⍴2)⊤1+2⊥ITEL
[15]   IAD←((~SPR)∧MITEL)∨SPR∧5↓16↑OP
[16]   →(~TAK)/11
[17]   MIAD←IAD
[18]   ⍝ END CONDITION
[19]   →(~∧/MIAD)/1
     ∇
```

```
          '01'[MG]
00000000000000000000000010000100110000
10001000000001001001010111001111000 0
10010000000000010000000000000000000 0
00001000000000100000000011000101000 0
10011000000001100000000000000000000 0
00000000000000000001101010000101000 0
10000111111111111010001000000011000 0
10000111111111111000000010000100011 00
00000000000000000000010000000000100 00
00001000000010010010000010101101000 0
10000111111111111000000000000000000 0
10000111111111111001000010000100010 10
10000111111111111000000010000100010 00
10000111111111111000000010000100010 10
10000111111111111001000010000100010 00
10000111111111111000000010000100011 10
```

test address gates

condition

program 16. Microcode control.

These solutions of course need much further development. For instance, in division several decisions are taken in succession, as can be seen in lines 9-11 of GATEDIV. The specialised control wastes no time on these decisions. In the primitive microcode an extra cycle is often required. As the designer improves his microcode, these delays certainly will be removed.

Microcode programs

Program 17 shows the implementation of several operations by means of the microprogram.

```
     ∇ MIKROPOS
[1]    ⍝ PROLOGUE
[2]    P←(ρA)ρ0
[3]    Q←(ρA)ρ0
[4]    R←A
[5]    ⍝ CALL MICROPROGRAM
[6]    MIAD←(11ρ2)⊤7
[7]    MIKRO
[8]    ⍝ EPILOGUE
[9]    B←P
     ∇

     ∇ MIKROMIN
[1]    ⍝ PROLOGUE
[2]    P←A
[3]    Q←(ρA)ρ0
[4]    R←B
[5]    ⍝ CALL MICROPROGRAM
[6]    MIAD←(11ρ2)⊤11
[7]    MIKRO
[8]    ⍝ EPILOGUE
[9]    B←P
     ∇
```

```
     ∇ MIKROMUL
[1]    ⍝ PROLOGUE
[2]    P←BW
[3]    Q←VR
[4]    R←VT
[5]    ⍝ CALL MICROPROGRAM.
[6]    MIAD←(11ρ2)⊤8
[7]    MIKRO
[8]    ⍝ EPILOGUE
[9]    PD←P,Q
     ∇

     ∇ MIKRODIV
[1]    ⍝ PROLOGUE
[2]    P←(ρDR)↑DT
[3]    Q←(ρDR)↓DT
[4]    R←DR
[5]    ⍝ CALL MICROPROGRAM
[6]    MIAD←11ρ0
[7]    MIKRO
[8]    ⍝ EPILOGUE
[9]    RT←P
[10]   QT←Q
[11]   OVDL←OVLP
     ∇
```

program 17. Calling programs for absolute value, subtraction, multiplication and division.

These programs are now reduced to a prologue, call and epilogue. The simulation of a division controlled by the microcode requires a fraction of a minute of terminal time. An extensive test therefore might become rather slow. However, it is not necessary to test the machine design in this in-

volved manner. Rather, the design step taken, the transformation of GATEDIV into the proper words of microstorage, should be verified. This is a rather simple step, which, if desired, could be automated.

Simultaneous sequences

The procedures described all have a single sequence. They represent the bulk of the design effort. However, simultaneous operation of several sequences can also be simulated. This has been done successfully by introducing a supervisory routine which calls on these routines in turn.

Translation to realisation form

Once an implementation is considered satisfactory, its description must be translated into a form suitable for the realisation. This translation involves the substitution of parameters and the explosion of vectors to individual elements. However, prior to these steps, it is advantageous to perform as many tasks as possible in the concise vector notation. For example, an initial matching of logical statements with components, identifying expressions which cannot be build, is best performed at this time.

The substitution of parameters results in the elimination of administrative statements. Thus the concise description of the ripple adder of program 6 is replaced by the 'unrolled' description of program 18, once it is known that the operands have 4 bits each.

```
     ∇ RIPPLE4
[1]    G←A∧B
[2]    T←A≠B
[3]    C← 0 0 0 0 ,CIN
[4]    C[3]←G[3]∨T[3]∧C[4]
[5]    C[2]←G[2]∨T[2]∧C[3]
[6]    C[1]←G[1]∨T[1]∧C[2]
[7]    C[0]←G[0]∨T[0]∧C[1]
[8]    COUT←C[0]
[9]    S←T≠1↓C
     ∇
```

program 18. 4-Bit ripple adder.

The latter program was in fact obtained by an APL program, written for this purpose (Bouwmeester [4]). However, to perform these actions effectively, procedures should be handled like data. Such a function, although anticipated, is not available in APL at present.

The explosion of vectors could similarly be performed as an APL program. In fact, all realisation details could also be described in APL. However, the absence of compilers and limitations in speed and size of the current implementations make APL less attractive for the production work, which the realisation requires. Although these disadvantages may be remedied in the future, they require at present the translation into another language. Such a translation including the explosion of vectors has been written in PL1 (Elsenaar [8]).

4. CONCLUSION

This paper illustrates the design of a computer with the aid of the terminal implementation of APL. The architecture of the part under consideration, the divide operation, was specified with this language and the terminal was used to verify that this was indeed the function intended. The various steps in the design of an implementation, starting with a constructive algorithm and ending with a data flow and microprogram, each again used an APL description. Furthermore, the transition from one step to the next could be made in a controlled manner using various means to assure the correctness of the algorithm. Finally, the ultimate implementation specification could be transformed to permit easy translation to a description required for the automation of the realisation design. The computer aided design process has been described only for this succession of steps. However, with the design available on a terminal, further improvements can be obtained by subjecting it to additional procedures, such as those dealing with timing, diagnostics and component selection.

ACKNOWLEDGEMENT

The material presented is part of a course on machine implementation presented at the department of Electrical Engineering of the Technische

Hogeschool Twente. The author is indebted to students and staff for many valuable suggestions. Besides those whose contributions have been mentioned in the references, drs. J. Al, B. van den Dolder, B.t.w., ir. A. van der Knaap and J.G. Raatgerink should be mentioned in particular.

Scalar Dyadic Functions

X+Y X plus Y
X-Y X minus Y
X×Y X times Y
X÷Y X divided by Y
X*Y X to the Y-th power
X⌈Y maximum of X and Y
X⌊Y minimum of X and Y
X|Y X-residue of Y (see Table)

X<Y X less than Y
X≤Y X less than or equal to Y
X=Y X equal to Y
X≥Y X greater than or equal to Y
X>Y X greater than Y
X≠Y X not equal to Y
X∧Y X and Y
X∨Y X or Y
X⍲Y not both X and Y (X nand Y)
X⍱Y neither X nor Y

} result is 1 if the relation holds, 0 if it does not

X	Y	X∧Y	X∨Y	X⍲Y	X⍱Y
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

Scalar Monadic Functions

+Y Y
-Y 0-Y
×Y sign of Y (-1, 0, 1)
÷Y reciprocal of Y
*Y e to the Y-th power
⌈Y ceiling of Y } (See Table)
⌊Y floor of Y }
|Y magnitude of Y
?Y a random integer from the vector ⍳Y
~Y not Y

Y	⌈Y	⌊Y
3.14	4	3
-3.14	-3	-4

	X|Y
X≠0	Y-X×⌊Y÷X
X=0	Y

Mixed Functions

X⍴Y Reshape Y to have dimension X
⍴Y Dimension of Y
X[Y] The elements of X at locations Y
X⍳Y First location of Y within vector X
⍳Y The first Y consecutive integers from Origin (0 or 1 as set by set origin command)
X∈Y Each element of X ∈ Y is 1 or 0 if the corresponding element of X is or is not some element of Y
X⊤Y Representation of Y in number system X
X⊥Y Value of the representation Y in number system X
X?Y X integers selected randomly without repetition from ⍳Y
X⍉Y Transpose by X of the coordinates of Y
⍉Y Ordinary transpose of Y
X,Y Y catenated to X
,Y Ravel of Y (make Y a vector)
X↑Y { If X positive take first X elements of Y / If X negative take last |X elements of Y
X↓Y { If X positive leave first X elements of Y / If X negative leave last |X elements of Y
X⌽Y X specified by Y
⍋X The indices of values of the vector X in sorted ascending order
⍒X The indices of values of the vector X in sorted descending order

∇ Program Definition Section
⍝ Comment

In the entries below ⍟ stands for "any scalar dyadic operator"

Generalized Reduction

i.e., insert the symbol ⍟ between each pair of elements of Y

⍟/Y The ⍟ reduction along the last dimension of Y
⍟⌿[Z]Y The ⍟ reduction along the Zth dimension of Y
⍟⌿Y The ⍟ reduction along the first dimension of Y

Compression and Expansion

X/Y X (logical) compressing along the last dimension of Y
X/[Z]Y X (logical) compressing along the Zth dimension of Y
X⌿Y X (logical) compressing along the first dimension of Y
X\Y X (logical) expanding along the last dimension of Y
X\[Z]Y X (logical) expanding along the Zth dimension of Y
X⍀Y X (logical) expanding along the first dimension of Y

Generalized Matrix Operations

X+.×Y Ordinary matrix product of X and Y
X⍟.⍟Y Generalized inner product of X and Y
X∘.⍟Y Generalized outer product of X and Y

All scalar functions are extended to operate element-by-element on dimensioned operands; i.e., vectors, matrixes, and higher-dimensional arrays.

A scalar or one-component vector may be used as one argument of a scalar dyadic function and will be extended to conform to the dimension of the other argument.

Special Symbols

( ) Parentheses. Expressions may be of any complexity and are executed from right to left except as indicated by parentheses.

→X Branch to X, where X is a scalar or vector. If X is an empty vector, go to the next line in sequence. If X is not in the range of statement numbers in the function, leave the function.

'XYZ' The literal characters XYZ.

Appendix A. Subset of APL used in this paper.
(from APL reference data IBM world trade)

LITERATURE

[1] Blaauw, G.A. (1952), Application of Selenium Rectifiers as Switching Devices in the Mark IV Calculator, Ph.D. Thesis, Harvard University, Cambridge, Massachusetts.

[2] Blaauw, G.A. (1954), De A.R.R.A, In: Nederlands Rekenmachine Genootschap 1959-1969, Nederlands Rekenmachine Genootschap, Amsterdam.

[3] Blaauw, G.A. (1970), Hardware Requirements for the Fourth Generation, In: Fourth Generation Computers: User Requirements and Transition, Prentice Hall, Englewood Cliffs, N.J.

[4] Bouwmeester, A.J.H. en A.T.F. Grooters (1971), Het elimineren met behulp van APL van niet of moeilijk bouwbare uitdrukkingen van een uitrustingsbeschrijving in APL, Verslag Baccalaureaatsopdracht, T.H. Twente, Enschede.

[5] Breed, L.M. and R.H. Lathwell (1968), The implementation of APL/360, In: Interactive Systems for Applied Mathematics, Academic Press, New York.

[6] Du Croix, A.J. (1970), Equivalentie van uitrustingsalgorithmen, Verslag Doctoraalopdracht, T.H. Twente, Enschede.

[7] Duijvestijn, A.J.W. en W.A. Vervoort (1971), Correctheid van algorithmen, Memorandum no. 11, Onderafdeling der Toegepaste Wiskunde, T.H. Twente, Enschede.

[8] Elsenaar, J.G. (1970), Een studie van de vertaling van een uitrustingsbeschrijving in APL naar Intern Formaat, Verslag Doctoraalopdracht, T.H. Twente, Enschede.

[9] Falkhoff, A.D., K.E. Iverson and E.H. Sussenguth (1964), A formal description of SYSTEM/360, IBM System Journal, vol. 3, no. 3, pg. 198-263.

[10] Frankel, S.P. (1957), Logical Design of a general purpose computer (LGP-30), IRE Transactions on Electronic Computers vol. EC-6, no. 1, pag. 5-14.

[11] Friedman, T.D. and S.C. Yang (1969), Methods Used in an Automatic Logic Design Generator, IEEE Transactions on Computers, vol. C-18, no. 7, pg. 593-613.

[12] Iverson, K.E. (1962), A Programming Language, Wiley, New York.

[13] McCracken, D.D. (1970), Whither APL? Datamation, vol. 16, no. 11, pg. 53-55.

[14] Pakin, S. (1968), APL/360 Reference Manual, Science Research Associates, Chicago.

[15] Rhyne, V.T. (1971), A Simple Postcorrection for Nonrestoring Division, IEEE Transactions on Computers, vol. C-20, no. 2, pg. 213-214.

[16] Stabler, E.P. (1970), System Description Languages, IEEE Transactions on Computers, vol. C-19, no. 12, pg. 1160-1173.

[17] Wijngaarden, A. van (1964), Rekenen in Nederland, In: NRMG 1959-1964, Nederlands Rekenmachine Genootschap, Amsterdam.

ADDRESSING PRIMITIVES AND THEIR USE IN HIGHER LEVEL LANGUAGES

A.J.W. Duijvestijn, G.A.M. Kamsteeg-Kemper, J.P. Schaap-Kruseman *)

STARTING POINTS OF THE ADDRESSING SYSTEM

We consider a set of information units that can be accessed by one co-ordinate: an integer called *"displacement"* DPl. The set plus the co-ordinate is a linear addressing space. It is called a *"segment"*. A segment may be considered as a one dimensional vector. The vector elements are the smallest addressable units. These elements may be an instruction, a constant or a data field etc. We distinguish between text- and data segments. A text segment contains only instructions or constants. The representation of a text segment never changes due to the execution of an instruction or the reference to a constant. A data segment contains information that changes by assigning new values during the execution of instructions of a text segment. A *"segment group"* is a set of information units plus a co-ordinate consisting of a two-tuple (SN,DPL). Every information unit can be accessed by the co-ordinate (SN,DPL), where SN is a segment number and DPL the displacement in the segment with number SN. SN and DPL are both integers. The co-ordinate is the address of the information unit. It is called a *"local co-ordinate"* or a *"local address"*. The integer SN is called a *"local segment number"*. A segment group is therefore a set of segments. The segments are numbered 1,2,3,... .

We can also consider a set of segment groups. Information units then can be found by a 3-tuple (SGN,SN,DPL), where SGN is an integer: segment group number, SN is the segment number in the segment group and DPL the displacement in the segment.

The set of information units (say S) and the 3-tuple co-ordinate is called the space of segment groups which we denote by R_1. To the set of information units S we can add a new co-ordinate consisting of a 2-tuple: (NSN,DPL), where NSN is a new segment number. This space is denoted by R_2 and is called the total space of segments. This space is obtained by con-

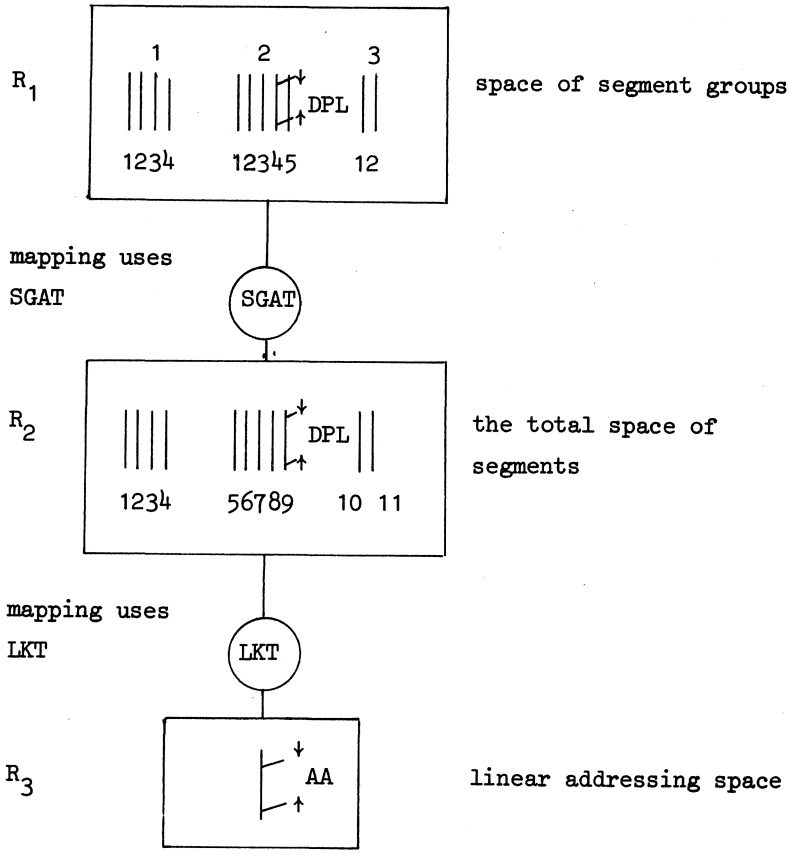
*) Technological University, Twente

sidering the total set of segments of all segment groups. The segments are numbered from 1, 2, ..., total number of segments. The co-ordinate (NSN,DPL) is called an *"interlocal number"* or *"interlocal address"*, the integer NSN *"interlocal segment number"*.

The linear addressing space R_3 is a set of information units S^* plus one co-ordinate: the absolute address AA. The space R_3 could be considered as a segment.

We assume $S \subset S^*$. The set S is a proper subset of S^*.

The space R_1 can be mapped <u>onto</u> R_2 and R_2 can be mapped <u>into</u> R_3.



Finally R_3 is mapped into the hardware addressing space.
The mapping from R_1 onto R_2 uses a Segment Group Addressing Table: SGAT.
The mapping from R_2 into R_3 uses a LinK Table that is denoted by LKT. These mappings are described in [1] and [2].

REQUIREMENTS OF THE ADDRESSING SYSTEM

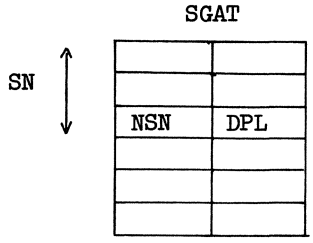
The requirements of the addressing system described in [1] and [2] are the following:

1. Independent translation of an arbitrary source language to a segment group of a common object language. Different segment groups of object language must be linked together without retranslation. References between segment groups must be possible.
2. After translation into objectlanguage no changes are allowed in text segments. Prior to execution a data segment does not contain initial values. Text- end data segments can be distinguished.
3. The mapping of segment groups on the hardware addressing space must be dynamically changeable. This means dynamic relocation must be possible.
4. The addressing space must be dynamically extendable.
5. There must be an inherent read and write protection of the information on the basis of the definition of the addressing space. Only information of its own addressing space is accessible. "Outside" the addressing space is not defined and therefore does not exist. The mapping tables do not belong to the addressing space.

FORMAT OF SGAT AND LKT

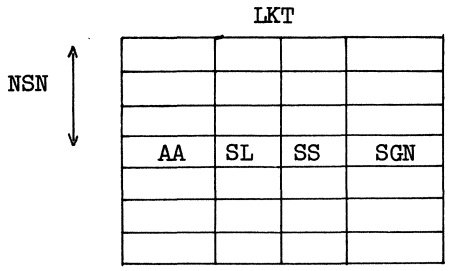
In order to give an idea of the necessary information of the mapping tables we give a description of the format of SGAT and LKT [2].

SGAT is an array of a structure consisting of two integers nl. new segment number (NSN) and displacement (DPL).



The index of the array SGAT is the segment number SN.

The linktable LKT is an array of a structure consisting of an absolute address (AA), a segment length (SL), an indication (SS) denoting whether the segment is a data segment or text segment and finally a segment number (SGN) giving access to an appropriate SGAT.



The address calculation of an information unit within a segment group with table SGAT, having as co-ordinate the 2-tuple (SN,DPL1) is in principle:

```
nsn:= NSN of SGAT[SN]; absolute address:= AA of LKT[nsn] +
                                          DPL of SGAT[SN] + DPL1;
                       if absolute address >(AA of LKT[nsn] +
                                          SL of LKT[nsn])
                       then error;
```

PRIMITIVE ADDRESSING FUNCTIONS

We now assume a real or virtual machine that has the availability of one SGAT table and a LKT table with the following addressing functions (instructions of the machine).

1. Local contents (n,p) denotes an information unit in a segment group where n is the segment number and p the displacement. In this case the SGAT and LKT table both are used.
   The notation x:= local contents (n,p) means the information unit denoted by local contents (n,p) is assigned to x.

   If we write local contents (n,p):=x we mean that the contents of the location in the segment group with co-ordinate (n,p) gets the value of x. The mapping uses the table SGAT which gives a new segment number NSN. With the aid of the table LKT an absolute address is obtained.

2. The function: interlocal contents (nn,p) denotes an information unit in the total set of segment groups with the interlocal co-ordinate new segment number nn and displacement p. The mapping into the absolute addressing space uses only the table LKT. The absolute address is obtained from (AA of LKT[nn]) + p.

3. The function: save SGAT (NSN,DPL,from,to) stores (saves) the array elements SGAT[from], SGAT[from+1], ..., SGAT[to] in the interlocal address with co-ordinates (NSN,DPL), (NSN,DPL+1), ..., (NSN,DPL+to -from+1).

4. The function: restore SGAT (NSN,DPL,from,to) stores the information units with interlocal co-ordinates (NSN,DPL), (NSN,DPL+1), ..., (NSN,DPL+to-from+1) in the array elements SGAT[from], SGAT[from+1], ..., SGAT[to].

5. The function: fill SGAT (x,y) performs the operation
   NSN of SGAT[x]:= y.

6. The function: get data segment (space,NSN) asks the operating system for a data segment with segmentlength space. The operating system returns an interlocal segment number NSN.

7. The function: return data segment (space,NSN) returns a segment to the operating system. The segment has an interlocal number NSN and a segmentlength space.

Remark: It is not necessary to assume that the functions get- and return data segment are interpreted by the operating system. It is equally well possible to achieve this by a set of allocation routines.

USE OF THE ADDRESSING FUNCTIONS IN ALGOL-60

To each block in ALGOL-60 we assign a data segment. As soon as a block is entered a data segment is created. We assume that this is done by the interpreter instruction Create BLock (CBL). We reserve space in this data segment for all variables that are declared in the block. In this way a number of data segments may have been created. As soon as a block is left a data segment is returned to the operating system by the interpreter instruction leaVe BLock (VBL).

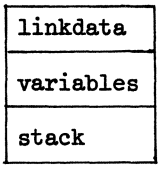
The data segment belonging to the block of which the statements are executed and the data segments belonging to its embracing blocks form a segment group. This segment group has its own SGAT.

Each simple variable can be accessed by a local address (SN,DPL). An array element can be accessed by means of the descriptor of the array, which has an address (SN,DPL).

In each data segment we assume that a stack is implemented for storing intermediate results. The stack is denoted by the array S with index AP (Accumulator Pointer [3,4,5,6]).
Therefore we assume that the machine which interpretes the addressing functions has two (real or virtual) registers called: AP and CBN.
The register CBN contains the local segment number belonging to the block (or procedure) of which statements are executed.
It is called the Current Block Number.

The register AP contains an integer such that S[AP] is the top of the stack S. It has the local address (CBN,AP).

The lay-out of the data segment is therefore:



In particular the following format is used:

SAVE AP	SAVE Accumulator Pointer [3,4,5,6]
SAVE WP	SAVE Working space Pointer [3,4,5,6]
SAVE NN	SAVE interlocal address NN of this data segment
SAVE SPACE	SAVE SPACE of this data segment
SAVE NNAR	SAVE interlocal address NN of an ARray data segment that contains all arrays of this data segment
SAVE SPACEAR	SAVE SPACE of the ARray data segment
SAVE CBN	SAVE Current Block Number. This is the local number of this data segment
SGAT	
VARIABLES	
STACK	

The two interpreter instructions: create block and leave block are declared as follows:

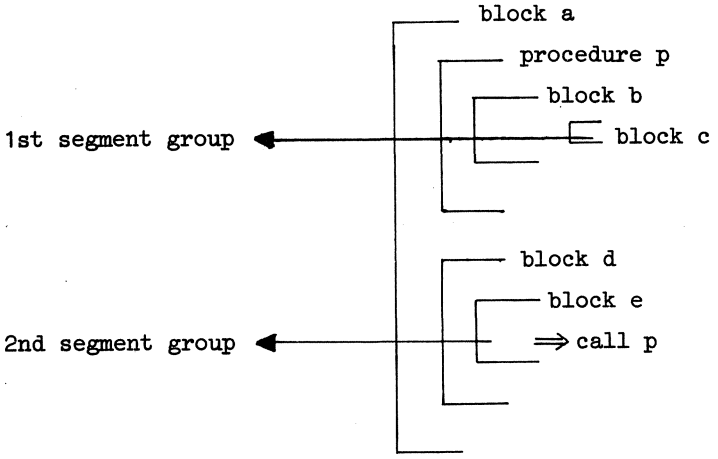
```
procedure CBL(space segment,WP); comment Create BLock; integer space segment, WP;
    begin integer i,nn;

        local contents(CBN,SAVEAP):= AP; comment save AP of the data
                    segment that is left;

        get data segment(space segment,nn); comment a data segment is
                    created of length: space segment, the result is
                    an interlocal number nn;

        CBN:= CBN+1; comment increase current block number: the local
                    number of the new data segment;

        fill SGAT(CBN,nn); comment SGAT[CBN]:= nn. By means of local
                    contents(CBN,DPL) we can access information in
                    the created data segment;

        local contents(CBN,SAVE WP):= WP; comment the working space
                    pointer (WP) is the displacement in the segment
                    such that (CBL,WP) is the local address of the
                    first stack location;

        local contents(CBN,SAVE NN):= nn;
        local contents(CBN,SAVE SPACE):= space segment;
        local contents(CBN,SAVE SPACE AR):= 0; comment the necessary
                    array data segment has not yet been created. This
                    is done by the instruction Reserve ARray. There-
                    fore we initialize SAVE SPACE AR with 0;

        local contents(CBN,SAVE CBN):= CBN;
```

```
            comment the link information SAVE AP,...,SAVE CBN has a
                    fixed length. Let the displacement pSGAT be such
                    that the link information occupies the locations
                    with co-ordinates (CBN,0), ..., (CBN,pSGAT-1).
                    The length of SGAT depends on CBN. The table SGAT
                    occupies the locations with co-ordinates (CBN,pSGAT),
                    ..., (CBN,pSGAT+CBN). The displacement of the first
                    declaration is therefore pSGAT+CBN+1;
            for i:= pSGAT+CBN+1 step 1 until WP-1 do
                        local contents(CBN,i):= unused;

            comment the contents of all declared variables are initial-
                    ized with the value unused. This gives the possibil-
                    ity to detect that a variable is used without assign-
                    ing a value different then unused;
            AP:= WP
    end;

procedure VBL; comment leaVe BLock;

    begin if local contents(CBN,SAVE SPACE AR) ≠ 0

        then return data segment(local contents(CBN,SAVE SPACE AR),
                                 local contents(CBN,SAVE NNAR));

            comment if the block that is left contains arrays then the
                    array data segment is returned to the operating
                    system;

            return data segment (local contents(CBN,SAVE SPACE),
                                 local contents(CBN,SAVE NN));

            comment the data segment belonging to the block that is left
                    is returned to the operating system;

            CBN:= CBN-1; comment the embracing block now acts as current
                                 block;

            AP:= local contents(CBN,SAVE AP); comment AP is restored;
    end;
```

PROCEDURES

A procedure is considered as a block. If a procedure is entered a data segment is created. The calling procedure is left temporarily. The original segment group is left and a new segment group is entered. The new segment group consists of data segments belonging to the entered procedure and its embracing blocks.

EXAMPLE



The data segments of block a, block d and block e form a segment group. The data segments of block a, procedure p, block b and block c form another segment group. Notice that two segment groups may have common segments. The addressing functions can be used in this case. In fact we deal with a special case, because only one segment group can be active at a time. The addressing mechanism described in [1] and [2] can also access information in different segment groups.

The format of a procedure data segment is as follows:

link information
parameters and declarations
stack

The detailed format is described below:

SAVE AP	Accumulator Pointer of the stack of this segment
SAVE WP	Working space Pointer
SAVE NN	Interlocal block number of this segment
SAVE SPACE	Length of this segment
SAVE NNAR	Interlocal number of array segment belonging to the outermost block of this procedure
SAVE SPACEAR	Necessary space of array segment
SAVE NN CALLER	Interlocal number of calling segment block or procedure
SAVE RETURN ADDRESS	Return address in text segment after completion of the procedure
SAVE CBN	Current local block number of this segment
SGAT	
result procedure	
parameters	
declarations	
STACK	

Because we have chosen for saving SGAT in the segment of the calling procedure or block it is necessary to provide space for this link information in both a block segment and a procedure segment.

The creation of a procedure data segment is done by the interpreter instruction Create PRocedure (CPR).
The declaration of this instruction is:

```
procedure CPR(BNp,space segment,WP); integer BNp, space segment, WP;

    begin integer i,nn,nn caller;

        local contents(CBN,SAVE AP):= AP;
        nn caller:= local contents(CBN,SAVE NN);
            comment keep the interlocal number of the calling data seg-
                                                                ment;

        SAVE SGAT(nn caller,pSGAT,1,CBN+1);
            comment the SGAT table is saved in the calling data seg-
                    ment. The number of elements that have to be saved
                    is CBN+1. The first location of the saving space
                    has the interlocal co-ordinate (nn caller,pSGAT);

        get data segment(space segment,nn);
            comment create data segment with space segment information
                    elements. The result is an interlocal number nn;

        CBN:= BNp;

        fill SGAT(CBN,nn); comment the new SGAT is current;

        local contents(CBN,SAVE WP):= WP;

        local contents(CBN,SAVE NN):= nn;

        local contents(CBN,SAVE SPACE):= space segment;

        local contents(CBN,SAVE SPACEAR):= 0;

        local contents(CBN,SAVE NN CALLER):= nn caller;

        local contents(CBN,SAVE CBN):= CBN;
```

```
            comment the link information including SGAT use the loca-
                    tions with co-ordinates (CBN,0), ...,
                    ..., (CBN,pSGAT-1), (CBN,pSGAT), ...,
                    ..., (CBN,pSGAT+CBN). The first available location
                    for result procedure, parameters and declarations
                    has the co-ordinate (CBN,pSGAT+CBN+1);
            for i:= pSGAT+CBN+1 step 1 until WP-1 do

                    local contents(CBN,i):= unused;

            comment the space for result, parameters and declaration
                    are marked unused. For parameters and results this
                    is not necessary. It is done for reasons of ease;

            AP:= WP

    end;
```

FETCH AND STORE INTERPRETER INSTRUCTIONS

In case we deal with non-formal variables we use basically three instructions n.l. FeTCh, ADdRess and STOre which are defined as follows:

```
procedure FTC(n,p); comment fetch the variable with local co-ordinate (n,p)
                    and put it on top of the STACK: S;

            integer n,p;

    begin if local contents(n,p)= unused then goto error;

          S[AP]:= local contents(n,p);

          AP:= AP+1

    end;
```

```
procedure ADR(n,p); comment fetch the address of the variable with local
                        co-ordinate (n,p) and put it on top of the STACK: S;
              integer n,p;
    begin
          S[AP]:= n; S[AP+1]:= p; AP:= AP+2;
               comment we could also store n and p in S[AP];
    end;

procedure STO;
    begin AP:= AP-3; local contents(S[AP], S[AP+1]):= S[AP+2] end;
```

In case we deal with a variable that is formal in a procedure the instructions FTC and ADR cannot be used. We assume that before calling the procedure the actual parameters have been characterized. We only treat the characterisation of a variable and an expression. The characterisation of a variable is a structure with three integers n.l. code, nn, pp. See also [5].

The first information unit of this structure has the co-ordinate (CBN,pk). By code of local contents (CBN,pk) we mean that part of the structure that is denoted by code. The characterisation is done by the interpreter instructions CVA and CEX respectively.

```
procedure CVA(pk,nv,pv); comment Characterize VAriable;
              integer pk,nv,pv;
    begin comment (nv,pv) is the co-ordinate of the variable, while pk is
                        the displacement of the characterisation;
          code of local contents(CBN,pk):= code var;
          nn of local contents(CBN,pk):= interlocal contents(local contents
                        (CBN,SAVE NN CALLER), pSGAT+nv);
```

```
            comment the data segment of the procedure is current. The
                    SGAT table necessary for the interlocal number of
                    the data segment that contains the variable, has
                    been saved in the calling data segment;

        pp of local contents(CBN,pk):= pv

    end;
```

In case of a characterisation of an expression we assume a structure with three integers: code, nn, address expression.

The interpreter instruction CEX is defined as follows:

```
procedure CEX(pk,address expr); comment Characterise EXpression;

            integer pk, address expr;

    begin comment pk is the displacement of the characterisation. address
                  expression is the displacement in the program seg-
                  ment, where the first instruction can be found
                  necessary for evaluating the expression that is
                  substituted for the formal parameter;

        code of local contents(CBN,pk):= code expression;

        nn of local contents(CBN,pk):= local contents(CBN,SAVE NN CALLER);

        address expression of local contents(CBN,pk):= address expr

    end;
```

Finally the calling of the procedure takes place by means of the interpreter instruction CALl: CAL. This happens when the data segment of the procedure has been created and the characterisations have been filled.

The declaration of the instruction CAL is as follows:

```
procedure CAL(start address procedure, return address); integer start
                    address procedure, return address;

    begin local contents(CBN,SAVE RETURN ADDRESS):= return address;
          IC:= start address procedure; comment IC:= instruction counter;

    end;
```

EXAMPLE

The next ALGOL-60 program has been translated in interpreter instructions.

```
    begin real z1;

          real procedure b(x,y); real x,y;

               b:= x+y;

               z1:= 7;

               begin real z2;

                     z2:= 5;

                     z1:= b (z2,z1+z1*z2)

               end

    end
```

The interpreter instructions are:

```
          CBL    space segment= ..., WP= ...

          ADR    z1
          FTC    7
          STØ

          CBL    space segment= ..., WP= ...

          ADR    z2
          FTC    5
          STØ

          ADR    z1
          CPR    BN=2, space segment= ..., WP= ...
          CVA    p characterisation variable, nv z2= 2, pv z2= 1
          CEX    p characterisation expression, L1
          CAL    L2, L3
L3        STO
          VBL
          VBL
READY     STOP

L2        ADR    b
          FFV    x      This instruction will be treated later
          FFV    y
          RTN           return procedure

L1        FTC    z1
          FTC    z1
          FTC    z2
          MPY           multiply
          ADD           add
          XTR           return parameter subroutine
```

From the theory of language transformations we know that the fetch instruction FTC cannot be used if the variable is a formal variable of a procedure. In that case we need an instruction Fetch Formal Variable: FFV.

Using the addressing functions we declare the instruction FFV as follows:

```
procedure FFV(n,p, return address); integer n,p, return address;

    begin comment the co-ordinate (n,p) is the local co-ordinate of the
                  characterisation of the parameter. The return
                  address is only necessary in case the actual para-
                  meter is an expression. In that case return address
                  is needed after evaluation of the parameter expres-
                  sion;

        integer nn;

        if code of local contents (n,p)= codevar

        then begin comment the actual parameter is a variable;

                  S[AP]:= interlocal contents(nn of local contents
                          (n,p), pp of local contents(n,p));

                  if S[AP]:= unused then goto error;

                  AP:= AP+1

             end

        else if code of local contents(n,p)= code expression

             then begin comment actual parameter is an expression. The
                       current data segment is left temporarily. AP and
                       SGAT are saved in this data segment. The SGAT and
                       AP of the segment group in which the parameter sub-
                       routine must be evaluated are restored;
```

```
            nn:= local contents(CBN,SAVE NN);

            IC:= address expression of local contents (n,p);

            local contents(CBN,SAVE AP):= AP;

            save SGAT(nn, pSGAT,1,CBN+1);

            CBN:= interlocal contents(nn of local contents(n,p),
                 SAVE CBN);

                 comment the data segment necessary for evalua-
                         tion of the expression is made current;

            restore SGAT(nn of local contents(n,p), pSGAT,1,CBN+1);

            AP:= local contents(CBN,SAVE AP); comment restore AP;

            S[AP]:= return address;

            S[AP+1]:= nn; comment link information for returning
                                  from parameter routine;

            AP:= AP+2

        end

end;
```

For the sake of completeness we consider also the instructions return procedure (RTN) and the instruction XTR which is used for returning from the parameter subroutine.

These instructions are declared as follows:

```
procedure RTN;

    begin integer result procedure, nn caller;

          result procedure:= local contents(CBN,pSGAT+CBN+1);

          IC:= local contents(CBN,SAVE RETURN ADDRESS);

          nn caller:= local contents(CBN,SAVE NN CALLER);

          if local contents(CBN,SAVE SPACEAR) ≠ 0

          then return data segment(local contents(CBN,SAVE SPACEAR),
                                   local contents(CBN,SAVE NN AR));

               comment if necessary return array data segment;

          return data segment(local contents(CBN,SAVE SPACE),
                                   local contents(CBN,SAVE NN));

               comment return the procedure data segment;

          CBN:= interlocal contents(nn caller,SAVE CBN);

          restore SGAT(nn caller,pSGAT,1,CBN+1);

               comment the calling data segment is current again;

          AP:= local contents(CBN,SAVE AP); comment restore AP;

          S[AP]:= result procedure;

          AP:= AP+1

    end;
```

Finally the return instruction from the parameter subroutine is as follows:

```
procedure XTR;

    begin integer result expression, nn caller;

          result expression:= S[AP-1];

          nn caller:= S[AP-2]; comment nn of data segment that temporarily
                                       was left for the computation of the
                                       expression;

          IC:= S[AP-3];

          local contents(CBN,SAVE AP):= AP-3;

               comment free link information by lowering AP and save in
                       SAVE AP;

          CBN:= interlocal contents(nn caller,SAVE CBN);

          restore SGAT(nn caller,pSGAT,1,CBN+1);

               comment the original, temporarily left data segment is
                       current again;

          AP:= local contents(CBN,SAVE AP); comment restore AP;

          S[AP]:= result expression;

          AP:= AP+1

    end;
```

We do not treat all other interpreter instructions here. The instructions create data segment formal procedure,

formal procedure call,
characterisation procedure,
reserve array
and get array element

have been described in [1].

A complete semantic description of all interpreter instructions is described in ALGOL-68 and in a new language: VGP language [7] and [8].

CONCLUSION

By means of the addressing functions one obtains a design of the interpreter that is clean and easy to understand. Therefore the correctness is better mastered.

The mapping of the functions on the available hardware is still free. The functions are abstractions that serve as a model in terms of which the interpreter is easy to describe.

ACKNOWLEDGEMENTS

The discussions with W.A. Vervoort are acknowledged.

REFERENCES

[1] Duijvestijn, A.J.W. (1970). Language transformations (in Dutch). Lecture notes course 1970-1971. Technological University Twente, Enschede, The Netherlands.

[2] Vervoort, W.A. (1970). An addressing system (in Dutch). Thesis-work for bachelors degree of electrical engineering. Technological University Twente, Enschede, The Netherlands.

[3] Dijkstra, E.W. (1961). An ALGOL-60 translator for the X1. ALGOL Bull. Suppl. N010.

[4] Dijkstra, E.W. (1961). Making a translator for ALGOL-60. A.P.I.C. Bull. 7, pp. 3-11.

[5] Medema, P. Object program ALGOL-60. Report N645. Philips Computer Centre, Eindhoven. March 62 - January 63.

[6] Medema, P., Maris, L.A. (1965). Documentation ALGOL-60. Translator PASCAL. Chapter 5: Object phase, Reportno: CD 65/55/318R. N.V. Philips-Electrologica.

[7] Kamsteeg-Kemper, G.A.M. (1969). Subset ALGOL-translator with dynamic creation of data segments. Description in ALGOL-68. Documentation number ALG.VGP 007. Translator group of the Technological University Twente, Enschede, The Netherlands.

[8] Schaap-Kruseman, J.P. (1971). THT ALGOL-60 translator with dynamic creation of data segments. Description in VGP language. Documentation number ALG.VGP 008. Translator group of the Technological University Twente, Enschede, The Netherlands.

ON A METHODOLOGY OF DESIGN

E.W. Dijkstra *)

On an occasion like this it is very tempting to look backwards, to play the modern historian, to give a survey of what has happened over the last 25 years, and to interpret this history in terms of trends and the like. This is so tempting that I shall try to resist the temptation. Instead, I would like to pose a question and to speculate about its answer. The question is roughly "Can we expect a methodology of design to emerge in, say, the next ten years?".

Let me first explain why I pose this question in my capacity as a programmer. There is a very primitive conception of the programmer's task, in which the programs produced by him are regarded as his final product. It is that conception which has led to the erroneous idea that a programmer's productivity can be measured meaningfully in terms of number of lines of code produced per month, a yardstick which when accepted, is guaranteed to promote the production of insipid code. A sounder attitude regards as the programmer's final product the class of computations that may be evoked by his program: the quality of his program will depend among many other things on the effectiveness with which these computations will establish their desired net effect. In this point of view a programmer designs a class of computations, a class of happenings in time, and the program itself then emerges as a kind of generic description of these happenings, as the mechanism that evokes them. I regard programming as one of the more creative branches of applied mathematics and the view of a program as an abstract mechanism makes it perfectly clear that designing will play an essential role in the activity of programming. Conversely, the view of a program as an abstract mechanism suggests us that a good understanding of the programming activity will teach us something relevant about the design

---

*) Technological University, Eindhoven

process in general. In actual fact this hope is one of the major reasons for my being interested in the task of programming for digital automata, digital automata which confront us with a unique combination of basic simplicity and ultimate sophistication. On the one hand programming is very, very simple; on the other hand processing units are now so fast and stores are now so huge that what can be built on top of this simple basis has outgrown its original level of triviality by several orders of magnitude. And it is this fascinating contrast that seems to make programming the proving ground par excellence for anyone interested in the design process of non-trivial mechanisms.

It is this combination of basic simplicity and ultimate sophistication which presents the programming task as a unique and rather formidable intellectual challenge. I consider it as a great gain that by now scope and size of that challenge have been recognized. Looking backwards we can only regret that in the past the difficulty of the programming task has been so grossly underestimated and that apparently we first needed the so-called "software failure" to drive home the message that in any non-trivial programming task it tends to be very difficult to keep one's construction intellectually manageable.

As soon as programming emerges as a battle against unmastered complexity, it is quite natural that one turns to that mental discipline whose main purpose has been since centuries to apply effective structuring to otherwise unmastered complexity. That mental discipline is more or less familiar to all of us, it is called Mathematics. If we take the existence of the impressive body of Mathematics as the experimental evidence for the opinion that for the human mind the mathematical method is indeed the most effective way to come to grips with complexity, we have no choice any longer: we should reshape our field of programming in such a way that the mathematician's methods become equally applicable to our programming problems, for there are no other means. It is my personal hope and expectation that in the years to come programming will become more and more an activity of mathematical nature. I am tempted to add that a development in that direction can already be observed, but I refrain from doing so: such a statement has too much the flavour of wishful thinking and besides that, such a statement could easily be an overestimation of the relative impor-

tance of the rather limited part of the field that lies within my mental horizon.

I have used vague terms like "the mathematician's method" and "an activity of mathematical nature"; I did so on purpose and let me try to explain why.

In one meaning of the word we identify Mathematics with the body of mathematical knowledge, with the subject matter dealt with in mathematical theses, articles appearing in mathematical journals etc. Im am not ashamed of admitting that most of it never passes my eyes; I also have a feeling that most of it - although of course one never knows - is hardly of any relevance for the programming task. If we identify Mathematics with the subject matter with which mathematicians have occupied themselves over the last centuries, it is indeed hard to see how mathematics could be highly relevant to the art of programming. In view of the programming problems facing us now, we can only regretfully observe that preceding generations of mathematicians have neglected a now important field. There is of course no point in blaming our fathers and grandfathers for this neglect. In their time, prior to the advent of the actual computing equipment, there was very little incentive: for lack of machines programming was no problem.

In a second meaning of the word we identify Mathematics with a human activity, with patterns of reasoning, with methods of exploiting our powers of abstraction, with traditions of mixing rigour with vagueness, with ways of finding solutions. It is in this second meaning that I judge mathematics as highly relevant for the programming task.

It is perhaps worth noting that, at least at present, the second interpretation of mathematics does not seem to be the predominant one at the Universities. In the academic curricula the fruits of research are transmitted very explicitly, how one does do research, however, is taught only very implicitly, at most as a kind of by-product. We teach solutions, we teach hardly how to solve. At first sight this is amazing, taking into account that one of the assumptions underlying the University is that we can educate researchers! But there are explanations.

One observation is that many mathematicians of the current generation - in Euler's time it may have been different - seem to worry very little

about problems of methodology; stronger, they resist it and are shocked by the mere suggestion that, say, a methodology of mathematical invention could exist. Although we profess to be yearning for knowledge, insight and understanding, we are fascinated by the unknown and many a creative mathematician is fascinated by his own inventive ability thanks to the fact that he does not know how he invents. He enjoys his share in the spark of genius, untarnished by any understanding of the inventive process. We just like mysteries.

Secondly, with the growth of mathematical literature, particularly the publications of the type "Lemma, Proof, Lemma, Proof etc." mathematics has very much acquired the image of a "hard" science. It is regarded by many as the prototype of a "hard" science. But the result is that the mathematician tends to feel himself superior, that he looks disdainfully down upon all the "soft" sciences surrounding him. As a result a serious research effort into discovery and development of a methodology of mathematical invention would have a hard fight for academic respectability. And we all know that the pressures for academic respectability are very strong. It takes a respectable scientist, supported by fame, to embark upon it. Polya did it with his "Mathematics and the Art of Plausible Reasoning" and I admire his courage.

The final reason why we teach so little about problem solving, however, is that we knew so little about it, that we did not know how to do it. But I honestly believe that in the last fifteen years the scenery has changed. Polya has written the book I mentioned, Koestler has written a book of 600 pages called "The Act of Creation", Simon delivers at the IFIP Congress 1971 a talk on "The Theory of Problem Solving", just to mention a few examples. And there is a fair chance that this development will influence our teaching of mathematics. I think it will.

After this digression we return to our original question, can we expect a methodology of design to emerge? In designing one designs a "thing" that does "something". Over the last decades the most complicated things designed to do something have been programs; on account of their abstract nature we can regard programs as the "purest" mechanisms we can think of and if we can find some sort of answer to the specific question "What about a programming methodology?", that answer seems relevant with respect

to our original question.

It is my impression that there is a point in discussing programming methodology separate from problem solving as it is treated usually. Most of the literature about problem solving that I have seen deals with how to hit an unexpected but simple solution - simple of course once you have found it. In the case of programming this simplicity of the final solution is very often an illusion: programs, even the best programs we can think of for a given task, are often essentially very large and complicated. And by structure they are more akin to complete mathematical theories than to an ingenious solution to some sort of combinatorial puzzle. In other words, programming tasks seem to be of a different size.

From a programming methodology we require two main things. It should assist us in making better programs - i.e. we have desires regarding the final product - it should also assist us in the process of composition of the design - i.e. even if we have established what kind of programs we should like to design, we would like to discover ways leading to such a design. As it is hard to talk about strategies that might assist you in reaching your goal without having a clear picture of the goal itself, we shall deal with the first question first: what kind of programs should we like to make? If we talk about "better programs", what standards do we apply in judging their quality?

I have raised this question urgently and repeatedly in the first half of the sixties but at that time it turned out to be impossible to reach even the illusion of a consensus of opinion and the question was discarded by the attitude that it was all a matter of taste. The common experience of the next five years has certainly changed the situation. This common experience with large and vital programs was very often disastrously bad, and as a result of this sobering experience more and more people agree that requirement number one is not only that a program should be correct, but that its correctness can be established beyond reasonable doubts. An analysis of the possible ways for increasing the confidence level of programs has shown that for that purpose, program testing must be regarded as insufficient: program testing can be used very effectively to show the presence of bugs, but never to show their absence. Proving program correctness remained as the only sufficiently powerful alternative. And here I

don't necessarily mean "formal proofs": I regard axiomatics as the accountancy of mathematics, or to use another metaphor: a formal treatment relates to my power of understanding as a legal document to my sense of justice.

The concern about correctness proofs had an immediate consequence. If proofs get longer and longer they loose their convincing power very, very quickly. It also emerged that the length of a correctness proof could depend critically upon the structure of the program concerned and with this observation a legitimate objective of program structuring emerged, viz. to shorten the length of the proofs required to establish the confidence in the program's correctness. Such considerations gave rise to a computing science folklore for which I am partly responsible; it centers around keywords such as "hierarchical design" and "levels of abstraction".

The programs should be "correct", but that is certainly not the whole story. Correctness proofs belong to "hard" science - and the more formal the proofs, the harder the science. Considerations about the relation between program structure and proof length are already at the outskirts of hard science. Softer still is the equally vital requirement that the program, the mechanism in general, be adequate, that it be a sufficiently realistic implementation of the model we have in mind. Let me explain this with a simple example. In ALGOL 60 the integer variable is a key concept: whenever it is manipulated it stands for an integer value, but in understanding a program, you don't care about its specific value, you have abstracted from it. Caring about its actual value is something you leave to the arithmetic unit, you yourself understand the program in terms of variables and relations between their values, whatever these values may be. In order not to complicate matters we restrict ourselves to applications where integer values remain quite naturally within the range that might be imposed by the implementation. Suppose now that our machine has a very funny adder, funny in the sense that each integer addition takes one microsecond except when the sum formed happens to be a prime multiple of 7, in which case the addition takes a full millisecond. How do you program for a machine like that? You might prefer to ignore this awkward property of the adder, but if you do so I can change the machine, slowing down the exceptional additions by another factor of thousand, and if necessary I do

so repeatedly. Comes the moment that you can no longer afford to ignore this awkward property: by that time you feel obliged to organize your computations in such a way that the exceptional additions are avoided as much as possible. Then you are in trouble, for a vital abstraction, viz. that of an integer variable that stands for an integer value but you don't care which, is denied to you. And when a vital abstraction is denied to a user, I call the implementation inadequate.

The requirement of adequacy has a direct bearing on our hierarchical design, more precisely on the number of levels we can expect to be distinguishable in such a design. Mind you, I am all in favour of hierarchical systems, we have hierarchical systems all around us. We understand a country in terms of towns, towns in terms of streets, streets in terms of buildings, buildings in terms of floors and walls, floors and walls in terms of bricks, bricks in terms of molecules, molecules in terms of atoms, atoms in terms of electrons and nuclei, nuclei in terms of what-have-you etc. It is a pattern you find all over the complete spectrum ranging from science to children's behaviour. At each next level, however, we describe phenomena of a next order of magnitude. In the example given it is a spatial order of magnitude, in the case of mechanisms where we want to understand what happens, we find ourselves faced with happenings to be understood in different grains of time. It seems characteristic for an adequate design that when we go from one level to the next, the appropriate grain of time will increase by an order of magnitude. If this impression is correct, our adequacy requirement imposes an upper bound on the number of levels admissible in our hierarchy, even if we start at the bottom at nano-second level. Then we must conclude that, although essential, hierarchical levelling cannot be the only pattern according to which "Divide and Rule" is to be applied.

Now the design process itself. Many of its aspects can better be treated by greater experts in the field than myself; let me confine myself to what I have found lacking in the literature. On the one hand you find authors writing about problem solving: they stress psychological conditioning in order to hit the unexpected solution and search strategies. Their descriptions of the inventive process seem honest, their guidelines seem relevant, but they confine themselves to small size problems. On the

other hand I have met people trying to organize large scale design projects. They were mostly Americans and talked with the self-assurance that we tend to connect with competence. I am perfectly willing to admit that once or twice I have been taken in by their eloquence, but never for long and I have come to the conclusion that the organization expert, although potentially useful, will not provide the final answer. A few things have struck me very forcibly. Firstly, they persist in thinking in exactly two dimensions - this must have something to do with the two-dimensional paper on which they draw their organization charts. Secondly, they are obsessed by reducing the elapse time; this gives them the opportunity to introduce their dear tools such as PERT diagrams, base-lines etc. but I am much more interested in the designs we don't know how to achieve even if we are not in such a great hurry. Thirdly, they have such preset notions about documentation standards and the holy rules of the game - such as design reviews - that the whole design effort looses the ring of reality and degenerates into a complicated parlour game. But the fourth thing is probably the worst: apparently they do not know the essential difference between "vague" and "abstract" where it is the function of abstraction to create a level of discourse where one can then be absolutely precise!

Let me now give you what I regard as my expectations. I leave it to you to decide whether you prefer to regard them as my hopes.

Our insight in the effectiveness of patterns of reasoning when applied to the task of understanding why mechanisms work correctly and adequately, has been growing considerably in the recent past and I expect it to grow still further.

Our insight in the design process will also increase. In particular I expect that more recognition will be given to the circumstance that designing something large and sophisticated takes a long time. As a result we must take the intermediate stages of the design into consideration and must be clear about their status in relation to each other and to the complete design. I expect a clearer insight in the abstractions involved in postponing a commitment.

From a better understanding of the relation between the final design and its intermediate stages I expect a body of knowledge that will enable us to judge the adequacy of descriptive tools such as programming languages.

In the course of the design process we are envisaging a final product: how well it behaves will ultimately only be known by the time the design is completed and the mechanism is actually used. By its very nature the design process makes heavy demands on our predictive powers. In connection with that I expect two things to happen. On the one hand our predictive techniques will be refined: at present, for instance, the outcome of simulation studies tends to be the source of heated arguments and it appears that we can simulate but lack the proper discipline that tells us what weight to attach to these simulations. Refinement of predictive techniques is one thing, the other thing I expect is that we shall learn how to reduce the need for them. In the design process it is not unusual that some sort of performance measure is dependent in a complicated and only partially known way on a design parameter whose value has to be chosen. There are two usual approaches to this problem and they seem to be equally disastrous. One of them is to give a group the duty to discover the best value of the parameter. As they don't know how to do this, any answer they produce will fail to be convincing and as a rule this approach leads to heated arguments and an overall paralysis of the design process. The other approach leaves the parameter free, so that the user can set it, suited to his own needs. Here the designer has shirked his responsibilities and leaves to the user that part of his task that he could not do himself: this second approach is disastrous because often the user is equally unable to fix the parameter in a sensible way. Both approaches being equally unattractive I expect the competent designer to become more alert when the problem of the parameter with unknown optimum value presents itself. The most efficient way to solve a problem is still to run away from it if you can, and one can try to restructure a design in such a way that the parameter in question looses its significance. In individual applications the performance might be less than optimal but this can be easily outweighed by greater adequacy over a wider range of applications and the easier justification of the remaining decisions. This is an example of the impact of the requirement of "designability" upon the final product.

This was a very rough sketch of a few of my expectations of an emerging methodology of design. I am not going to refine in this talk the picture any further for part of my expectation is that further refinement

will require the next ten years. But by that time I expect a body of teachable knowledge that can be justly called "a methodology of design". Other authors are less modest in their expectation: Herbert A. Simon argues in his little booklet "The Sciences of the Artificial", which I can recommend warmly, that what he dares to call "a science of design" is already emerging. He may very well be right; personally I feel that I lack the wide experience needed to judge his prophecies.

I would like to end with a final remark in order not to raise false hopes. The remark is that a methodology is very fine but in isolation empty. We expect a true methodology of design to be relevant for a wide class of very different design activities. The counterpart of its generality is by necessity that it can only have a moderate influence on each specific design activity, i.e. we must expect each specific design activity to be heavily influenced by the pecularities of the problem to be solved. And that is where knowledge about and deep understanding of the specific problem enters the picture. Yet a methodology, although absolutely insufficient in itself, may be of great value. It should give us the delimitation of our human abilities, it could very well result in a modest list of "don't"'s, rules that we must obey and can only transgress at our own peril.

A SURVEY OF STABILIZED RUNGE-KUTTA FORMULAE

P.J. van der Houwen *)

1. INTRODUCTION

The greater part of the literature on Runge-Kutta methods is devoted to increasing the order of accuracy. Relatively little attention is paid to improving the stability of these methods. Yet, when standard Runge-Kutta methods are applied to stiff equations or partial differential equations, it is the severe stability condition which make them unattractive for numerical integration.

In 1966 Lawson [10] published a fifth order method using 6 points of which the real stability boundary is maximized. Although the gain factor is nearly 1.8 compared with the fifth order formula of Nystrom, it is still much too small for efficient integration of stiff equations, while for partial differential equations a fifth order scheme is seldom required. In 1968 Lomax [12] gave a set of second order Runge-Kutta formulae with increased *real* stability boundaries. These formulae may be used for the integration of diffusion problems and are, although not optimal, very useful. Unfortunately, they are hardly known in literature. In the same year we published in [4] a first, second and fourth order method of which the *imaginary* stability boundaries are maximized. These formulae were developed for the computation of the water elevation on the North Sea and are more generally applicable to symmetric hyperbolic systems. In 1969 further formulae were given [5]. In 1970 we concentrated on the problem of maximizing the real stability boundary as was already tried by Lomax. In [7] results are given for formulae of order 1, 2, 3 and 4.

When dealing with stiff equations, the methods developed in the references mentioned above are not appropriate. Better methods can be constructed by applying the idea of "exponential fitting" (a terminology introduced by Liniger), which was first used by Pope [13] in 1963, to Runge-Kutta type methods. In [6] some first results were given. In this paper a more general treatment of the application of this fundamental idea will be given.

*) Mathematical Centre, Amsterdam.

2. THE GENERAL RUNGE-KUTTA FORMULA

Suppose it is required to find the (approximate) solution of the set of simultaneous first-order differential equations

$$\frac{dy}{dx} = f(x,y) \tag{2.1}$$

with the initial condition

$$y(x_0) = y_0. \tag{2.2}$$

In order to solve equation (2.1) we shall consider numerical integration methods of the type

$$\begin{cases} y_{m+1} = y_m + \sum_{j=0}^{n-1} \theta_j k_j, \\ k_j = h_m f(x_m + \mu_j h_m, y_m + \sum_{l=0}^{j-1} \lambda_{jl} k_l), \quad \mu_0 = 0, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad j = 0, 1, \ldots, n-1, \end{cases} \tag{2.3}$$

where θ_j, μ_j and λ_{jl} are real parameters to be determined; y_m is the numerical approximation to $y(x_m)$, x_m is defined by the relation

$$x_{m+1} = x_m + h_m, \quad m = 0, 1, \ldots$$

and h_m is the step length.

Scheme (2.3) is called an explicit n-point single-step Runge-Kutta formula.

One may represent such a Runge-Kutta formula in a more condensed form (cf. Butcher [1]) by the matrix

$$(2.3') \qquad R = \begin{bmatrix} \mu_1 & \lambda_{10} & 0 & 0 & \cdots & 0 \\ \mu_2 & \lambda_{20} & \lambda_{21} & 0 & \cdots & 0 \\ \mu_3 & \lambda_{30} & \lambda_{31} & \lambda_{32} & \cdots & 0 \\ \cdot & \cdot & & & & \\ \cdot & \cdot & & & & \\ \cdot & \cdot & & & & \\ \mu_{n-1} & \lambda_{n-10} & \cdots & & \lambda_{n-1\,n-2} & \\ \theta_0 & \theta_1 & & \cdots & \theta_{n-1} & \end{bmatrix}.$$

3. CHARACTERISTIC PARAMETERS

To the general Runge-Kutta formula (2.3) we may associate a second parameter matrix

$$(3.1) \qquad C = \begin{bmatrix} \beta_1 & 0 & 0 & & \cdots & & 0 \\ \beta_2 & 0 & 0 & & \cdots & & 0 \\ \beta_3 & \beta_{31} & 0 & & \cdots & & 0 \\ \beta_4 & \beta_{41} & \beta_{42} & \beta_{43} & 0 & \cdots & 0 \\ \cdot & & & & & & \\ \cdot & & & & & & \\ \cdot & & & & & & \\ \beta_n & & & & \cdots & & \end{bmatrix},$$

where the entries are given functions of the Runge-Kutta parameters θ, μ and λ. For instance,

$$\beta_1 = \sum_{j=0}^{n-1} \theta_j,$$

$$\beta_2 = \sum_{j=1}^{n-1} \theta_j \mu_j.$$

For further definitions we refer to [8].

The parameters β are characteristic for the corresponding Runge-Kutta

formula. Consistency and stability conditions can be expresssed directly in terms of these variables. These conditions determine, in fact, the parameters β. Suppose that we have established the matrix C. Then the problem arises to find the generating matrix R. This leads to the solution of a set of, partly non-linear, algebraic equations for the Runge-Kutta parameters θ, μ and λ. In the present work we shall not elaborate on this problem, since a detailed treatment is given in [8]. We only remark that we have tried to find solutions which minimize the storage requirements in order to make the Runge-Kutta formula suitable for the integration of very large sets of differential equations such as the ones originating from partial differential equations.

4. CONSISTENCY CONDITIONS

Let us expand y_{m+1}, defined by (2.3), in terms of powers of h_m. We then obtain a series of the form (cf. [8])

$$(4.1) \qquad y_{m+1} = y_m + \beta_1 h_m y'_m + \beta_2 h_m^2 y''_m + \beta_3 h_m^3 D_m y''_m + \\ + \tfrac{1}{2}\beta_{31} h_m^3 (y'''_m - D_m y''_m) + \ldots,$$

where D_m represents the Jacobian matrix of system (2.1) at the point (x_m, y_m).

We shall call a Runge-Kutta formula p-th order exact when expansion (4.1) agrees with the solution of the differential equation through the point (x_m, y_m) to p terms in a Taylor series. This definition leads to the following table of consistency conditions (cf. [8])

p	β_1	β_2	β_3	β_{31}	β_4	β_{41}	β_{42}	β_{43}
1	1							
2	1	1/2						
3	1	1/2	1/6	1/3				
4	1	1/2	1/6	1/3	1/24	1/12	1/8	1/4

In the cases p = n these conditions determine the characteristic matrix C. When n > p some parameters remain undetermined. One may use them to improve the stability of the Runge-Kutta formula.

5. THE STABILITY POLYNOMIAL

Let y_m and y_m^*, m = 0, 1, 2, ..., be two solutions of the Runge-Kutta scheme (2.3) and let

$$e_m = y_m - y_m^*. \tag{5.1}$$

Then it can be proved that (cf. [8])

$$e_{m+1} = P_n(h_m D_m)e_m + O(e_m^2) \quad \text{as } e_m \to 0, \tag{5.2}$$

where

$$P_n(z) = 1 + \beta_1 z + \beta_2 z^2 + \ldots + \beta_n z^n. \tag{5.3}$$

We shall call $P_n(z)$ the stability polynomial associated to the Runge-Kutta scheme (2.3).

Furthermore, let δ be an eigenvalue of the Jacobian D_m. The expression $P_n(h_m\delta)$ is said to be the amplification factor corresponding to the eigenvalue δ at the point (x_m, y_m). Clearly, the amplification factors have to do with the stability properties of the Runge-Kutta formula. We shall adopt the following definition of stability: *A Runge-Kutta formula is stable when all amplification factors corresponding to eigenvalues δ with a non-positive real part are on or within the unit circle.* This implies that the points $h_m\delta$, for which Re $\delta \leq 0$, are required to be in the stability domain S defined by

$$S = \{z \mid |P_n(z)| \leq 1\}. \tag{5.4}$$

6. SOME POLYNOMIAL PROBLEMS

In this section we indicate the difficulties which arise when we try to solve by a standard Runge-Kutta method the following classes of equations:

(1) parabolic differential equations;

(2) hyperbolic differential equations;

(3) stiff differential equations.

We shall show that, in order to overcome these difficulties, we are led to three different polynomial problems which will also be discussed in the following three subsections.

6.1. PARABOLIC DIFFERENTIAL EQUATIONS

Many parabolic differential equations (e.g. diffusion equations) lead, after discretization of the space variables (cf. Goodwin [3], p. 113), to an ordinary differential equation of type (2.1) of which the Jacobian matrix D has negative eigenvalues δ with the property

$$|\delta|_{min} << |\delta|_{max}. \tag{6.1}$$

Let $[-\beta,0]$ be the segment of the real axis which belongs to the stability region S of the Runge-Kutta formula to be used. Then, the stability condition becomes

$$-\beta \leq h_m\delta \leq 0, \tag{6.2}$$

or, equivalently,

$$h_m \leq \frac{\beta}{|\delta|_{max}}. \tag{6.2'}$$

For the standard Runge-Kutta formulae the (real) stability boundary β is relatively small (β = 2, 2, 2.5 and 2.8 for n = p = 1, 2, 3 and 4, respectively). Hence condition (6.2') is a fairly stringent one in the case

of parabolic equations.

It is possible to construct Runge-Kutta formulae with a relatively large stability interval on the negative axis. This may be achieved by solving the following polynomial problem:

Let p *and* n *be given numbers,* $n > p$, *and let* $P_n(z)$ *be of the form*

$$P_n(z) = 1 + z + \frac{1}{2!} z^2 + \ldots + \frac{1}{p!} z^p + z^{p+1} B_q(z),$$

where $B_q(z)$ *is a polynomial with real coefficients of degree* $q = n-p-1$. *Then it is required to determine* $B_q(z)$ *such, that the real stability interval is as large as possible.*

When $p = 1$ this problem is solved by the polynomials $T_n(1+z/n^2)$ with $\beta = 2n^2$. For $p > 1$ we did not succeed to find an analytical solution. In [7] a numerical solution method is presented. In table 6.1 some of the results are listed.

Table 6.1 Real stability boundaries

p \ n	1	2	3	4	5	6	7	8	9	10	$n \to \infty$
1	2	8	18	32	50	72	98	128	162	200	$2\,n^2$
2	-	2	6.3	12	19	28	39	51	65	81	$.82n^2$
3	-	-	2.5	6	10	16	22	30	38	48	$.49n^2$
4	-	-	-	2.8	4.8	10	14	19	26	32	$.34n^2$

Note that the stability boundary is proportional to n^2!

6.2. HYPERBOLIC DIFFERENTIAL EQUATIONS

Cauchy problems for symmetric hyperbolic differential equations reduce by discretization of the space variables to sets of ordinary differential equations of which the Jacobian has purely imaginary eigenvalues. As in the case of parabolic equations the value of $|\delta|_{max}$ is usually very large. When

$[-i\beta, i\beta]$ is the stability interval on the imaginary axis we find the stability condition

$$h_m \leq \frac{\beta}{|\delta|_{max}} . \tag{6.3}$$

The standard Runge-Kutta formulae of order 1 until 4 have imaginary stability boundaries 0, 0, 1.7, 2.8, respectively. As in the case of real eigenvalues one may try to construct Runge-Kutta formulae with larger stability intervals on the imaginary axis. This leads to a similar problem as stated in the preceding section.

Putting $p = 1$ we found by analytical methods the following polynomials [4,5]:

$$\begin{cases} P_2(z) = 1 + z + z^2, \ \beta = 1 \\ P_3(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{4}z^3, \ \beta = 2 \\ P_4(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4, \ \beta = 2\sqrt{2}, \\ P_n(z) = T_{\frac{n-1}{2}}\left(\frac{(n-1)^2+2z^2}{(n-1)^2}\right) + 2z\,\frac{(n-1)^2+z^2}{(n-1)^3}\,U_{\frac{n-3}{2}}\left(\frac{(n-1)^2+2z^2}{(n-1)^2}\right), \\ \beta = n-1, \ n = 1, 3, 5, \ldots . \end{cases} \tag{6.4}$$

We note that for $n \geq 3$ these stability polynomials are compatible with the consistency conditions of second order exact schemes and for $n = 4$ even with the conditions of a fourth order exact scheme. Furthermore, we observe that, unlike the real eigenvalue case, the effective value of β, i.e. β/n, does not change much as n increases. These considerations have decided us not to try to solve the cases where n is even or $p > 2$, but to use the standard fourth order Runge-Kutta formulae in the case of hyperbolic equations.

6.3. STIFF DIFFERENTIAL EQUATIONS

Many physical systems give rise to ordinary differential equations of which the eigenvalues with a negative real part can be placed in a few

widely separated clusters. Such equations are said to be stiff.

Let us consider the case where the eigenvalues δ are situated in three clusters as illustrated in figure 6.1

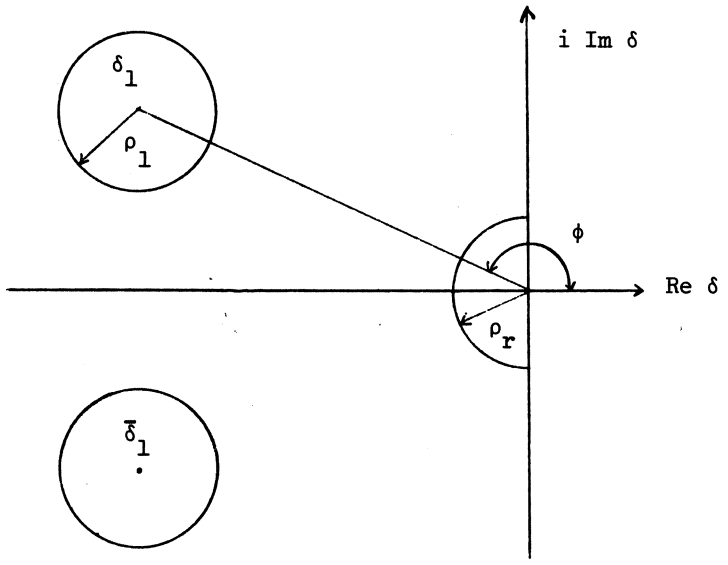


fig. 6.1 Eigenvalue spectrum of a stiff equation

This spectrum differs from the ones considered in preceding sections by the fact that the eigenvalues are located in a clearly disconnected region, instead of a connected region. This suggests to construct stability polynomials with a disconnected stability domain, that is a stability domain composed of three regions which are centered at the points $z = z_1 = h_m\delta_1$, $z = \bar{z}_1$ and the origin. Since the stability region near the origin of a stability polynomial is mainly determined by its first terms it is easy to adapt $P_n(z)$ to the right hand cluster by prescribing the first coefficients of $P_n(z)$. We are then led to the following problem:

Let $R_n(z)$ *be a given polynomial of degree* r *in* z, $r < n$ *and let* $P_n(z)$ *be of the form*

$$P_n(z) = R_r(z) + z^{r+1} L_1(z), \tag{6.5}$$

where $L_1(z)$ *is a polynomial with real coefficients of degree* $l = n-r-1$. *Then it is required to determine* $L_1(z)$ *such that the stability domain of* $P_n(z)$ *contains a neighbourhood of a given point* z_1 *in the left half-plane, which is as large as possible on the by-condition that the coefficients of* $L_1(z)$ *remain bounded as* $z_1 \to 0$.

The polynomial $R_r(z)$ in representation (6.5) is supposed to be determined by the consistency conditions and the right hand eigenvalue cluster. In order to select $R_r(z)$ appropriately one may use the results presented in section 6.1 and 6.2.

Integration methods with stability polynomials of the type described above will be called cluster methods. In the remaining sections some recent results of our study of such methods will be given.

7. STABILITY REGIONS OF CLUSTER METHODS FOR LARGE STEP SIZES

It may be expected that the stability condition associated to the polynomial which solves the problem stated in section 6.3 is more severe as $|z_1|$ is larger. Therefore, we first consider this problem for $z_1 \to \infty$, i.e. h_m relatively large.

Our starting point is the observation that the zeroes of the polynomial $P_n(z)$ will also be situated in clusters centered at $z = z_1$, $z = \bar{z}_1$ and $z = 0$. This suggests to write $P_n(z)$ in the form

$$P_n(z) = F_{n_1}(z)\, C_{n_2}(z), \quad n_1 + n_2 = n, \tag{7.1}$$

where the polynomials $F_{n_1}(z)$ and $C_{n_2}(z)$ are supposed to have their zeroes respectively far from and close to the origin.

Two cases will be distinguished. The case where z_1 and $\bar{z}_1$ are widely separated and the case where the left hand clusters are close to the real axis. In the first case we assume for the sake of simplicity that n_1 is even. We may then write

$$P_n(z) = \prod_{j=1}^{q} (z-z(j)) \prod_{j=1}^{q} (z-\bar{z}(j))\, C_{n_2}(z), \tag{7.2}$$

where $z(j)$ represents a zero of F_{n_1} in the neighbourhood of z_1. In this neigbourhood the polynomial $P_n(z)$ behaves as a polynomial $Q_q(z)$ of degree q in z which is given by $(z_1 \to \infty)$

$$(7.3) \qquad Q_q(z) = \prod_{j=1}^{q} (z-z(j)) \prod_{j=1}^{q} (z_1-\bar{z}(j))\, C_{n_2}(z_1).$$

We now apply the following lemma (cf. [14]):

LEMMA OF ZARANTONELLO

Of all polynomials $Q_q(z)$ of degree q in z normed by the relation

$$Q_q(0) = 1,$$

the polynomial

$$\left(\frac{z_1-z}{z_1}\right)^q$$

has the smallest maximum norm over the circle $|z-z_1| = \rho \leq |z_1|$.

This lemma indicates that the best thing we can do is to choose all zeroes $z(j)$ in the point z_1 and, therefore, all zeroes $\bar{z}(j)$ in the point $\bar{z}_1$. This leads to the representation

$$(7.4) \qquad P_n(z) = (z-z_1)^q\, (z-\bar{z}_1)^q\, C_{n_2}(z).$$

Clearly we wish to give $P_n(z)$ as large a number of zeroes as possible in the left hand stability regions. From the representation (6.5) it follows that we have l+1 free parameters so that

$$(7.5) \qquad n_1 = 2q = l+1, \quad n_2 = r.$$

The coefficients of the polynomial $C_{n_2}(z) = C_r(z)$ are determined by the relations

$$P_n^{(j)}(0) = \beta_j, \quad j = 0, 1, \ldots, r. \tag{7.6}$$

We shall use, however, a more simple method which yields approximate values for the coefficients of $C_r(z)$. Consider the representations (6.5) and (7.4) for small values of z. We then may write

$$P_n(z) \sim R_r(z)$$

and

$$P_n(z) \sim |z_1|^{2q} C_r(z);$$

hence

$$C_r(z) \sim |z_1|^{-2q} R_r(z). \tag{7.7}$$

Substitution into (7.4) yields

$$P_n(z) \sim |z_1|^{-2q} (z-z_1)^q (z-\bar{z}_1)^q R_r(z). \tag{7.8}$$

Having found the stability polynomial, the stability regions are easily established. From (7.8) it follows that the right hand stability region, as already observed in the preceding section, is bounded by the curve

$$|R_r(z)| = 1 \quad \text{as } z_1 \to \infty. \tag{7.9}$$

The left hand stability regions follow from the representations

$$P_n(z) \sim |z_1|^{-2q} (z-z_1)^q (z_1-\bar{z}_1)^q \beta_r z_1^r$$

and

$$P_n(z) \sim |z_1|^{-2q} (z-\bar{z}_1)^q (\bar{z}_1-z_1)^q \beta_r \bar{z}_1^r,$$

which are valid in the neighbourhood of z_l and $\bar{z}_l$, respectively. A simple calculation yields the circles

$$|z-z_l| = \frac{|z_l|^{\frac{q-r}{q}}}{2|\sin\phi|\,\beta_r^{\frac{1}{q}}} \tag{7.10}$$

and

$$|z-\bar{z}_l| = \frac{|z_l|^{\frac{q-r}{q}}}{2|\sin\phi|\,\beta_r^{\frac{1}{q}}} \tag{7.10'}$$

, $q = \frac{l+1}{2}$.

Here, ϕ denotes the argument of the point z_l in the complex plane (see figure 6.1).

In a similar manner the stability regions in the two-cluster case (z_l close to the real axis) can be derived. We find, as $z_l \to \infty$, a right hand stability region bounded by the curve (7.9) and a left hand stability region bounded by the circle

$$|z-z_l| = \frac{|z_l|^{\frac{l+1-r}{l+1}}}{\beta_r^{\frac{1}{l+1}}} \quad . \tag{7.11}$$

In this case l is allowed to assume every integer value.

Note that the left hand stability regions depend on the value of the step size h_m. When h_m increases these stability regions become relatively smaller. In order to see this let h_m increase by a factor 2; then the radius of the cluster of $h_m\delta$ values also increases by a factor 2, but the radius of the stability circle only increases by a factor $2^{\frac{q-r}{q}}$ and $2^{\frac{l+1-r}{l+1}}$, respectively. Thus, for a given cluster of eigenvalues we always have a bound on the step size h_m. These maximal step size is easily derived from (7.10) and (7.11). We have

$$h_{max} = \frac{c_3^{\frac{q}{r}}}{|\delta_1|\beta_r^{\frac{1}{r}}}, \qquad c_3 = \frac{|\delta_1|}{2\rho_1|\sin\phi|} \tag{7.12}$$

and

$$h_{max} = \frac{c_2^{\frac{l+1}{r}}}{|\delta_1|\beta_r^{\frac{1}{r}}}, \qquad c_2 = \frac{|\delta_1|}{\rho_1} \tag{7.13}$$

for the three- and two-cluster case, respectively. The constants c_3 and c_2 characterize the position and relative magnitude of the clusters of eigenvalues.

In order to compare conditions (7.12) and (7.13) with the stability conditions of the methods discussed in section 6.1 and 6.2 we give some values of the stability boundary $\beta = h_{max}|\delta_1|$ for the cases

$$r = p \quad c_3 = 100, \tag{7.14}$$

$$r = p \quad c_2 = 100. \tag{7.15}$$

Since $r = p$ implies $\beta_r = 1/r!$ we have, respectively,

$$\beta = (100\ p!)^{\frac{n-p}{2p}} \tag{7.16}$$

and

$$\beta = (100\ p!)^{\frac{n-p}{p}} \tag{7.17}$$

for the three- and two-cluster method.

Table 7.1 Stability boundaries of the three-cluster method

r=p \ n	3	4	5	6	7	8	9	10
1	100	-	10^4	-	10^6	-	10^8	-
2	-	14.1	-	200	-	2828	-	$4\ 10^4$
3	-	-	8.4	-	71	-	600	-
4	-	-	-	7	-	49	-	350

Table 7.2 Stability boundaries of the two-cluster method

r=p \ n	2	3	4	5	6
1	100	10^4	10^6	10^8	10^{10}
2	-	14.1	200	2828	$4\ 10^4$
3	-	-	8.4	71	600
4	-	-	-	7	49

An examination of these values reveals that for equations with clustered eigenvalue spectra characterized by a constant c_2 or c_3 of magnitude 100, the cluster methods are superior over the other stabilized Runge-Kutta methods.

It should be remarked that in cases where it is known that all left hand eigenvalues are real, i.e. negative, it is not an optimal strategy to concentrate all left hand zeroes of the stability polynomial in one point. Stability condition (7.13) can be relaxed by distributing the zeroes over a neighbourhood of z_1 on the negative axis.

8. EXPONENTIAL FITTING

In the preceding section the stability polynomial of the cluster methods was determined for large values of $|z_1| = h_m|\delta_1|$. This polynomial cannot be used as $h_m \to 0$, because then the coeffients β_j of the polynomial $L_1(z)$ in representation (6.5) become singular. In order to overcome this difficulty we consider the amplification factors of the differential equa-

tion given by $\exp(h_m\delta)$ and, analogous to the stability polynomial $P_n(z)$ of the difference scheme,we introduce the stability function exp z of the differential equation. It is natural to require that in the neighbourhood of z_1 the polynomial $P_n(z)$ behaves as exp z as $z_1 \to 0$. This may be achieved by putting

$$P_n^{(j)}(z_1) = e^{z_1}, \quad j = 0, 1, 2, \ldots, \frac{l-1}{2} \tag{8.1}$$

in the three-cluster case (z_1 complex) and

$$P_n^{(j)}(z_1) = e^{z_1}, \quad j = 0, 1, 2, \ldots, l \tag{8.2}$$

in the two-cluster case (z_1 real). In this way the coefficients β_j, $j = r+1, \ldots, r+l+1 = n$ are real and bounded as $z_1 \to 0$, while for $z_1 \to \infty$, i.e. exp $z_1 \to 0$, the same polynomials are obtained as derived in the preceding section. We may interpret the polynomial $P_n(z)$ defined by (6.5) and (8.1) or (8.2) as a two-point Taylor expansion of teh stability function exp z (in the neighbourhood of z = 0 the consistency conditions guarantee that $P_n(z) \sim \exp z$).

It may be remarked that fitting the amplification factors of a difference scheme with those of the differental equation in other points than the origin, is frequently encountered in literatur. We mention Pope [13], Liniger and Willoughby [11] and Fowler and Warten [2]. Furthermore, in the analysis of convection difference schemes a global fitting is often used, that is one tries to minimize some norm of the difference of the amplification factors of the difference scheme and the differential equation.

9. EXAMPLES OF STABILIZED RUNGE-KUTTA FORMULAE

In this section the generating matrix R of some stabilized Runge-Kutta formulae are given. For numerical results obtained by these formulae we refer to [9].

A first and second order scheme appropriate for the integration of diffusion equations is given by respectively

$$R = \begin{bmatrix} \frac{1}{64} & \frac{1}{64} & 0 & 0 \\ \frac{1}{20} & 0 & \frac{1}{20} & 0 \\ \frac{5}{32} & 0 & 0 & \frac{5}{32} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad h_m \leq \frac{32}{|\delta|_{max}}$$

and

$$R = \begin{bmatrix} .04621218 & .04621218 & 0 & 0 \\ .15616897 & 0 & .15616897 & 0 \\ .5 & 0 & 0 & .5 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad h_m \leq \frac{12}{|\delta|_{max}}.$$

A second order scheme appropriate for the integration of hyperbolic equations is given by

$$R = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 \\ \frac{1}{6} & 0 & \frac{1}{6} & 0 & 0 \\ \frac{3}{8} & 0 & 0 & \frac{3}{8} & 0 \\ \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad h_m \leq \frac{4}{|\delta|_{max}}.$$

A first order exact scheme appropriate for the integration of stiff equations, the eigenvalues of which can be placed in two or three clusters, is given by

5.18

$$R = \begin{bmatrix} \frac{16\beta_3}{16\beta_2-3} & \frac{16\beta_3}{16\beta_2-3} & 0 \\ \frac{16\beta_2-3}{12} & 0 & \frac{16\beta_2-3}{12} \\ \frac{1}{4} & 0 & \frac{3}{4} \end{bmatrix},$$

where

$$\beta_2 = \frac{-2\cos\phi}{|z_1|} - \frac{4\cos^2\phi-1}{|z_1|^2} - \exp(|z_1|\cos\phi)\,\frac{\sin(|z_1|\sin\phi-3\phi)}{|z_1|^2\sin\phi},$$

$$\beta_3 = \frac{1}{|z_1|^2} + \frac{2\cos\phi}{|z_1|^3} + \exp(|z_1|\cos\phi)\,\frac{\sin(|z_1|\sin\phi-2\phi)}{|z_1|^3\sin\phi}.$$

REFERENCES

[1] Butcher, J.C., Implicit Runge-Kutta processes, Math. Comp. 18, 50 (1964).

[2] Fowler, M.E., R.M. Warten, A numerical integration technique for ordinary differential equation with widely seperated eigenvalues, I.B.M. J. Res. Dev., 11, 537 (1967).

[3] Goodwin, E.T. (editor), Modern computing methods, Her Majesty's Stationery Office, London (1961).

[4] Houwen, P.J. van der, Finite difference methods for solving partial differential equations, MC tract 20, Mathematisch Centrum, Amsterdam (1968).

[5] Houwen, P.J. van der, Difference schemes with complex time steps, Report MR 105, Mathematisch Centrum, Amsterdam (1969).

[6] Houwen, P.J. van der, One-step methods for linear initial value problems II, Applications to stiff equations, Report TW 122/70, Mathematisch Centrum, Amsterdam (1970).

[7] Houwen, P.J. van der, J. Kok, Numerical solution of a minimax problem, Report TW 123/71, Mathematisch Centrum, Amsterdam (1971).

[8] Houwen, P.J. van der, Stabilized Runge-Kutta methods with limited storage requirements, Report TW 124/71, Mathematisch Centrum, Amsterdam (1971).

[9] Houwen, P.J. van der, P. Beentjes, K. Dekker, E. Slagt, One-step methods for linear initial value problems III, Numerical examples, Report TW 130/71, Mathematisch Centrum, Amsterdam (1971).

[10] Lawson, J.D., An order five Runge-Kutta process with extended region of stability, SIAM J. Numer. Anal. 3, 593 (1966).

[11] Liniger, W., R.A. Willoughby, Efficient integration methods for stiff systems of ordinary differential equations, SIAM J., Numer. Anal. 7, 47 (1970).

[12] Lomax, H., On the construction of highly stable, explicit, numerical methods for integrating coupled ordinary differential equations with parasitic eigenvalues, NASA Technical Note, NASA TN D - 4547 (1968).

[13] Pope, D.A., An exponential method of numerical integration of ordinary differential equations, Communications of the ACM, 6, 491 (1963).

[14] Varga, R.S., A comparison of the successive overrelaxation method and semi-iterative methods using Chebyshev polynomials, SIAM J., Appl. Math. 5, 39 (1957).

ON THE BOOKKEEPING OF SOURCE-TEXT LINE NUMBERS DURING THE EXECUTION PHASE OF ALGOL 60 PROGRAMS

F.E.J. Kruseman Aretz *)

SUMMARY

Analysis of the algorithm by which the MC ALGOL 60 compiler for the EL X8 generates instructions in the object code necessary for the bookkeeping of source-text line numbers during the execution phase of ALGOL 60 programs for the purpose of diagnostics.

1. INTRODUCTION

The subject of this paper is of a rather specific and technical nature. It describes the method by which, for the ALGOL 60 system on the EL X8 as developed at the Mathematical Centre, run-time error messages are provided with a line number. This number refers to the line of the ALGOL source text on which the first basic symbol occurs of the smallest statement whose execution has been left incompleted.

For the production of this line number in the case of an error the object code as generated by the ALGOL compiler is interwoven with instructions that assign the appropriate value to a (global) system variable called "LNC" (for line counter). This method leads to the problem of where to produce these instructions: for the sake of both object-program length and execution time one wishes to keep the number of occurrences of the LNC-setting macro as small as possible.

First, of course, one has to agree upon the level of semantic analysis to be used by the compiler for the decisions where to produce the LNC-setting macro. For instance, the statement "S" in "if false then S" will never be executed. Some compilers might detect this fact and use it for a reduction of the number of insertions of the LNC-setting macro. In the case of the MC ALGOL 60 compiler the level of semantic analysis is restricted to certain simple rules as given in section 2.

The original algorithm, constructed in 1965, was designed more or less

*) Philips Research Laboratories, Eindhoven

intuitively, using examples and counterexamples. Only last year were we able to give a complete analysis of the problem and to prove the correctness of a slightly modified algorithm: the original scheme suffered from some minor errors. It is this analysis that the paper is devoted to.

One could wonder why it took so much time to clarify the situation. In our opinion this is due to the fact that one has to deal with three different elements:

(i) the static source text of a given ALGOL program;

(ii) the reactions of the ALGOL compiler on the source text during the transformation of the program into object code;

(iii) the dynamic flow of action during execution of the object code.

Only by a clear distinction between these three elements can one avoid confusion. The analysis presented here goes roughly along the following lines:

In section 3 it is examined what information can be obtained for the value of the variable LNC upon completion of the execution of a given statement from its static structure, given the semantic rules of section 2.

In section 4 an algorithm is given that derives that information in a left to right scan of the given statement.

In section 5, finally, a very simple set of rules is given for the points in the object program where the variable LNC has to be given a (new) value, and its optimality is demonstrated.

For the author this study has been an exercise in recursion, in the semantics of ALGOL 60, and in the analysis of algorithms for correctness. The treatment of the paper will be rather informal, although, to the author's conviction, a complete formalization will be possible and fruitful.

The author is much indebted to B.J. Mailloux who contributed considerably to the original algorithm, and to P.J.W. ten Hagen, H.L. Oudshoorn and H.W. Roos Lindgreen for many hours of discussion of the subject.

2. SEMANTIC RULES

As has been stated in section 1, a run-time error message in the MC

ALGOL 60 system for the EL X8 refers to the line of the program source text on which the first basic symbol occurs of the smallest statement that at the moment of error detection has been taken into execution and not yet been completed. In the case of the program:

```
begin integer i, j; integer array a[1:1];
      a[1]:= 1;
      for i:= 1 step 1 until 10 do
      if i = 3 then
      for j:= 0, a[j] do
      print(j)
end
```

the error message will refer to the fifth line of this program.

The line number to be displayed in an error message is taken from the system variable LNC. During the execution of a program this variable must be updated each time a statement that might lead to an error is entered or left, unless its old and new values surely coincide. Therefore we need to decide:

(i) which statements can suffer from errors that are not made within a substatement;

(ii) what can be said about the value of LNC each time a statement is entered or left.

In the MC ALGOL 60 compiler for (i) the following rules are used:

a) the execution of a dummy statement is empty; no error can be made in it at all;

b) the execution of a compound statement consists merely of the execution of substatements; no error can be made in it that is not made within one of its substatements;

c) each assignment statement can lead to an error;

d) each goto statement can lead to an error;

e) the execution of a statement "<if clause> S" consists of either the elaboration of the if-clause only, or the elaboration of the if-clause followed by the execution of S; the elaboration of the if-clause can lead to an error;

f) the execution of a statement "<if clause> S1 else S2" consists of either the elaboration of the if-clause followed by the execution of S1, or the elaboration of the if-clause followed by the execution of S2; the elaboration of the if-clause can lead to an error;
g) the execution of a statement "<for clause> S" consists of an alternation of the following two actions:
   1) elaboration of the for-clause,
   2) execution of S;
   the series of actions opens and closes by an elaboration of the for-clause; each elaboration of the for-clause can lead to an error;
h) the execution of a block consists of the elaboration of its declarations followed by the execution of its statements; the elaboration of its declarations can lead to an error;
i) the execution of a procedure statement consists of some unspecified action which can lead to an error followed by the execution of one or more statements somewhere in the program;
j) the execution of a statement "<label> : S" consists of the execution of S.

In the MC ALGOL 60 compiler for (ii) the following rules are used:

a) upon entering the execution of a labeled statement the value of LNC is undefined;
b) upon entering the execution of a procedure body the value of LNC is undefined;
c) the evaluation of a function designator does not change the value of LNC.

In the next sections we will only consider algorithms for the decisions where to insert the LNC-setting macro that do not generate this macro for a dummy or a compound statement. This is no restriction, for, if one has a correct and optimal distribution of LNC-setting macros in an object program, one can move all LNC-setting macros that occur within dummy statements to the actions that follow dynamically on the dummy statements without destroying the correctness. The same kind of argument applies to compound statements. By correct is meant that all error messages will display the intended line number, by optimal is meant that no LNC-setting

macro is generated unless it is necessary for guaranteeing the correctness, given the semantic rules of this section.

In the next section some information concerning the value of LNC upon completion of the execution of a statement will be given. It will be clear that such information is important for predictions about the (old) value of LNC upon entering the execution of a statement, as statements often are executed one after another. The information given in section 3 can be derived from the semantic rules given above together with the following assumption: it is assumed that the algorithm that decides where to generate the LNC-setting macro is correct and optimal.

3. THE VALUE OF LNC UPON COMPLETION OF THE EXECUTION OF A STATEMENT

According to the assertions that can be made about the value of LNC upon completion of the execution of a statement, all statements can be divided into three classes:

class A of statements that do not affect LNC at all; for these statements all assertions about the value of LNC upon exit have to be derived from the context in which these statements occur;

class B of statements for which the value of LNC upon exit will be undefined;

class C of statements for which lower and upper bounds for the value of LNC upon exit can be given.

These three classes are listed in table I. For class C in this table the lower and upper LNC-bounds are also given.

In table I a number of metalinguistic variables are used, some of which are defined below. All production rules that are not presented here can be found in [1].

<X>::= <any statement of class X> X = A,B,C

<C1>::= <C>

<C2>::= <C>

<dummy tail>::= end | <A> ; <dummy tail>

<d and s option>::= <s option> | <declaration> ; <d and s option>

<s option>::= <empty> | <statement> ; <s option>

By "fbs" we denote the line number on which the first basic symbol of the statement occurs. With "lower(C)" we express the lower LNC-bound of the constituent C-statement of the given statement, by "upper(C)" its upper LNC-bound.

TABLE I

A. Statements that do not affect LNC at all

<dummy statement>
begin <dummy tail>

B. Statements for which the value of LNC upon exit will be undefined

<label> : <A>
<procedure statement>
<if clause> <B>
<if clause> <B> else <statement>
<if clause> <statement> else <B>
begin <d and s option> <B> <dummy tail>
<label> : <B>

C. Statements for which lower and upper LNC-bounds upon exit can be given

	lower LNC-bound	upper LNC-bound
<assignment statement>	fbs	fbs
<goto statement>	fbs	fbs
<for clause> <statement>	fbs	fbs
<if clause> <A>	fbs	fbs
<if clause> <A> else <A>	fbs	fbs
<block head> ; <dummy tail>	fbs	fbs
<if clause> <C>	fbs	upper(C)
<if clause> <A> else <C>	fbs	upper(C)
<if clause> <C> else <A>	fbs	upper(C)
<if clause> <C1> else <C2>	lower(C1)	upper(C2)
begin <d and s option> <C> <dummy tail>	lower(C)	upper(C)
<label> : <C>	lower(C)	upper(C)

The following remarks may help the reader to verify that the assertions of table I are indeed valid.
In an unlabelled compound or in an unlabelled block, a dummy tail does not affect LNC, as can be verified recursively. Therefore the effect of the compound statement or block on LNC depends on the most right-hand non-A statement, if any, of its compound tail. If no such statement occurs, a compound statement does not affect LNC and a block will set LNC equal to the line number of its first basic symbol (unless LNC surely did point to that line already).
In the case of execution of a for-statement, the last action performed is an elaboration of the for-clause, during which LNC has to point to the line on which the for-symbol occurs.
No special concern is necessary for the presence of a goto-statement as component of another statement. During the execution of that other statement it is either skipped, and then does not influence the value of LNC, or control is transferred to some statement of the program. In that case the execution of a labelled statement is entered and the value of LNC will be considered to be undefined.

The bounds given in table I are sharp, i.e. these values can be obtained by LNC indeed. By induction the following relation can be easily proved for any statement belonging to class C:

fbs $\leq$ lower LNC-bound $\leq$ upper LNC-bound $\leq$ lbs

in which lbs denotes the line number on which the last basic symbol of the statement occurs. Consequently in the case of a statement of the form "<if clause> <C1> else <C2>" indeed lower (C1) $\leq$ upper (C2).

In the next section we present an algorithm that evaluates the lower and upper LNC-bounds for any statement of class C.

4. COMPILER EVALUATION OF THE LOWER AND UPPER LNC-BOUNDS OF A CLASS-C STATEMENT

In this section a simple algorithm is presented that a.o. evaluates the lower and upper LNC-bounds for any statement of class C. It is assumed that the compiler consumes the symbols of the source text one by one from left

to right, and that the symbol-reading routine keeps a record of the line count in a global compiler variable "lnc". Moreover two other global compiler variables called "lower" and "upper", and a stack will be used.

The effect of the algorithm on the values of lower and upper is as follows:

after the compilation of a statement of class A both lower and upper have been left unchanged;

after the compilation of a statement of class B the value of lower is -1;

after the compilation of a statement of class C lower and upper contain the lower and upper LNC-bounds for that statement.

We now describe the algorithm:

1) after occurrence of a label, lower is set equal to -1;
2) if the first basic symbol of an unlabelled statement is a letter or "goto" or "if" or "for", both lower and upper is set equal to lnc;
3) if the first basic symbol of a statement is "begin", the value of lnc is saved and the next basic symbol is analysed; if this symbol is a declarator (<type> or "array" or "procedure" or "switch"), the value of lnc saved is assigned both to lower and to upper;
4) if the first basic symbol of a statement is "for", the value of lnc is saved in the stack; after compilation of the for-statement this value is unstacked and stored both in lower and in upper;
5) if the first basic symbol of a statement is "if" the value of lnc is saved in the stack; the compilation of the statement after "then" is done (it will influence the value of lower and upper according to its class). If there is no "else", the value of lower is replaced by the value saved in the stack unless lower = -1; otherwise, the value in the stack and the value of lower are interchanged and after the compilation of the statement after "else" the value of lower is replaced by the minimum of lower itself and of the value saved in the stack;
6) after compilation of a procedure statement the value of lower is replaced by -1;
7) before entering the compilation of a procedure declaration the value of lower is saved in the stack, and the value -1 is given to lower; on

completion of the compilation the value in the stack is unstacked and assigned both to lower and to upper.

By enumeration and induction the validity of the assertions given above about the final values of lower and upper can be proved. Note that not all actions in the algorithm are necessary for the proof; e.g., the fact that if the first symbol of a statement is "for" both lower and upper is set equal to lnc. These superfluous actions, however, make sense in the next section, which presents a correct and optimal algorithm for the points in the object program where the LNC-setting macro has to be inserted.

5. THE GENERATION OF THE LNC-SETTING MACRO

A remarkably simple set of rules suffices for the decision where to generate the LNC-setting macro:

1) if the translation of an unlabelled statement that is neither a dummy nor a compound statement is started, and if the value of lnc differs from the old value of lower (i.e. its value before the value of lnc is stored both in lower and upper), the lnc-setting macro is generated in front of the translation of the statement. Note that for this rule the value of upper is irrelevant;
2) if, on completion of the translation of the statement following the "do" of a for-statement, either lower equals -1 or upper is greater than the value of lnc on top of the stack (see section 4), then the LNC-setting macro is generated in front of the macro that transfers control to the further elaboration of the for-clause.

We proceed with an analysis of the correctness and optimality of these rules.

As has been explained in section 2 only those algorithms are considered which do not generate the LNC-setting macro for a dummy or a compound statement. All other statements consist of, or start with, an action that can lead to an error. Therefore from the beginning of their execution onwards LNC has to point to the correct line and if it does not it has to be made so. The only action in a statement which is not itself a statement and which is not started by starting the execution of the statement is the

further elaboration of a for-clause. Therefore only in this case can we not restrict ourselves to the generation of LNC-setting macros in front of the translation of a statement.

If a statement that is neither a dummy nor a compound statement is labelled, the LNC-setting macro is required, because the value of LNC upon entering the execution of a labelled statement is considered to be undefined. It will be generated indeed because lower = -1 < lnc after a label.

If an unlabelled statement that is neither a dummy nor a compound statement is a constituent element of a compound tail, four cases can be distinguised:

1) in the compound tail the statement is, apart from statements of class A, preceded by a statement of class B. In that case the LNC-setting macro is required and indeed produced since lower = -1 < lnc after a statement of type B;
2) in the compound tail the statement is, apart from statements of class A, preceded by a statement of class C. In that case, for the value of LNC upon starting the execution of the statement considered the following relation holds: lower $\leq$ LNC $\leq$ lnc, where lnc points to the line of the first basic symbol of the statement considered. If and only if lower = lnc it is certain that the value of LNC at execution time will be correct and no LNC-setting macro is required.
3) the statement is, apart from statements of class A, the first statement of a block. In that case, both LNC upon entering the execution of the statement and lower upon entering the translation of the statement points to the line on which the first symbol (a "begin") of the unlabelled block occurs, and their value is at most lnc. If lower $\neq$ lnc the LNC-setting macro is required and produced.
4) the statement is, apart from statements of class A, the first statement of a compound statement. Essentially, the execution of a compound statement is entered by entering its first non-class-A statement, and neither LNC at execution time nor lower at compilation time is affected by the occurrence of the "begin" or by a class-A statement. Therefore, if the relation between LNC and lower is correct at the position where

the compound statement occurs, it is correct also for its first non-class-A statement.

In addition to occurring as a constituent of a compound tail, a statement can occur, according to [1], in four other syntactical positions:

1) as the statement following the "do" of a for-statement. During the analysis of the for-clause LNC has to point to the line on which the "for" occurs. Upon entering the for-statement this will be guaranteed by rule 1 for the generation of the LNC-setting macro, and after each execution of the statement following "do" by rule 2. For either LNC, lower and upper are not affected by the statement following "do" (when it is of class A), or LNC is undefined and lower = -1 (when the statement is of class B), or the stacked value of lnc $\leq$ lower $\leq$ LNC $\leq$ upper after translation/execution of the statement. Upon entering the statement following "do", both LNC at execution time and lower at compile time points to the line on which the "for" occurs and therefore rule 1 is applicable.
2) the statement occurs as the statement following "then" in a conditional statement. In this case both LNC at execution time and lower at compile time points to the line at which the "if" occurs and rule 1 is applicable.
3) the statement occurs as the statement following "else" in a conditional statement. Due to the fact that upon entering the translation of a conditional statement the value of lnc is saved in the stack and that before entering the translation of the statement after "else" the value in the stack and the value of lower are interchanged (see section 4) both LNC at execution time and lower at compile time points to the line on which the "if" occurs and rule 1 is applicable.
4) the statement occurs as procedure body. Upon entering the statement in this case, LNC is considered to be undefined at execution time, and lower at compile time is set equal to -1. If the statement differs from a dummy statement or a compound statement the LNC-setting macro is required and produced indeed.

With this last case all possible occurrences of a statement have been covered and therefore the analysis is completed.

REFERENCES

[1] P. Naur (ed), Revised Report on the Algorithmic Language ALGOL 60, Regnecentralen, Copenhagen, 1960.

SOME NOTES ON THE HISTORY OF ALGOL

W.L. van der Poel *)

This lecture can only be an anecdotic approach to the history of ALGOL. Its full treatment possibly warrants a longer story than can be given in one hour. Furthermore this lecture is not purely meant as a scientific effort to treat the history of ALGOL but as an attempt to lift the veil of what has happened in the Working Group 2.1 on ALGOL of the International Federation for Information Processing and for this occasion especially to talk about the rôle Van Wijngaarden has played in this group.

Let me first recall some of the basic facts of the early history of ALGOL. I shall not go into the period before 1962 when the ALGOL movement was purely an undertaking of a more or less well defined group of individuals. This resulted in the Report on the Algorithmic Language ALGOL 60 [1], written by 13 people under the editorship of Peter Naur. Shortly after its publication it became clear that there still were some gross errors left in addition to many more subtle ambiguities. In 1962 the original authors accepted that any collective responsibility which they might have with respect to the development, specification, and refinement of the ALGOL language would be transferred to a newly formed Working Group, installed by the Technical Committee 2 of IFIP. The result of this meeting was also the issue of a Revised Report on the Algorithmic Language ALGOL 60 (Ed. P. Naur) [2]. But all this can be read much better in the introduction to the Revised Report.

The Rome meeting did not count as a meeting of WG 2.1. This started in August, 1962 in Munich. After this a long and sometimes irregular series of meetings was held:

Sept. 1963 in Delft, The Netherlands. March 1964 in Tutzing, Germany. Sept. 1964 in Baden, Austria in conjunction with a working conference on Formal Language Description Languages. May 1965 in Princeton, U.S.A. .

---

*) Technological University, Delft.

Oct. 1965 in St. Pierre de Chartreuse near Grenoble, France. Oct. 1966 in Warsaw, Poland. May 1967 in Zandvoort, The Netherlands. June 1968 in Tirrenia near Pisa, Italy. August 1968 in North Berwick, Great Britain. December 1968 in Munich, Germany. Here the circle was closed and the Report on the Algorithmic Language ALGOL 68 was accepted for publication. After this further meetings have been held in Sept. 1969 in Banff, Canada. July 1970 in Habay-la-Neuve, Belgium. March 1971 in Manchester, Great Britain and the last one in August 1971 in Novosibirsk, USSR. Furthermore there was an informal but important meeting of a few people in spring 1966 in Kootwijk, The Netherlands.

The period from 1962 to 1965 was devoted to defining a subset for ALGOL 60 and for defining some basic Input/Output procedures for ALGOL 60. Although I know that some people do not agree, this period is in my own opinion not the most glorious period of the Working Group. It was more a kind of cleaning up of previous things and getting acquainted with our way of working. This period has been covered mainly in the form of 194 quotations from letters collected by R.W. Bemer, A Politico-Social History of ALGOL [3]. Although Bemer has given an outside view on the inner workings of WG 2.1 and that period certainly merits the view of an insider, I shall not dwell any longer before 1965.

The meeting in 1965 was some kind of turning point in the actions of the WG. We had our hands freed from ALGOL 60 and we could think on a new ALGOL. As we did not know under which year the new ALGOL would appear it was informally termed ALGOL X. This was going to be the short term goal. Possibly another more extended language could result later, which very unofficially was termed ALGOL Y. Randell gave the following definition:

> ALGOL X is a language which could be described, if necessary, in such a way that entities comprising the text of a program are completely distinct from the entities whose significance can be changed by the program. ALGOL Y is a name for a suggested successor to ALGOL X in which this distinction may well be removed.

Some individuals could perhaps have had another understanding of X and Y but the Randell definition has always been adhered to in the WG.

This Princeton meeting was a kind of churn of ideas. Many new ideas

were brought in and a lot of scattered ground work was done in subcommittees. Only one complete proposal of a language was on the table, i.e. EULER of N. Wirth [4]. In skimming through the papers I see the case clause emerging, I see fundamental proposals on operators, on the parameter mechanism, on basic concepts etc. But the chaotic state of affairs can perhaps be seen from a remark I made as chairman at that time:

> I am appalled at the lack of decisiveness of the committee. Having been present at the subcommittee meeting and previewing its report, there was a remarkable lack of decisions. Discussions centered around [many details followed here on small points only]. It seems to me that taking the restricted view is putting the clock backward and we revert to an efficient FORTRAN kind of computer. Even PL/I goes further. We must seriously ask ourselves: what do we want to accomplish? Where are we going? Do we want to compete with PL/I or do we really want to make a breakthrough by providing something more powerful and better defined than before. We have a few excellent reports on the table and what we do is losing our time on a number of small, perhaps incompatible issues of which it is not even known whether they can be combined. We must seriously consider our course of action and I would recommend that we will adopt one complete proposal as the guiding principle and then try to fit in a number of details which are missing or which have not yet been considered. When we go on with fighting over details first and when the main lines of the issue of ALGOL X are not fixed in principle, then ALGOL X will always, like the camel, be a horse designed by a committee.

Between our discussions of serious matters we were very well aware of our sometimes silly behaviour as was jocularly expressed in "working rules" by P.Z. Ingerman in the following way:

1) Whenever a point shows a danger of being clear, it shall be referred to a committee.
2) A subcommittee shall prepare two or more contradictory clarifications of the point referred to it.
3) These clarifications shall be reported by the individual members of the committee, no committee report having been achievable.

4) The clarifications shall be discussed until one of them is in danger of being understood, at which point return to 1).

There was always some time devoted to discussion of ALGOL Y. As source of inspiration two articles of Van Wijngaarden were used. One was called Generalised Algol [5] and delivered at the Rome conference, the other was called Recursive Definition of Syntax and Semantics [6] and was given at the Baden conference. The EULER language can be considered as an outgrowth of these ideas, as Wirth was one of Van Wijngaarden's pupils in the time he spent at Berkeley. Let's recall a little discussion in Baden on ALGOL Y.

Bauer: From Tutzing we agreed provisionally that vW's generalized ALGOL approach could do what we want in AY. Can it be unified with Böhm's lambda-calculus?

Van Wijng.: I don't know. Don't know if the operations I allow myself can match.

Bauer: How close are we to agreement?

V.d. Poel: Wirth and vW. are very close.

Bauer: Shall we exclude the lambda-calculus?

Garwick: Hopefully AY will be simpler than A60. How many symbols are required to define A60?

Van Wijng.: Don't know, but all would go on one sheet of paper.

Garwick: Then you consider it practical?

Van Wijng.: Yes.

Unfortunately there exist no informal minutes from the Princeton meeting, but one important decision was taken. Everybody was invited to write his approximation of a complete report for ALGOL X for the next meeting.

At the beginning of the St. Pierre de Chartreuse meeting we saw three volunteers who had done their job: the first was by Van Wijngaarden, Orthogonal design and description of a formal language [7]. Premature and preliminary edition, intended for use by IFIP WG 2.1 only it says on its cover. As only some 30 copies of it were produced this certainly is a collectors item nowadays. The second was from Niklaus Wirth, A Proposal for a Report on a Successor of ALGOL 60 [8]. This was also produced at the Mathematical Centre as Wirth was working there for some time. The third was from

Gerhard Seegmüller, A Proposal for a basis for a report on a successor to ALGOL 60 [9].

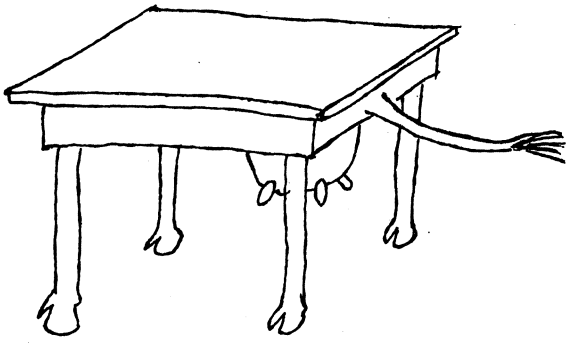
Furthermore there was an important paper on a topic called Record Handling by C.A.R. Hoare. This paper brought in many ideas on the way how to bring in the what we now call "structured values". From this report on Record Handling from Hoare I cite:

> For example the question "who is the mother of" is answered by asking the value of the reference field "mother" which is allocated for this purpose in the declaration of the record class for cows.
> In real life, most relationships are confined to holding between members of given classes; for example a house cannot be the mother of a cow, nor can a cow contain a table. ...

As a reaction to this Van Wijngaarden circulated the following rebuke around the table of which I happen to have a copy. This illustrated how a cow can contain a table and how an object can be both a cow and a table. He furthermore asks: "Can't a house be the mother of a cow if even a mountain can be the mother of a mouse?"

COMMENTS TO "RECORD HANDLING" BY C.A.R. HOARE

Page 5: Can't a cow contain a table?



Page 2: Can't an object be both a
a cow and a table?

Page 5:

Can't a house be the mother of
a cow if even a mountain can be
the mother of a mouse?

A. van Wijngaarden, 25.10.1965

This St. Pierre meeting can be considered as the true beginning of the new language. The first two formal resolutions taken at that meeting ran as follows:

1. That a subcommittee be set up, consisting of Hoare, Seegmüller, Van Wijngaarden and Wirth: the subcommittee to be charged to prepare a draft report from the existing material, mainly that which they have themselves presented, taking into consideration to the best of their knowledge and ability what the committee here has expressed as its wishes and views. The subcommittee should exist and hopefully work between this meeting and the next.
2. Whatever language is finally decided upon by WG 2.1, it will be described by Van Wijngaarden's metalinguistic techniques as specified in MR 76. [The orthogonal design etc.]

The method of description introduced in MR 76 consisted of a second level of production rules, the so-called meta rules, to produce in their turn the production rules of the language. So we recognise for example:

F sequence: F; F sequence, F.
G list : G; G list, comma symbol, G.

The meta syntactic notions are indicated here by single capital "syntactic marks". This made some of the rules terribly hard to read. As a result a request was made to make these meta syntactic notions more intelligible by using words for them. It is strange to note that in later meetings the opposite view was heard that the whole syntax description was too verbose, should be made shorter and could not be translated in foreign languages (natural languages).

Sentences such as:

> The text of the program is considered to be presented as an ordered sequence of symbols. This order will be called the lexicographic order. Typographical display features such as blank space and change to a new line do not influence this order.

can be found almost unchanged in the final Report on ALGOL 68.

At that time the Orthogonal Design made a distinction between three different kinds of minus, i.e. the "mines symbol" for the sign of signed

numbers, the "minus symbol" for the monadic negation operator, and the "minuss symbol" for the dyadic subtraction operator. APL kept this distinction between the sign of a number and an operator, ALGOL 68 dropped it later out of practical reasons.

Discussion went on on many topics such as exemplified in:

Hoare: The purpose of the for statement is to single out the common cases of loops, to provide convenient notation for such cases, and to help the reader realize this. Therefore I propose that the controlled variable be invisible outside the for statement and constant inside the controlled statement, and defined as by Wirth rather than by Seegmüller.

Bauer: I prefer Seegmüller's description.

...

Van Wijng.: We should allow programmers to omit parts of the step until element where the standard use was required.

Naur and Randell objected to this as being an instance of empty options.

Bauer: We should either allow all 8 possible omissions or no omissions at all.

Alan Perlis, one of the original ALGOL 60 report authors joined us as an observer in that meeting for the last time. I include here a few of his quotations.

Perlis: There are three main categories of programming, namely scientific computation, business data processing, and symbol manipulation. These are separate at least in as much as their practitioners are separate. We should propose an ALGOL X which is satisfactory for all three fields.

Another was during discussion of parallel processing:

Perlis: Your language allows parallel execution, but does not permit accurate description of the effects of different rates of execution.

As you will see, not all wishes ventilated in the WG were taken in the final report. The same is true for the next statement:

> Perlis: An implicit declaration of the controlled variable removes flexibility - for instance it cannot be other than single precision.

In general the St. Pierre meeting was a most constructive meeting. Naur introduced his notion of Environment Enquiries and also contributed a proposal for the introduction of the report, explaining its aims and purpose. Although we were always short of time and often worked during the evening time as well on subcommittees, there was a lot of fun and jokes. In my own notes I find a lot of funny jokes and quips of which I cannot always trace its author or inventor.

> Would you please call a syntaxi for me?
> We have forgotten to put Gargoyle on the fire!

(Gargoyle was a system writing language devised by Jan Garwick [21].) Another double bottomed saying (I think coming from Ingerman, who always was a maker of pun and fun) was:

> This language fills a much needed gap.

As a sideline I find here a nice definition of artificial intelligence.

> Artificial intelligence is the misusing of machines to act like human beings.

The four people designated by the committee to write a single report for the next meeting, i.e. Hoare, Seegmüller, Wirth and Van Wijngaarden actually came together between meetings in April '66 in Kootwijk Radio, Holland. At that time Barry Mailloux had joined the Mathematical Centre as a research student. Mailloux attended the Kootwijk meeting as an observer. I also could attend as an observer in my quality as chairman of WG 2.1. Very soon it became clear that the Kootwijk meeting would become a breakpoint of opinion between Hoare and Wirth on one side and Seegmüller and Van Wijngaarden on the other side. Hoare and Wirth had progressed between themselves so far in the direction of particular proposal which took the direction of the "diagonal approach" that it was very difficult to recon-

cile this with the "orthogonal approach". The orthogonal approach is the approach in which all possible combinations of two or more independent concepts were allowed, while the diagonal approach only would insert those possibilities in the language as were seen fit for some purpose. Such situation arose e.g. in the declaration of simple quantities as constants or as variables and in the same way declaring procedures as constants or variables. The same diagonality appears in the conformity relation, which under the diagonal approach would be made applicable only to records (as has been laid down in SIMULA 67, another offshoot of the record ideas of Hoare), but would also be applicable to other "modes" such as united modes in the orthogonal approach. Also the speed by which the different parties thought they could produce a final report was not agreed upon. Let me quote some passages from the subcommittee report.

> The other two members of the subcommittee (Hoare and Wirth) felt that their primary duty was to produce a report which WG 2.1 would have a good chance of accepting as ALGOL 66 [!], even if such a report should be inferior to one which might be accepted in the following year.
>
> ...
>
> In view of this fundamental disagreement on approach, it was agreed that Wirth and Hoare should proceed on the original plan to edit the most important of the unanimously agreed improvements into the "Contribution" and a summary of the changes is being sent to members of WG 2.1 for their consideration.
>
> ...
>
> All abbreviations in the metalanguage and metametalanguage should be replaced by full words of the English language.

The "Contribution to the development of ALGOL" by Wirth and Hoare [10] ultimately resulted in ALGOL W, developed at Stanford University [11]. This was taken out of the realm of activity of WG 2.1.

Barely before the meeting in Warsaw (which had already been postponed because of the enormous difficulties in turning out a document) a new proposal was sent around. This proposal did not yet contain any I/O procedures and had to be considered as incomplete. Nevertheless it was accepted at the meeting as the working document, commissioned from the subcommittee working

in Kootwijk. Hoare supported the document if it would not be simply rubber-stamped by WG 2.1, but he withdrew from the subcommittee. Wirth on the other hand was not able to come to Warsaw. He wrote a letter in which he stated that he was not prepared to discuss the report before Van Wijngaarden had fulfilled the task taken upon him in Kootwijk. He further thought that the document should be released if and when an implementation of the language had been proved to be practically possible.

The document as it was now on the table contained the parameter mechanism with the identity declaration as it is now in the Report on A68. But many things had not been invented yet. There were no coercions, there were no definable operators yet. Let me again quote from what was said. (These quotations are partially derived from the informal minutes of the meetings, partly from my own note book. The informal minutes were not formally accepted at the meetings, but they are pretty reliable, thanks to our different secretaries, R. Utman for Princeton and before, B. Randell for St. Pierre and W. Turski from Warsaw onward.)

In the introductory discussion on the just submitted new document:

Van Wijng.: I think that the delay in producing the final version may not be very long.

We still were optimistic at that time! The parameter mechanism was explained.

McCarthy: I could propose a new notation but I think I should do it in writing. Two questions more: 1. Can we have other constructions like quaternions in the language. 2. Could we do the list processing in the language.

Van Wijng.: Yes, to both questions.

...

McCarthy: Why cannot we have overloading as defined by Hoare in the NATO Summer School Notes. I advocate it!

Van Wijng.: I do not need this concept. I can do everything without it, via an appropriate procedure. ... The procedure is a much more fundamental thing. I am against having too many specialized things.

McCarthy: So it is a matter of taste. If I put a resolution for having overloading would we vote on it, or would it upset all other things. ... Could you, or any other expert, help me to work it out.

Van Wijng.: O.K.

We now know that it actually got in between Warsaw and Zandvoort. (Overloading was a term, which was used for operators, which had to be redefined for other data types such as matrices or quaternions.)

Woodger: Stop talking about notation. Overloading should belong to ALGOL Y.

But a few minutes later the same Woodger said:

Woodger: If overloading should be in ALGOL Y, why not in X. Why do we not put a good thing into ALGOL X when we find one, other than because of the fear that all good things would be put into ALGOL X and nothing will be left for ALGOL Y.

References always were a hot topic. Wirth had struggled with them before and had declared himself against several times.

Bauer: That was the problem of conceptual economy to merge references and references to records. Was it difficult to bring them together?

Van Wijng.: I recognize them as being the same thing. It did not cause any difficulty, on the contrary, it simplified the matters.

...

Landin: What if I want to remember whether the name of the variable to which I assigned the value had a "q" in it. ... I should have a mechanism of manipulating an identifier.

Van Wijng.: Not at all - the identifier is used for identification only, it has no inherent features. If you want your language to be conscious of the identifier structure please do so, but that is not ALGOL.

...

Van Wijng.: I will tell you the history of my thoughts. In Princeton, there was no talk on records and, of course, not on references and their restriction. In Grenoble, I did not include any records into the orthogonal language because I did not feel safe on these grounds. It has been decided, however, that we should include records into the language. We were thinking how to glue these things together, and the best way we could do it was adopted. I did not change my mind, my thoughts have envolved.

In this way these meetings went on. It is really very hard to swallow new thoughts from somebody else in another frame of reference. You must first map it back to your own frame of reference before you can really digest it.

S. Moriguti was our designated representative from Japan. But Japan is far away and the Japanese wanted rather to give a different person the opportunity to go to the meeting. So we had had in the previous meetings three times another representative for Mr. Moriguti. In Warsaw we had for the first time a fourth representative, Nobuo Yoneda. But he was quite different from all others. He mastered the English language fluently and he was one of the sharpest analytical minds in our group. At the table speeches on the closing banquet it was remarked that "Yoneda was the best Moriguti we ever had". He was going to stay as a member in his own right since then.

A few more quotations from the end of the meeting:

Van Wijng.: I want to add that the document published will include I/O. ... I do not expect that future differences in documents will be very big.

Again what an unwarranted optimism!

V.d. Poel: I think the document should be published in the ALGOL Bulletin and should be submitted to as many journals as will accept it free of charge.

Randell: By the time it comes out in, say, CACM, it will be obsolete.

McCarthy: As a piece of scientific information it will not be obsolete.

When discussing some points on structured values for which of course the example person was always chosen, I remember the following sentence enunciated by McCarthy:

McCarthy: Persons may be said to be cartesian products of some members and their father.

Historical!

At last the following formal resolution was taken:

a) The document identified as Warsaw 2 [this was the submitted document] be amended in the manner indicated in the discussion of this document. These amendmends will include at least the incorporation of the I/O Proposal and the addition of missing sections of explanation, motivation and pragmatics.

b) This amended document is to be published as a working paper of WG 2.1 in the ALGOL Bulletin and offered for publication to other informal bulletins. This working paper is not to be offered to any formal or refereed journal for publication.

c) The editing committee working on this document will take into particular account those weaknesses and deficiencies, if any, discovered in the course of implementation of the language.

Van Wijngaarden was then asked to act as editor, which he accepted. The period between Warsaw and Zandvoort was rather long, too long in the opinion of some members. I have gone into the previous meetings rather in a detailed fashion, because the basic principles were laid down in these early meetings. As the day of final acceptance drew nearer, more and more time was devoted to formal and procedural discussions. But let us first look at some quotations again.

Van Wijng.: If you recall the Warsaw meeting you should remember that I agreed to write the report but under the specific condition that no time pressure will be brought to bear on us. We have incorporated two new features: the Samelson's feature and the overloading. "We" in this context means myself and Messrs. Mailloux and Peck who worked as devils.

Indeed, the complete report was now reworked. The Samelson device was the handing in of parametrizable forms, or lambda forms as they are called in other languages, as actual parameters. The coercions also were invented by that time but they were by no means leak proof yet.

The old-fashioned notation real x now was an extension, a kind of abbreviation for ref real x = loc real, although the syntactic sugar tasted a little bit less sweet in these days. So a ref could be dropped sometimes. But real pi = 3.14 could not be extended. This prompted Seegmüller to ask the following question:

Seegmüller: When we go from strict into extended language ... we drop something in one case but not in the other. Is this not slightly misleading?

Mailloux: It is very simple. You just explain to people that = is a negative reference.

...

Bauer: Why to use the term "generator" and describe its action as creation?

Van Wijng.: It is difficult to find better terms. If you could give them to us we should be happy.

Bauer: The words you use are so ambitious.

Parallelism was discussed to great extent. There were "elementary" actions defined in the report (what now are called "inseparable actions") but pressure was exerted on the editing committee to insert the P and V operations of Dijkstra as the means of synchronization. The discussion was rather inconclusive and the dangers of only giving some quotations from the full discussions are great. Nevertheless I want to quote the following:

Randell: Dijkstra says that taking a value and assigning a value are the only two elementary actions.

Mailloux: We have already agreed to give you p and v.

...

Van Wijng.: ... We shall say we are not ignorant of the problem but the state of art is such that it does not yet permit for inclusion of parallelism in ALGOL (67). But, whatever will be the outcome of the research on parallelism, the concept

of elementary action and elementary symbol will be in, so let us let them in.

He was wrong in that statement. For a while the elem symbol stayed in together with the p and v operation, later called up and down operation. It was Niklaus Wirth who transmitted the message that not both concepts could stay in together, although he had left the committee as a member later. As a consequence the elem operation disappeared, but the elem symbol reappeared in a later version, now as the 'n-th element of' symbol.

Wirth: It would be funny to take a parallelism out, but only part of it.

Randell: It is like taking half a tooth out.

All this talk on parallel phrases made some of us invent a nickname for Fraser Duncan. He was called a parallel fraser.
At this moment I recall another anecdote. When planning the Warsaw meeting Bauer asked, why in Warsaw. What can you buy in Warsaw on Saturday. Answer: the same as on Friday. From the Zandvoort meeting I also have the quip:

If the bible had been written like this there would have been many less christians.

In the very beginning of the WG 2.1 on ALGOL there were attempts from the side of IBM to get a unified effort of developing FORTRAN and ALGOL. Later there has been an effort from SHARE, which developed the NPL (later being known as PL/I) to bring this New Programming Language and ALGOL X together. Actually both committees have exchanged observers at some time. McIlroy once attended our meeting as an observer in Baden and I have attended a SHARE meeting on NPL in Hursley. But the principles of designing a language were so far apart that these efforts soon bled to death. Not only the scientific starting points were different, but in particular the commercial viewpoint was different. How could a firm as IBM who felt responsible for the language PL/I put things in the hands of such an irresponsible bunch of scientists. On the other hand it must be said that the selling power of the IFIP is not always what it could or should be. We sometimes made the joke of saying: PL/I for the IBM CCCLX.

At the request of Yoneda there was an added syntactical chart to the

document. Yoneda himself produced several fine specimina in his very precise handwriting. One produced by Peck had the motto:

> People who like this sort of thing will find this the sort of thing they like. (Abraham Lincoln).

Speaking about motto's I always regret that one motto has disappeared from the final report, but figured in one of the numerous intermediate versions.

> Yes, from the table of my memory I'll wipe away all trivial fond records. (Hamlet, Shakespeare).

This motto headed the chapter on structured values, formerly called records after the idea of Hoare. The editors apparently thought it unkind to wipe away all fond memory of records.

Another very long year went by before we had the next meeting in Tirrenia, Italy. The editing committee had now grown to 4 people: Van Wijngaarden, Mailloux, Peck and C.H.A. Koster. Koster mainly worked on transput (i.e. Input and Output). Peck became the specialist in syntax and coercion, Mailloux worked out implementation [14], Van Wijngaarden was the party ideologist. In the mean time a "Draft report on the algorithmic language ALGOL 68" (Report MR93 of the Mathematical Centre, Amsterdam) [13] had been mailed in February 1968 as supplement to ALGOL Bulletin 26 to the subscribers of the AB. The Tirrenia meeting would be informal except for the last day, because of the $3\frac{1}{2}$ month rule of Zantvoort. Only so long after the mailing could the meeting be convened to give the proper opportunity to the members to read the revised document. But alas, nobody ever read the papers of anybody else in this committee. (Or is this true in other committees too?) Naur did not believe in committees any more as the stated in BIT [20]:

> A committee is a group of people unwilling to work, organised by other people incapable of doing so to do work which is probably useless.

Well, the editing committee certainly has not been unwilling to do work. If I only measure the height of the stack of iterations of documents I come to some 75 cm. And it may be true that a committee wastes hours, it keeps minutes.

The meeting in Tirrenia was about the last one where really technical matters were discussed. There were some strong objections to the publication of the Draft Report because some copies indeed had penetrated to refereed journals and even some copies were found for sale in a London bookshop. But all this was smoothed out. It was after all only distributed as an ALGOL Bulletin Supplement.

A point of discussion was the description method. Several times other methods of description for the same language were invited but no reports were submitted except for one from Duncan, which reverted to the use of angle brackets. It did not convince the majority that it really was another kind of description, instead of just another notation for the same thing. We often asked ourselves in how far the language to be defined was independent of the method of definition. Would not it be another language if the defining method is completely changed.

The MR93 certainly was difficult for the uninitiated reader. I quote here from a personal letter from Duncan of 25th March, 1968.

> ... In London we have been trying to get to grips with MR93. Landin has a fortnightly seminar, which the other 3 of us [Hill, Russell, Lasky?] usually attend. ... I think it is no exaggeration to say that a widespread opinion is that the document itself is extremely difficult to begin to understand (and unnecessarily so), but that inside it there may well be a good language trying to get out. Maar niemand wil een kat in de zak kopen! [Duncan knew Dutch]

There was growing a good deal of opposition to the document and the language. Here is a quotation from a letter of Dijkstra (undated! but my date is 2nd April, 1968):

> Motto [one of them]: "there are writings which are lovable although ungrammatical, and there are other writings which are extremely grammatical, but are disgusting. This is something that I cannot explain to superficial persons." [from Chang Ch'ao] ...
> Thank you for sending me MR93, which has absorbed a considerable fraction of my available mental energy since it is in my possession. [Dijkstra was seriously ill at that time] It must have been very hard work to compose it; alas, it also makes rather grim reading. The docu-

ment turned out like I expected it to be, only much more so.
The more I see of it, the more unhappy I become. I know it is a hard thing to say to an author who has struggled for years, but the proper fate of this document may indeed range from being submitted to minor corrections to being completely rejected. ...

Here is another reaction of H. Bekič of 23rd April, 1968:

... My first impression was that it is much richer ... much more complete ... and also more condensed than previous versions. ...
I cannot help deploring many of the reactions to the Report, even though, in a sense, I share them. It is an amazing question how it can be that a Committee which has charged you to do that work and has had the chance of watching the direction into which it moves and of voting on intermediate results, now produces such reactions; and I think it would be worthwhile to analyse this question from the Minutes or from some more complete private recordings. The main concern seems to be about matters of style, and of inderstandability. Now style is a very important thing, but very difficult to argue about. ...
... I for one find it difficult to get a really thorough and connected view of such a big thing like the ever-growing informal definition of PL/I, or our formal definition of it, or now your Report, and others may find themselves in a similar necessity to divide their energies.
...

Much of the critique came in directly to the Editor. These letters form an enormous stack together. In the same style as introduced in MR93 using two letter abbreviations as PP for Preliminary Prologue er EE for Ephemeral Epilogue, series of remarks from certain places got abbreviations too. E.g. AA for the Amsterdam Ameliorations, BB for the Brussels Brainstorms, CC for Calgary Cogitations (Peck was in Calgary again), MM for the Munich Meditations, LL for Landin's Laments and even greek letters such as $\phi\phi$ for the Philips Philisophies. The BB's have been issued later as Report R96 from the Manufacture Belge de Lampes et de Matériel Electronique, where four very active members were working: M. Sintzoff, P. Branquart, J. Lewi and P. Wodon. This report alone contains 197 BB's and is 2 cm thick [16].

A few quotations from the Tirrenia meeting:

[1:2] real b = (3.14, 2.78)

Goos: [The above clause] is undefined by the language.

Van Wijng.: It is not.

Goos: Then I can treat it, I can see it easily.

V.d. Poel: So you want to forbid only cases which you cannot see?

Goos: Yes.

Yoneda raised a new point and insisted that unions should be defined in such a way that they are commutative so that union (int, real) would be the same as union (real, int). They also should be accumulative. This has become one of the showcases of what could be done with the syntactic formalism but when you ask me personally, I still find it ugly and too complicated. But as usual, if the editors saw a way to satisfy the wishes of the members expressed in their voting, they tried to do it and they often succeeded.

The struggle for acceptance had begun, we neared completion and the technical content of the meeting went down, the formal matters going up.

Van Wijng.: I have been a long long time in the Algol Committee. I have had bad experience with producing working papers for WG 2.1. People have published what I couldn't publish (Orthogonal design). Therefore it is a fair request of the authors: If you like it, take it; otherwise, we publish. I have not fulfilled my task if you consider (what is not in the Minutes) the talks before closing the Warsaw meeting. This WG has worn out its first editor, Peter Naur. Then it has worn out two authors, Wirth and Hoare. If I understand right, it has worn out now four authors.

...

Bauer: Aad, [Van Wijng.] don't throw away the baby because the shoes don't fit. You want the committee to accept not only the language you have defined but also the peculiar form of description you have chosen for your definition.

...

V.d. Poel: Perhaps this is the last chance for a Committee to design a language.

At the beginning of the North Berwick meeting, I seriously considered to invite a psychologist as an observer to study the behaviour of this very peculiar group of scientists. If ever somebody thought that a language design could be made on reasonable grounds alone, he is mistaken. I have never seen so many emotional arguments being brought in as in this WG.

Gradually a dissident party could be discerned in the group. Ranging from "drop the whole thing" to "it should be more formally defined" the discussions were sometimes very chaotic. When discussing on in and out procedures, we found the appropriate terms: insane and outrageous. I find it very difficult indeed to give a clear account of the very subtle shades of opinions, which were sometimes ventilated in rather fierce attacks in words. The best I can do is still give some literal quotations, but I am aware of the fact that even the selection I had to make could give a partial impression. I can assure you that I found these last two meetings before the final acceptance the most difficult ones.

The last formal resolution of Tirrenia read:

> The authors are invited to undertake to edit a document based on MR93, taking into account the questions and remarks received before, on, and possibly after, this meeting to the best of their power in the time they can afford for this job. This document will be submitted to the members of WG 2.1 before 1st October, 1968. This document will be considered by WG 2.1. Either WG 2.1 accepts this document, i.e. submits it to TC 2 as Report on ALGOL 68, or it rejects it.

The first days of North Berwick were used up in a rather fruitless polling of opinion on the most important topics for the future. Among them were Maintenance of ALGOL 68, self-extending languages, primitives, abandonment of ALGOL 68, operating systems, conversational programming, shared data bases and so on. Many of these terms were only O.K. words and were not defined.

Dijkstra: Condensing of the interest is very interesting and promising but I would recommend to the members a bit of soul-searching to discover the extent to which they were lured by a number of the O.K. words. I am extremely verbally thinking and my thinking can be led astray for days by vague associations caused by O.K. words.

The possibility was discussed to have a minority report going with the document.

Van Wijng.: Is it really necessary to have the minority report? The ALGOL Working Committee prepared documents published in 1958, 1960, 1962 and 1964. On all these occasions there was no one who agreed in every respect with the documents. In many cases the precise formulation of the documents was not even known, but the names were attached. The voting on the Subset was on the verge, the minority was very substantial, yet there was no minority report.

Randell: It would be very nice to believe that the intersection of our opinions is to be published, it is obviously premature to believe that the minority report will be necessary. But the ruling that there is not going to be a minority report is as deplorable as I can imagine. ... There must be a vote in December. Until then the discussion about the minority report is premature.

...

Zemanek: We should take into account the effort undertaken.

Dijkstra: The amount of effort has no bearing on the successfulness. I cannot honestly see why we should take into consideration the amount of efforts put into work. Amount of efforts should bot influence the judgement, should not put pressure on us. I am still using mild expressions.

Zemanek: Sir, I know you think of blackmail. I am not putting any pressure on you.

Van Wijng.: I want to make my personal interpretation of Zemanek's statement. The amount of efforts was put into activity by the request of the members who were kind enough to attend last meetings. This puts some responsibility on the members who requested this effort. This does not put any responsibility on members who showed no interest. I would only like them to continue not showing any interest in the future.

Sometimes the atmosphere was nasty as you see. There was a kind of loss of trust as Randell expressed it.

Bauer: Can we go back to the idea of working parties? ... To me it seems that "primitive" and "self-extending" people could sit together, I do not know about the others. I hope that the Chairman has enough wisdom to help such parties to be created.

V.d. Poel: Do you suggest that these parties submit their results to the whole group or that they should have their own rules?

Bauer: I refuse to give you the answer.

Dijkstra: I am trying to picture such a liberal grouping. The group to which I would be most attracted would be less decided by the subject of the work and more by the attitudes of other members in such party.

Turski: Is it that you do not care what you do as long as you do it with whom you like doing it?

Dijkstra: Certainly not, but what you can achieve depends on the attitude as much as on the subject. I can be better cooperative in the group which is better suited to my slow-wittedness.

As we know now a new Working Group was formed later, called WG 2.3 on Programming tools. In contrast to WG 2.1 this group had not the task of developing ALGOL, that is, a definite language to be used.

Here I come across a Guiding Principle, invented during the coffee breaks, i.e. the Bauer Principle.

Ross: Do I understand that you have simple modules of the language, and the experimental ones?

Van Wijng.: We apply the Bauer Principle: who does not want to use complex facilities, does not pay for them. If the user wants to use them he has to pay a little.

Lindsey, who had written "ALGOL 68 with fewer tears" [22] was one of our new observers at that time and soon afterwards became a member.

Lindsey: ... The built-in operators, like + and -, should be imple-

mented in an efficient manner, not by procedures. ... The operator definitions certainly should not be permitted to be recursive.

...

Van Wijng.: Is it meant that this WG 2.1 is recursive in its decisions in the sense that we may undo decisions on which two years of work were based? If we accept this point of Lindsey we will produce a FORTRAN-like language, by this I mean its intellectual level.

Somewhat later the difficulties of storage administration for the heap were discussed.

Hoare: ... We are still exploring the areas of storage administration and the solutions are yet unknown. ...

V.d. Poel: There is no real problem in it. In the single level store the garbage collection problem is solved now.

Ross: I would dispute this.

At this moment we know for sure that it has been solved!

Another example how different attitudes and frames of thinking were popping up repeatedly was the "assignment operator" as some people called it. This is perhaps true for a typeless language such as LISP with only a built-in dereferencing of one step working uniformly on operands but it is not true in ALGOL 68 where "soft := strong" and "firm + firm".

Hoare: ... Inability to extend the definition of the assignment operator, as you can extend other operators, is responsible for many coercions built into the language.

Van Wijng.: I disagree because := is not an operator at all.

Hoare: I agree that you made things very asymmetric.

That's how life is! The relevant formal resolutions stated that minority reports could be part of any final document produced, but then they must be or have been submitted in writing to all members present at the meeting at which the final document was to be accepted.

That brought us to the last meeting in which ALGOL 68 had to be accepted or dropped. I had indicated my wish to resign as chairman after

seven years of office at the end of the Munich meeting. As I stated in my opening word:

V.d. Poel: I am very happy that we returned to Munich. I do not know whether it is symbolic, whether it is the end of our meetings, or the work is endless, cyclic.

As several refinements had been put in between North-Berwick and Munich (two more complete reprintings, labelled MR99 and MR100) there was again quite a lot of technical explaining going on. The case-conformity clause was invented.

Landin: I thought that the case clauses were some sort of nested *if* clauses!

Van Wijng.: Yes, but you first have to find the value of i [in *case* i in ...].

Landin: O yes, I see!

...

Van Wijng.: I would like to ask that at least point 2 [on additional clarification asked for and motivations] is continuously on the menu.

Dijkstra: I think I disagree with that.

Van Wijng.: But I want to have the substance matter continuously on the menu.

...

Seegmüller: The idea of hardware language is introduced very vaguely. What is the distinction between the representation language and the hardware language?

Van Wijng.: It depends on your reading equipment.

...

Seegmüller: I would not go as far as this. But I would try to be a bit more precise.

Van Wijng.: Why do not you go, sit in the corner, and make proper wording.

Seegmüller: O.K., I shall try.

At this time the idea of the II, the Informal Introduction to ALGOL 68 [18] was brought up. Lindsey and Van der Meulen had volunteered to under-

take such a work. Lindsey made a presentation of the lay-out of this work.

Seegmüller: A part of my suggestion was that the II should be published together with the defining document. I would like to repeat this point.

Lindsey: You will have to wait.

Seegmüller: Then I would rather delay the publication of the Report.

V.d.Meulen: You cannot write the introduction before the Report is closed.

V.d. Poel: There was no commissioning of this work, we are entirely in the hands of the authors of II; when they finish it, it will be published.

Now the negotiations began on the wording of a cover letter for the Report. This took a long time and all controversies were raised again.

Duncan: I am not of the opinion that the document describes a language. Another point is that I do not know why anybody should be interested in my opinion.

He was feeling very low apparently and I know why. I am not going to disclose that piece of information. I also have my professional secrets. Later on description methods, Duncan's against Van Wijngaarden's.

Duncan: The description method [of vW.] failed my tests. Another thing is that if my objections as a member of this Group are not taken, then what a chance do I stand as a member of the public.

V.d. Poel: Many of your objections were taken into consideration.

Van Wijng.: I made improvements according to your suggestions even before I received your letter.

Turski: A clear example of Extra Sensory Perception!

As an intermission to these little fights let me just tell you about another nice procedure which would in one stroke promote ALGOL X into an ALGOL Y. At one time the following procedure was proposed:

```
proc execute = (string progr): ¢ the string progr is considered as a
possible closed clause and elaborated at the textual position of the
call ¢;
```

This procedure could transform a sequence of characters into a closed clause, i.e. a syntactic notion. In other terms, it could invoke the compiler at run time. A simple operating system could now be written as:

do execute (read)

Execute as a program what you have read as a string and read the next program when you are ready. The proposal was not accepted. What a pity!

For the authors, who tried to create a milestone, found that it had become very much a millstone. They had to be careful that it would not become their tombstone.

The Munich meeting had sailed clear of a minority report but in the last half day it still happened. Signed by Dijkstra, Duncan, Garwick, Hoare, Randell, Seegmüller, Turski and Woodger such a minority report was handed in. For the text I have to confer to the ALGOL Bulletin, in which it was published. I rather quote here from the less accessible documents.

The meeting concluded with a number of formal resolutions:

1. Resolved that WG 2.1 recommends to TC 2 to create a working group on Programming Tool Requirements and to reconsider the membership of WG 2.1. This was proposed by Van Wijngaarden [!] and seconded by Dijkstra and taken by 35 in favour, 2 against. The creation of WG 2.3 informally was a fact. In a second resolution Lindsey and Van der Meulen were thanked for their initiative in producing II.
2. Resolved that the Chairman, with the assistence of the Secretary, shall transmit to all members of TC 2 a copy of MR100, together with the text of the agreed Covering Letter. Subject to the approval of TC 2, the authors shall submit copies of MR100, together with the agreed Covering Letter, at least to the following journals: Comm. ACM, The Computer Journal, Numerische Math. [17], Kybernetika, Calcolo, Revue d'AFCET. The authors may introduce all necessary corrections to MR100 before submission and at the proof-reading stage. This last vote was taken with 27 in favour and 2 against (8 abstained).

After this 4 more meetings have been held, now under a new chairman but an old member: Manfred Paul. The membership has changed and the topic has reverted back to very deepgoing technical discussions. Of course some errors have been found, but they were lying very deep and are of no con-

cern to the ordinary programmer. Several long felt desires have been proposed and are readied for inclusion in a Revised Report as stipulated in the Covering Letter. Also a rather full implementation proved many expectations to be true. It is not a big compiler, it is efficient [23].

I shall not go into these years. This is good for another jubilee and for another author. For the next chairman it could be "the only most important case" as he once said in another context. For me it were the "longest 7 years I ever had" to paraphrase another anonymous remark on the longest 5 minutes somebody ever had.

REFERENCES

[1] P. Naur, (Ed.) Report on the Algorithmic Language ALGOL 60. Regnecentralen, Copenhagen, 1960.

[2] P. Naur, (Ed.) Revised Report on the Algorithmic Language ALGOL 60. Num. Math. 4 (1963) 420-453 and in many other journals.

[3] R.W. Bemer, A Politico-Social History of Algol. Annual Review in Automatic Programming, 5 (1969) 151-237.

[4] N. Wirth and H. Weber, EULER, A Generalization of ALGOL, and its Formal Definition. Techn. Report CS20, Comp. Sci. Dept., Stanford University, 1965.

[5] A. van Wijngaarden, Generalized ALGOL. In: Symbolic Languages in Data Processing, Proc. of the Symposium by ICC, March, 1962. Gordon and Breach, 1962.

[6] A. van Wijngaarden, Recursive Definition of Syntax and Semantics. In: T.B. Steel (Ed.), Formal Language Description Languages for Computer Programming, Proc. of the IFIP Working Conference on Formal Language Description Languages. North-Holland Publ. Co., 1966.

[7] A. van Wijngaarden, Orthogonal Design and Description of a Formal Language. Report MR76, Mathematisch Centrum, Amsterdam, 1965.

[8] N. Wirth, A Proposal for a Report on a Successor of ALGOL 60. Report MR75, Mathematisch Centrum, Amsterdam, 1965.

[9] G. Seegmüller, A Proposal for a Basis for a Report on a Successor to ALGOL 60, Bavarian Acad. Sci., Munich, 1965.

[10] N. Wirth and C.A.R. Hoare, A Contribution to the Development of ALGOL. Comm. ACM, 9 (1966) 413-432.

[11] H.R. Bauer, S. Becker and S.L. Graham, ALGOL W Language Description, Report CS89, Comp. Sci. Dept., Stanford University, 1968.

[12] O.J. Dahl, B. Myhrhaug and K. Nygaard, SIMULA 67 Common Base Language. Publication No. S-2, Norwegian Computing Center, Oslo, 1968.

[13] A. van Wijngaarden (Ed.), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Draft Report on the Algorithmic Language ALGOL 68. Report MR93, Mathematisch Centrum, Amsterdam, 1968.

[14] B.J. Mailloux, On the Implementation of ALGOL 68. Doctoral Thesis, Amsterdam, 1968.

[15] P. Naur, Environment Enquiries. ALGOL Bulletin 18.3.9.1. Oct. 1964.

[16] M. Sintzoff (Ed.), P. Branquart, J. Lewi and P. Wodon, Remarks on the Draft Reports on ALGOL 68. Report R96, MBLE, Brussels, 1969.

[17] A. van Wijngaarden (Ed.), B.J. Mailloux, J.E.L. Peck and C.H.A. Koster, Report on the Algorithmic Language ALGOL 68. Num. Math. 14 (1969) 79-218.

[18] C.H. Lindsey and S.G. van der Meulen, Informal Introduction to ALGOL 68. North-Holland Publ. Co. 1971.

[19] J.E.L. Peck, ALGOL 68 Implementation. Proc. of the IFIP Working Conf. on ALGOL 68 Implementation, Munich 1970, North-Holland Publ. Co. 1971.

In this last work a very extensive list of further references is given to ALGOL 68 papers.

[20] P. Naur, Quotation from BIT 9 (1969) 250.

[21] J. Garwick, Gargoyle User's Manual. Report S-24, Norwegian Defence Research Establishment, Nov. 1965.

[22] C.H. Lindsey, ALGOL 68 with fewer tears. ALGOL Bulletin 28.3.1. July 1968.

[23] P.M. Woodward and S.G. Bond, User's Guide to ALGOL 68-R. Royal Radar Establishment, Malvern, England, 1971.

THE CHANGING COMPUTER SCENE
1947 - 1957

M.V. Wilkes *)

SUMMARY

At the beginning of 1947, few people had heard of the Automatic Sequence Controlled Calculator at Harvard or of the ENIAC at the Moore School of Electrical Engineering in Philadelphia; even of those, not many appreciated with any clarity the part that digital computers would come to play in the world of scientific computing. The computing scene was dominated by desk machines and mathematical tables, with here and there primitive punched card installations. The emphasis was on making the best use of commercially available calculating machines, and much ingenuity was devoted to the exploitation in the context of scientific computing of machines developed for commercial accounting. There was also the analogue lobby, which pressed the claims of differential analysers and linear equation solvers of various kinds to be considered as the instruments destined to play a major role in the scientific computing of the future.

I had returned to Cambridge after war service about a year before, and the Mathematical Laboratory was getting into its stride. We were deeply involved in analogue computing and the EDSAC project had just begun. There was only a small amount of conventional computing going on in the Department. It was at this time that I first heard of the newly-formed Mathematical Centre. Professor van Wijngaarden, on his appointment to the Centre, wished to spend a period in England and flattered us by choosing the Mathematical Laboratory as his base. After his return to Amsterdam, the Mathematical Centre, on his advice, decided to concentrate on the organization of a computing department using the best desk calculating machines and methods available at the time. This department was very successful, and soon became well known in computing circles.

The early stored-program digital computers began to work in 1949, and

---

*) University of Cambridge

those professionally interested in computing gradually began to take notice of them as their powers were demonstrated in one application after another. It was appreciated from the beginning that computers could manipulate symbols and, looking back, one can see in work that then went on the germs of much modern work in artificial intelligence, algebra, man-machine relations, and so on. However, at that period, no-one was under any doubt that what computers were really for was numerical computation. Few of the early programmers, however, had any skill or experience in numerical analysis or in computing techniques, and the responsibility for educating them fell on groups such as those at the Mathematical Centre. Inevitably, as the computer field mushroomed, there were times when these groups felt that they were being overwhelmed. Looking back, however, one can see that the present computing world owes a great deal to the small number of people who were already experts in computing before digital computers came along.

The Mathematical Centre, however, was not content with desk computing, and also played a part in the development of digital computers. Two computers, the ARRA and the ARMAC, were built in Amsterdam during the period under review. The project that led to the production of the Electrologica X1 computer was also started at the Mathematical Centre. The significance of this computer is that it was the first commercial computer to have a modern interrupt system.

One of the signs of a born numerical analyst is a sharp eye for error. In early work with the EDSAC, the group in Cambridge had devised a subroutine for calculating digit by digit the binary value of an inverse cosine. This occupied only 33 half-words of memory, a point very much in its favour at a time when the EDSAC had only 256 words in all. The subroutine appeared in a laboratory report and was later published in a book [1]. The algorithm was as follows:

To find a, where cos a = x.
Form the sequence x_i such that

$$x_1 = x$$

$$x_i = 2x_{i-1}^2 - 1 \qquad i > 1.$$

Let a_i := if $x_i < 0$ then 1 else 0 $\qquad i > 0$

$$b_1 := a_1$$

$$b_i := \text{if } a_i \neq b_{i-1} \text{ then 1 else 0} \qquad i > 1.$$

Then b_i are the binary digits of a/π.

(I hope that the anachronism of using ALGOL notation in describing an algorithm of 1950 will be excused.)

In a paper presented at a conference in Paris in January 1951, van Wijngaarden pointed out that this algorithm is defective [2]. For general values of a it gives accurate results but if one or more of the functions cos 2a, cos 4a, cos 8a, has a value near unity, then there can be a serious error. The 'zones of danger' in which this occurs are very narrow, and this is the reason why the test that we had made of the routine with about 100 randomly chosen numbers failed to reveal them. As soon as van Wijngaarden drew our attention to the existence of these zones, we of course removed the subroutine from the EDSAC library. That, however, was not the end of the matter. The method was later re-discovered by D.R. Morrison, who described it in a paper published in Mathematical Tables and other Aids to Computation in 1956 [3]. Wheeler and I wrote a short note, published in a later issue of the same journal, pointing out the treacherous nature of the algorithm [4]. Possibly other people have later fallen into the same trap, although, as soon as digital computers came to have more than a few hundred words of high speed memory, the use of digit by digit methods ceased to be so attractive.

The subroutine in question is reproduced below from reference 1. Those who were familiar with EDSAC programming will view it with nostalgia; others may find it an interesting puzzle.

T4 Inverse cosine.

0.5 arc cos 2 C(4D) to 0D where $0 \leq C(4D) \leq 1/2$. Proceeds by finding successively the sign of 0.5 cos $2^n C(4D)$ formed from 0.5 cos $2^{n-1} C(4D)$ by $x_n = 4x_{n-1}^2 - 1/2$. The required result is built up digit by digit, using a negative strobe.

		G		K	
	0	A	3	F	
	1	T	28	θ	plant link
	2	T		D	
	3	A	32	θ	
20 →	4	T	6	D	strobe in 6D
	5	H	4	D	⎡
	6	V	4	D	⎢
	7	L	1	F	⎢ form $x_n = 4x_{n-1}^2 - 1/2$
	8	S	29	θ	⎢
	9	Y		F	⎣
	10	E	16	θ	test sign of x_n
	11	T	4	D	
	12	S		D	⎡
	13	A	6	D	⎣ form partial sum
	14	T		D	
	15	S	4	D	
10 →	16	T	4	D	
	17	A	6	D	⎡ shift strobe
	18	R		D	⎣
	19	Y		F	⎡ test for end of cycle
	20	G	4	θ	⎣
	21	H		D	
	22	N	30$\pi\theta$		multiply by $\pi/4$
	23	Y		F	
	24	E	27	θ	⎡
	25	T	4	D	⎢ take modulus
	26	S	4	D	⎣
24 →	27	T		D	
	28	(E		F)	
	29	‖I		F	= 1/2
		T	30πZ		⎡
	31	‖D	888	F	⎢ places $-\pi/4$ in 30$\pi\theta$
		T	30	Z	⎢
	30	‖O	699	D	⎣
		T	32	Z	
	32	‖K	4096	F	= -1

REFERENCES

[1] Wilkes, M.V., D.J. Wheeler and S. Gill, The Preparation of Programs for an Electronic Digital Computer, Addison-Wesley Press, Cambridge, Mass 1951, p. 152-3.

[2] Van Wijngaarden, A., Erreurs d'arrondiment dans les Calculs Systématiques, Centre National de la Recherche Scientifique, Colloques Internationaux, 37, 1953, p. 285.

[3] Morrison, D.R., A method for computing certain inverse functions, MTAC, 10, 1956, p. 202.

[4] Wilkes, M.V. and D.J. Wheeler, Note on a method for computing certain inverse functions, MTAC, 11, 1957, p. 204.

TWENTY-FIVE YEARS OF MATHEMATICAL PROGRAMMING

G. Zoutendijk *)

INTRODUCTION

In 1972 it will be 25 years ago that George B. Dantzig developed the simplex method for linear programming. Although a linear programming model for an economic problem had been developed 8 years earlier, in 1939, by the Russian mathematician L. Kantorovich, his work was not known outside the Sovjet Union until about 1955, while it was forgotten inside the Sovjet Union. Moreover it did not lead to a mathematical theory or to numerical methods. The year 1947 can therefore be considered as the origin of mathematical programming.

Mathematical programming is concerned with the problem of maximizing (or minimizing) functions under constraints. Since it originated it has grown in width and in depth. Nowadays it can be considered as a separate branch of applied mathematics, like mathematical statistics, having its own mathematical theory, computational methods and applications. Its mathematical basis is mainly in the theory of linear inequalities and in the theory of convex sets and functions. These fields have benefited tremendously from the development of mathematical programming. The major applications are in planning and scheduling problems (the allocation of scarce resources) and in technical design and control problems. The major characteristics of the former class of problems are size and simplicity. There are many relations of a relatively simple nature between many variables. Problems with thousands of constraints and variables are no exception. As long as it is a linear programming model of that size for a practical planning problem it can be solved but even the largest and fastest computer available to-day will have a hard time. With nonlinear constraints or integer valued variables many of these problems cannot be solved with present

---

*) University of Leiden

day computational methods and machines. Solving large mathematical programming problems has become an important and time-consuming job for many computers in industrial or government computing centres. The development of efficient algorithms for this has resulted in important contributions to numerical mathematics. Numerical analysis has benefited in another way too: mathematical programming methods have been applied to the solution of some well-known numerical problems: solving systems of nonlinear equations, least squares approximation under constraints, discrete as well as continuous Chebyshev approximation, solving initial or boundary value problems in differential equations and optimal control problems. The nonlinear programming problems which arise from technological models in design and control are usually much smaller with constraints of a more complicated nature. It is not surprising that for efficiency reasons other methods are often needed in this case. This need has for instance resulted in a number of efficient algorithms for the unconstrained optimization problem, another problem in numerical analysis that took advantage from mathematical programming.

Mathematical programming is still growing rapidly. A number of 7 international symposia have been held - the last one in The Hague in September 1970 with more than 500 participants -, together with numerous other conferences, summer schools, seminars, etc. A journal "Mathematical Programming" has just come into existence; the decision to establish an international society on the subject has been taken and a secretariat is being set up.

In this paper a far from complete survey will be given of some of the more important developments during the past 25 years. About in the middle of this period - at the Rand symposium in Santa Monica, California, in 1959 - three important new developments have been presented that stimulated research in the subsequent decade: the decomposition principle by Dantzig and Wolfe for large linear programming problems, the cutting plane methods for pure integer programming by Gomory and the methods of feasible directions for nonlinear programming by Zoutendijk. More recent important results have been obtained in unconstrained optimization, in the use of penalty functions to cope with nonlinear constraints and in branch and bound methods for mixed integer programming and related problems.

LINEAR PROGRAMMING

In its standard form the linear programming problem can be written as:

$$(1) \qquad \text{Max}\{p^T x \mid Ax \leq b,\ x \geq 0\}$$

with A m by n, p and $x \in R_n$ and $b \in R_m$.
In the standard form $b \geq 0$ will also hold. Deviations from the standard form like equality constraints, unrestricted variables or negative elements in the vector b can be easily coped with by applying artifices.

The principal mathematical theorem is the duality theorem. If the dual problem is defined by:

$$(2) \qquad \text{Min}\{b^T u \mid A^T u \geq p,\ u \geq 0\}$$

then with

$$(3) \qquad y = b - Ax \geq 0 \quad \text{and} \quad v = A^T u - p \geq 0$$

we have for the optimal solutions $\hat{x}$ and $\hat{u}$:

$$(4) \qquad p^T \hat{x} = b^T \hat{u}, \quad \hat{u}^T \hat{y} = 0, \quad \hat{v}^T \hat{x} = 0,$$

provided the primal problem (1) has a solution.

The simplex method for the solution of (1) performs a vertex to vertex path in the convex polyhedron defined by the constraints with non-decreasing values for the objective function. Since a vertex corresponds to a division of the set of all n+m variables x_j and y_i into a set of m basic or dependent variables and a set of n non-basic variables and two adjacent vertices differ in one different variable being basic this entails that a sequence of pivotal operations is performed like in the Gauss-Jordan method for solving a system of linear equations. The pivot column will be determined by the requirement that we want to make as much progress as possible (at least on an infinitesimal scale); the pivot row will then be determined by the requirement that none of the basic variables becomes negative when the non-basic variable just selected is being increased from zero.

Finiteness is ensured if an anti-cycling precaution is taken, like the one devised by Wolfe. The dual problem will be solved at the same time. We could therefore apply the simplex method to the dual problem instead and would obtain the primal solution at the same time. The dual simplex method is actually doing this, however without dualizing first but working within the organizational scheme of the primal method.

Many new developments have been reported and applied in linear programming in the last 25 years. We will mention just a few:

1. Computational algorithms

The most efficient and most commonly used algorithm is the so-called product-form algorithm in which the only information to be stored are the original matrix A and the vector p, the transformation vectors (transformed updated pivot columns) and the transformed b-vector. From these data all information required during the calculations can be reconstructed and - at least for large problems with a low percentage of non-zero's in the matrix - in a far more efficient way than in the straightforward algorithm used in hand calculations. Frequent re-inversions of the basis will be necessary and useful, so that a lot of attention has been given to efficient re-inversion techniques making as much use of the structure of the matrix as possible. Recently a product-form representation for the inverse of an LU decomposition of the basis B has been successfully applied.
Another important aspect of a linear programming computer code are the heuristics in it like near-zero criteria, special tricks to reduce the number of iterations, ways to break up ties during the pivot selection, re-inversion criteria, etc. By properly adjusting these heuristics to the problem solution time can often be decreased considerably. This aspect will probably receive more attention in the future.
It is also being realized nowadays that in order to use linear programming in a routine way for large problems more is needed than an efficient algorithm: automatic matrix generation from primary data, automatic scaling techniques and report writers. This all belongs to a well-designed linear programming package.

2. Special LP problems

Special methods have been designed for special linear programming problems. Most well-known are transportation problems and, more generally, network flow problems but there are other examples as well. General linear programming algorithms have been adapted to problems with simple or, more recently, generalized upperbounds. An example of a generalized upper bounds problem is:

$$\text{Max} \sum_{k=1}^{r} p_k x_k, \quad \text{subject to}$$

$$\sum_{k=1}^{r} A_k x_k \leq b, \quad x_k \geq 0 \quad \text{and}$$

$$\sum_{j=1}^{n_k} (x_k)_j \leq 1, \quad k = 1,\ldots,r$$

in which A_k is m by n_k, $x_k \in R_{n_k}$ and $p_k \in R_{n_k}$. The latter set of relations need not be stored and transformed separately, resulting in shorter transformation vectors and a considerable reduction in computing time.
Parametric programming and other postoptimal analysis routines can also be considered as special linear programs.

3. Large scale linear systems

Many attempts have been made to make an efficient use of the special structure which these systems always show. The general methods - applicable to any linear program - can be divided into decomposition and partitioning methods. To briefly explain two of these methods - primal partitioning and dual decomposition - we will write the problem in the form:

$$\begin{aligned} &\text{Maximize} && p^T x + q^T w \\ &\text{subject to} && Ax + Qw \leq b, \quad x \geq 0, \quad w \in W \end{aligned} \tag{5}$$

in which W is a convex polyhedron in the appropriate space.
A primal partitioning method works along the following lines:

(1) Fix $w = \bar{w} \in W$.

(2) Solve the LP problem

(6) $$\text{Max}\{p^T x \mid Ax \leq b - Q\bar{w}\}$$

Note: In many practical cases this problem can be subdivided in a number of independent subproblems.

(3) Adjust the "decision variables" w_k as much as possible under the assumption that the final division into basic and non-basic variables in (6) is a correct one, i.e. solve the adjustment problem

(7) $$\text{Max}\{(q^T - \bar{p}_B^T B^{-1} Q)w \mid B^{-1}Qw + \bar{x}_B = B^{-1}b,\ w \in W,\ \bar{x}_B \geq 0\}$$

in which B^{-1} is the final inverse in (6), $\bar{x}_B$ the set of final basic variables and $\bar{p}_B$ the corresponding coefficients of the objective vector.
This gives a new estimate for w and new values for the $(\bar{x}_B)_i$, including one or more zero's.

(4) Perform an optimality test by investigating dual feasibility.

(5) If not optimal, then we have to make a change of basis in (6) (with new values for $\bar{w}$). This is usually done by first determining a bottleneck among the basic variables $(\bar{x}_B)_i$ and by then applying the dual simplex pivot selection criterion in the chosen pivot row. This will remove the bottleneck.

(6) Repeat (3) - (5), etc.

It is clear that with this approach a special structure of the matrix like

$$\begin{pmatrix} A_{11} & 0 & 0 & Q_1 \\ 0 & A_{22} & 0 & Q_2 \\ 0 & 0 & A_{33} & Q_3 \end{pmatrix}$$

will be maintained. Moreover the subproblems in (6) may be of a simple form, e.g. may be network flow problems.

The same advantages hold for Benders' decomposition method which is dual to the one developed by Dantzig and Wolfe. To explain this method we again assume that the problem under consideration is of the form (6). This can be rewritten as:

$$(8) \qquad \hat{w}_0 = \operatorname*{Max}_{w} \{w_0 \mid w_0 \leqq q^T w + \max_{x}\{p^T x \mid A_x \leqq b - Qw,\ x \geqq 0\},\ w \in W\}.$$

The method then proceeds as follows:

(1) Start with $w^0 \in W$, $h = 0$.

(2) At step h solve the LP problem:

$$(9) \qquad \text{Max}\{p^T x \mid Ax \leqq b - Qw^h,\ x \geqq 0\},$$

preferably by using the dual simplex method. There are two possibilities:

a. Problem (9) is inconsistent. In that case there is an extreme ray in the dual feasible region. In the final tableau we will find this ray r and $r^T(b-Qw^h) < 0$ will hold. Assuming consistency in (5) or (8) we will therefore require in the masterproblem

$$(10) \qquad r^T(b-Qw) \geqq 0.$$

b. Problem (9) is consistent, hence has a solution x^h and dual solution u^h. We have according to the duality theorem:

$$\hat{w}_0 \geqq p^T x^h + q^T w^h = (u^h)^T(b-Qw^h) + q^T w^h.$$

Since u^h is dual feasible for any w, hence also for $\hat{w}$, it follows that

$$(u^h)^T(b-Q\hat{w}) + q^T\hat{w} \geqq (\hat{u})^T(b-Q\hat{w}) + q^T\hat{w} = \hat{w}_0.$$

We will therefore require in the next masterproblem:

$$(11) \qquad (u^h)^T(b-Qw) + q^T w \geqq w_0.$$

(3) Solve the master problem:

(12) $\quad \text{Max}\{w_0 \mid w \in W;$ all relations of type (10) or (11)$\}$;

(4) If not optimal, repeat steps (2) - (3) for w^{h+1}, the solution of (12), etc.

Each new relation of type (10) or (11) cuts off part of the dual feasible region; the dual decomposition method is a cutting plane method. Its big advantage is that it can also be applied to mixed problems, for instance mixed integer problems (w integer valued) or mixed linear-nonlinear problems (W a region determined by nonlinear constraints). It then subdivides the original problem in a linear part and a pure integer or pure nonlinear part.

4. Methods of feasible directions for linear programming.

Although these methods have primarily been developed for nonlinear programs they can as well be applied to linear programs, while a first explanation can best be given for this simple case. A method of feasible directions assumes a feasible point $\bar{x}$ to be available:

$$\bar{x} \in R = \{x \mid Ax \leqq b,\ x \geqq 0\}.$$

It then repeatedly performs the following operations:

(1) First find a feasible and usable direction $\bar{s}$ in $\bar{x}$, i.e. a vector satisfying the requirements:
(a) $s \in S(\bar{x})$, the cone of feasible directions in $\bar{x}$,

(13) $$S(\bar{x}) = \{s \mid a_i^T s \leqq 0 \text{ if } \bar{y}_i = 0 \text{ and } s_j \geqq 0 \text{ if } \bar{x}_j = 0\}.$$

The direction vector therefore has to satisfy a number of homogeneous linear relations.

(14) (b) $p^T s > 0$.

This means that we will make progress in the direction s.

(2) Having found $\bar{s}$ the steplength is determined by

(15) $$\lambda' = \max\{\lambda \mid \bar{x}+\lambda\bar{s} \in R\}.$$

(3) $\bar{x}$ (new) = $\bar{x}$ (old) + $\lambda'\bar{s}$.

The point $\bar{x}$ will be an optimal solution if no s can be found satisfying (13) and (14), hence if $\forall s \in S(\bar{x})$: $p^T s \leq 0$. It then follows from Farkas' theorem that

(16) $$\bar{x} \text{ maximum} \iff \exists \bar{u} \geq 0,\ \bar{v} \geq 0,\ p = A^T\bar{u}-\bar{v},\ \bar{u}^T\bar{y} = 0,\ \bar{v}^T\bar{x} = 0$$

from which the duality theorem follows immediately.

There are many different direction generators, i.e. methods to solve (13) and (14). They can be divided into two classes:

(1) Direction generators which solve the subproblem:

(17) $$\text{Max}\{p^T s \mid s \in S(\bar{x}),\ s \in T(\bar{x})\},$$

where $s \in T(\bar{x})$ indicates an additional set of restrictions to avoid infinite solutions like $||s|| \leq 1$ for any chosen norm (e.g. the Chebyshev norm $\forall j \quad -1 \leq s_j \leq 1$).

(2) Direction generators which try to find a feasible solution of the system of linear relations:

(18) $$s \in S(\bar{x}),\quad p^T s = 1,$$

by dividing the set of all variables (including the slack variables) into a set of basic and a set of non-basic variables such that (18) is satisfied.

Depending on the direction generator chosen we obtain a different method of feasible directions. Most direction generators lead to linear programming subproblems.

INTEGER AND MIXED INTEGER PROGRAMMING

The pure integer programming problem:

$$\sum_{j=1}^{n} a_{ij}x_j + y_i = b_i, \quad x_j \geq 0, \quad y_i \geq 0 \quad \text{and integer valued}$$

(19)

$$\sum_{j=1}^{n} p_j x_j \quad \text{to be maximized}$$

can be solved by first solving the corresponding linear program with upper bounds added to it (for instance, if $x_1 = 0,1,2,$ or 3, then we add $0 \leq x_1 \leq 3$).

This will give a final tableau:

$$\sum_{j=1}^{n} a_{ij}^{*}x_j^{*} + y_i^{*} = b_i^{*} \quad \text{plus a reduced cost row} \geq 0. \tag{20}$$

If x_j^{*} and y_i^{*} are integer valued we have obtained the optimum solution of (19). Otherwise we choose one of the relations with b_i^{*} fractional, say

$$\sum \alpha_j x_j + y = \alpha_0 \qquad \text{(the * has been omitted).}$$

We have

$$\alpha_j = [\alpha_j] + f_j, \quad 0 \leq f_j < 1, \quad j = 0,1,\ldots,n,$$

so that

$$\alpha_0 - \sum\alpha_j x_j - y = [\alpha_0] - \sum[\alpha_j]x_j - y + f_0 - \sum f_j x_j = 0.$$

Since $\sum f_j x_j \geq 0$ it follows that $[\alpha_0]-\sum[\alpha_j]x_j - y + f_0 \geq 0$.

Consequently $[\alpha_0] - \sum[\alpha_j]x_j - y \geq 0$ and integer must hold for integer valued x_j and y. For, if ≤ -1 the former inequality will not hold ($0<f_0<1$).

It follows

$$\alpha_0 - f_0 - \sum(\alpha_j - f_j)x_j - y \geq 0, \quad \text{integer or}$$

$$-\sum f_j x_j + s = -f_0, \qquad s \text{ integer.}$$

This relation is not satisfied by $x_j = 0$, so that we have cut off part of the feasible region. The relation will be added to the LP subproblem which will then be solved again. The choice of the y_i^* with fractional value can be made in such a way that a finite cutting plane method is obtained.

In addition to this method of integer forms, suggested by Gomory, there are other methods working along similar lines. Among them Gomory's all-integer integer programming method which in the case of integer a_{ij} and b_i leads to all-integer intermediate tableaus by restricting the choice of pivot elements to elements being -1. There have been many proposals for more efficient cuts. There have been many experiments but it still holds true that none of the methods is really satisfactory in the sense that it always works within a predictable time for problems up to a certain size.

Therefore it is not surprising that many attempts have been made to devise enumeration algorithms, especially for the 0-1 problem, i.e. a problem of type (19) in which the variables x_j are required to be 0 or 1 (no restriction on the y_i). In these methods it is tried by using a great number of exclusion rules to restrict the number of possibilities to be investigated (out of the total number 2^n) to such an extent that total computing time becomes acceptable.

Let us have a partial solution:

$J_0 = \{j \mid x_j = 0$ has already been assigned$\}$,

$J_1 = \{j \mid x_j = 1$ has already been assigned$\}$ and

$J_2 = \{j \mid x_j$ not yet assigned$\}$,

then examples of simple exclusion rules are

(1) if $\exists i \, \forall j \in J_2$, $a_{ij} \geq 0$ and $b_i < 0$, then reject the partial solution;

(2) if we know a feasible solution $\bar{x}_j$ (=0 or 1 for all j) and if $\sum_{j \in J_1} p_j + \sum_{j \in J_2} (p_j \mid p_j > 0) < \sum p_j \bar{x}_j$, then reject the partial solution;

(3) if $\exists i$: $\sum_{j \in J_2} (a_{ij} \mid a_{ij} < 0) > b_i - \sum_{j \in J_1} a_{ij}$, then reject the partial solution.

If no rejection can be made a branch must take place. The dichotomy need not be simple but can for instance be of the form: either $x_3 = 0$ and $x_7 = 1$ or $x_3 = 1$, $x_7 = 0$, $x_1 = 1$, $x_5 = 0$.
A sophisticated treesearch algorithm can become quite complicated. However, problems with 60-80 0-1 variables can then be solved, whereas the cutting plane methods will sometimes solve much larger problems but will at other times fail to solve a much smaller problem.

By using the dual decomposition method as explained earlier in this paper a mixed integer programming problem can be attacked. The efficiency of this approach will depend on the efficiency of the pure integer method to solve the master problem (12) as well as on the efficiency of the information interchange between subproblem and master problem. More large scale experimentation is certainly required. Far more practical experience has been obtained with branch and bound methods to solve (5). ($w \in W$ will now be integer valued). This approach, originally suggested by Land and Doig, will be outlined for the 0-1 case (hence $w_k = 0$ or 1).

First we solve the linear programming problem:

$$\text{(21)} \qquad \text{Max}\{p^T x+q^T w \mid Ax+Qw \leq b,\ x \geq 0,\ 0 \leq w \leq 1\}.$$

If this leads to integer valued w_k we have solved the problem. Otherwise we select one of the w_k at fractional value and solve the two subproblems resulting from adding the requirement $w_k = 0$ or $w_k = 1$. This way a tree is constructed. Some of the questions in this respect are:

(1) which variable at fractional value can best be taken;

(2) what should be the treediscipline, i.e. at which node shall we continue our search, the last one obtained or the one with the highest value or still another choice;

(3) are there other dichotomies leading to more efficient branching such as for instance either $w_3 = 0$, $w_6 = 1$, $w_8 = 0$, $w_{11} = 1$ or $w_3 + (1-w_6) + w_8 + (1-w_{11}) \geq 1$.

Extensive experimentation has led to answers to some of these questions which, due to the proprietary nature of many computer codes, are not

generally known.

Well-known examples of mixed integer programming problems are:

(1) The fixed charge problem

$$\sum_{j=1}^{n} a_{ij}x_j \leqq b_i, \qquad i = 1,\ldots,m;\ x_j \geqq 0$$

$$\sum_{j=1}^{n} p_j x_j + \sum_{j=1}^{n} q_j w_j \qquad \text{to be maximized}$$

in which $w_j = 0$ if $x_j = 0$, $w_j = 1$ if $x_j > 0$. The $q_j \leqq 0$ are fixed charges.

By adding relations of the type

$$0 \leqq x_j \leqq c_j w_j$$

with c_j being a known upper bound for x_j the problem can be formulated as a mixed integer problem.

(2) The warehouse allocation problem, i.e. the problem of selecting a number of plant locations from m possible sites. At site i there will be a fixed charge q_i and a capacity a_i. Demands b_j at n destinations are given as well as unit transportation costs. Total cost should be minimized. The mixed integer programming formulation for this problem is as follows:

Minimize $$\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{m} q_i w_i$$

subject to

$$\sum_{i=1}^{m} x_{ij} \geqq b_j, \qquad j = 1,\ldots,n;$$

$$\sum_{j=1}^{n} x_{ij} \leqq a_i w_i, \qquad i = 1,\ldots,m;$$

$$x_{ij} \geqq 0,\ w_i = 0 \text{ or } 1.$$

For any choice of the w_i the problem reduces to a transportation problem, so that the dual decomposition method seems to be the obvious choice in this case.

UNCONSTRAINED OPTIMIZATION

The problem Max $f(x)$, $x \in R_n$ is not a simple one. We will assume the existence of a continuous gradient vector $\nabla f(x)$, also denoted by $g(x)$. Most methods are methods of feasible directions, hence they determine at $\bar{x}$ a direction vector $\bar{s}$, satisfying $\nabla f(\bar{x})^T \bar{s} > 0$ and then they make such a step in that direction that progress is obtained ($f(x)$ increases). The steplength is usually determined by one-dimensional maximization or line search, i.e. by solving the problem in one variable:

$$\max f(\bar{x}+\lambda\bar{s}) \quad \text{or} \quad \nabla f(\bar{x}+\lambda\bar{s})^T\bar{s} = 0. \tag{22}$$

This line search problem is by no means simple. Besides all kinds of interpolation methods there have been developed search methods like the golden section and Fibonacci search. It is clear that the best we can aim at by proceeding this way is a local maximum. The problem of local maxima is with us in the whole field of nonlinear programming.

The most well-known method for unconstrained optimization is the variable metric method, originally suggested by Davidon and further developed by Fletcher and Powell, and working along the following lines:

(1) $x^0 \in R_n$ arbitrary, H_0 an n by n symmetric and positive definite matrix, $k = 0$;

(2) $s^k = H_k g^k$;

(3) λ_k obtained by solving (22);

(4) $x^{k+1} = x^k + \lambda_k s^k$;

(5) $H_{k+1} = H_k - P_k - Q_k$ with

$$P_k = \lambda_k \frac{s^k (s^k)^T}{(\Delta g^k)^T s^k}, \quad Q_k = \frac{H_k \Delta g^k (\Delta g^k)^T H_k}{(\Delta g^k)^T H_k \Delta g^k}, \quad \Delta g^k = g^{k+1} - g^k, \tag{23}$$

$$g^k = \nabla f(x^k);$$

(6) optimality test; if not optimal step up k and go to (2).

The justification follows from the following observations for a quadratic objective function $f(x) = p^T x - \frac{1}{2}x^T C x$:

(1) $(s^i)^T C s^j = 0$, $i \neq j$, the directions are mutually conjugate, hence the maximum will be found after at most n steps;

(2) $H_k C s^i = s^i$, $i = 0,1,\ldots,k-1$; hence $H_n C = I$, $H_n = C^{-1}$. The same holds for $\bar{P}_k = \sum_{i=0}^{k-1} P_i$, $k \geq 1$.

The variable metric method therefore has the quadratic termination property. Quadratic termination can also be obtained in the following way:

(1) $x^0 \in R_n$ arbitrary, s^0 such that $(g^0)^T s^0 > 0$.

(2) At step $k \leq n$ require for the feasible direction to be determined:
   a. $(\Delta g^k)^T s = 0$, $k = 0,1,\ldots,k-1$;
   b. $(g^k)^T s > 0$;

(3) λ_k to be determined by line search;

(4) $x^{k+1} = x^k + \lambda_k s^k$;

(5) optimality test; if not optimal go to (2).

Since for a quadratic function $\Delta g^k = -\lambda_k C s^k$ the requirement $(\Delta g^k)^T s = 0$ is equivalent to $(s^k)^T C s = 0$, so that we will obtain conjugate directions and therefore quadratic termination. In the case of a non-quadratic objective function we cannot continue adding relations of type (2)a to the direction finding problem. There are several possibilities like start afresh after each n steps with x^0 (new) = x^n (old) or (probably to be preferred) work with a moving tableau of n relations; at each step a new one is added and the oldest one is omitted.

The relations (2)a and b are of exactly the same type as those in (13) and (14). Therefore the same direction generators can be used as discussed earlier (formulae (17) and (18)). This makes it easy to develop a so-called method of conjugate feasible directions for linearly constrained nonlinear programming problems.

An interesting method in this class which competes favourably with the variable metric method is:

(1) x^0 arbitrary; H = I (or any reasonable estimate of the matrix of second partial derivatives);

(2) for k = 0,1,...,n-1:

a. s^k is the solution of the problem

$$\max\{(g^k)^T s \mid (\Delta g^i)^T s = 0,\ i = 0,1,\ldots,k-1;\ s^T H s \leq 1\}; \tag{24}$$

b. λ_k by solving (22);

c. $x^{k+1} = x^k + \lambda_k s^k$.

(3) optimality test; if not optimal go to (4);

(4) $H^{-1} = \sum_{k=0}^{n-1} \frac{\lambda_k s^k (s^k)^T}{(g^k)^T s^k}$, x^0 (new) = x^n (old), go to (2).

The matrix H^{-1} will be an approximation to the inverse of the matrix of second partial derivatives. Therefore the first step in each sequence of n steps with $s^0 = H^{-1} g^0$ will be a quasi-Newton step. Successive solution of (24) is equivalent to n steps of the type $s^k = H_k g^k$ like in the variable metric method with

$$H_0 = H^{-1}, \quad H_{k+1} = H_k - \frac{H_k \Delta g^k (\Delta g^k)^T H_k}{(\Delta g^k)^T H_k \Delta g^k}.$$

Another unconstrained optimization method which is a direct generalization of the method of conjugate gradients for solving a quadratic maximization problem (or equivalently a system of linear relations Cx = p with C symmetric and positive definite) has been suggested by Fletcher and Reeves. It has the advantage that no n by n matrix has to be stored. Steplengths are being determined in the usual way. The directions are determined by using the updating formula:

$$s^0 = \mu_0 g^0, \quad s^k = s^{k-1} + \mu_k g^k, \quad k \geq 1; \quad \mu_k = \frac{1}{(g^k)^T g^k}, \quad k \geq 0.$$

Polak has found that better convergence can be obtained by taking

$$\mu_k = \frac{1}{(g^k - g^{k-1})^T g^k} .$$

Since for a quadratic function the various gradients in the conjugate gradient method are mutually orthogonal this is indeed another method having the quadratic termination property.

Recently some results have been obtained in developing unconstrained optimization methods that do not require linear searches. Although some promising results have been obtained it is still too early to draw any conclusions. The reader is referred to a survey article by Powell in the first issue of the journal Mathematical Programming.

LINEARLY CONSTRAINED NONLINEAR PROGRAMMING

The problem will now be:

$$\text{(25)} \qquad \text{Max}\{f(x) \mid Ax \leqq b,\ x \geqq 0\}.$$

The optimality conditions are now:

$$\text{(26)} \qquad \bar{x} \text{ maximum} \longrightarrow \exists \bar{u},\bar{v} \geqq 0,\ \nabla f(\bar{x}) = A^T\bar{u} - \bar{v},\ \bar{u}^T\bar{y} = 0,\ \bar{v}^T\bar{x} = 0.$$

The conditions are not sufficient unless $f(x)$ is pseudoconcave. An interesting simple special case is obtained when $f(x)$ is quadratic, hence $f(x) = p^Tx - \frac{1}{2}x^TCx$. In that case the optimality conditions are linear,

$$\text{(27)} \qquad p = Cx + A^Tu - v,\ u^Ty = 0,\ v^Tx = 0,\ u \geqq 0,\ v \geqq 0,$$

and it can be tried to find a feasible solution of the relations $Ax + y = b$, $x \geqq 0$, $y \geqq 0$ and (27). This has first been done by Wolfe. The method we will describe has been suggested by Dantzig. Both methods assume C to be positive semi-definite. Dantzig considers the tableau

		u	x
y	b		A
v	-p	$-A^T$	-C

where the orthogonality relations are satisfied (y=b, u=0, hence $u^T y=0$), but there is no feasibility. The latter is obtained by complementary pivoting. After a number of pivotal operations variables z_i will be in the basis and w_i will be non-basic. Orthogonality is still maintained; z_i is complementary to w_i. We then proceed as follows:

(1) choose $z_i < 0$ (most negative one); set k = i;

(2) increase w_k until one of the basic variables which was $\geqq 0$ will become < 0 at a further increase of w_k; suppose this is z_r;

(3) if r = i, interchange z_i and w_k and go to 1;

(4) if $r \neq i$, interchange z_r and w_k, set k = r and go to (2).

It is not difficult to prove that after a finite number of pivotal operations we will be back at rule (1). The number of $z_i < 0$ has then been decreased by at least 1, so that the method is finite. Since orthogonality is always restored when we go to (1) the method will indeed give the solution of the quadratic programming problem.

Most other methods for the quadratic and linearly constrained non-linear programming problem are based on the principle of (conjugate) feasible directions. The steplength is determined by either (15) or (22) depending on which formula gives the smaller λ. If it is (15) conjugacy relations, if any, have to be omitted from the direction finding problem and the set of relations $a_i^T s \leqq 0$ and $s_j \geqq 0$ has to be adapted to the new point $\bar{x}$. If it is (22) a conjugacy relation $(\Delta g^k)^T s = 0$ will be added to the direction problem. To avoid zigzagging between hyperplanes we will further require $a_i^T s = 0$ or $s_j = 0$ when we hit the corresponding hyperplane for the second time. In this way we have obtained a whole class of finite quadratic programming methods. Which one we will get depends on the direction generator chosen. For more general objective functions we must still decide what to do when no usable direction can be found. Inspecting the set of linear relations we will choose the oldest relation that can either be relaxed (if equality can be replaced by inequality) or omitted (if it is a conjugacy relation). We continue relaxing and/or dropping until either a usable direction is found or no relations can be

relaxed or dropped any more, while there is still no usable direction. In the latter case we have arrived at a maximum.

It is also possible to adapt the variable metric method to cope with linear constraints. First we project the gradient onto the cone $S(\bar{x})$. This will give a projection matrix $P_{\bar{A}}$. We take $H_0 = P_{\bar{A}}$ and start applying the variable metric method which can be continued as long as λ is determined by (22). However, as soon as a new hyperplane is hit we must update $P_{\bar{A}}$ and restart the variable metric method in the last point obtained with $H_0 = P_{\bar{A}}$ (new).

GENERAL NONLINEAR PROGRAMMING

The general nonlinear programming problem will be defined by

$$\text{(28)} \qquad \text{Max}\{f(x) \mid f_i(x) \leqq b_i,\ i=1,\ldots,m;\ Ax \leqq q,\ x \geqq 0\}.$$

The optimality conditions (usually called Kuhn-Tucker conditions) now read

$$\text{(29)} \qquad \begin{aligned} \bar{x} \text{ maximum} \iff \exists \bar{u},\bar{v},\bar{w} \geqq 0,\ \nabla f(\bar{x}) &= \sum_{i=1}^{m} \bar{u}_i \nabla f_i(\bar{x}) + A^T\bar{w} - \bar{v}, \\ \textstyle\sum \bar{u}_i(b_i - f_i(\bar{x})) = 0,\ \bar{w}^T(q-A\bar{x}) &= 0,\ \bar{v}^T\bar{x} = 0. \end{aligned}$$

They are necessary under mild restrictions for the nonlinear constraint set $F = \{x \mid f_i(x) \leqq b_i\}$ which are for instance satisfied when the interior F_0 of F is non-empty, i.e. when there is an x such that $\forall i\ f_i(x) < b_i$. The constraint qualification is satisfied in most practical problems. The conditions (29) are sufficient when f(x) is pseudoconcave and the $f_i(x)$ quasiconvex. This includes convex programming (-f and f_i convex).

Quite some work has been done on the theory of nonlinear programming; duality theory has been developed, Lagrangian functions $\phi(x,u) = f(x) - \sum u_i(b_i - f_i(x))$, $u_i \geqq 0$ have been studied, for instance under which conditions it holds that

$$\max_{x\in L}\ \min_{u\geqq 0} \phi(x,u) = \min_{u\geqq 0}\ \max_{x\in L} \phi(x,u),$$

($L = \{x \mid Ax \leqq q,\ x \geqq 0\}$), so that ϕ has a saddlepoint, say in $(\bar{x},\bar{u})$. This

is for instance the case when $\bar{x}$, $\bar{u}$ satisfy the Kuhn-Tucker conditions and when -f and f_i are convex. In that case we can define

- the primal problem $\max\{\phi_1(x) \mid x \in L\}$ with $\phi_1(x) = \min\{\phi(x,u) \mid u \geqq 0\}$;
- the dual problem $\min\{\phi_2(x) \mid u \geqq 0\}$ with $\phi_2(u) = \max\{\phi(x,u) \mid x \in L\}$.

These theoretical developments have had practical value. Although attempts to find methods of solution based on the dual problem have not met with much success (the convexity assumptions moreover being a severe limitation for practical applicability) it has been shown, for instance that the dual of the so-called geometric programming problem (a highly nonlinear problem)

$$\text{Min}\{h_0(z) \mid h_k(z) \leqq 1,\ k=1,\ldots,p;\ z > 0\}$$

with

$$h(z) = \sum_{i=1}^{m} c_i \{ \prod_{j=1}^{n} z_j^{a_{ij}} \},\ c_i > 0,\ z_j > 0$$

is a linearly constrained nonlinear programming problem. This is an important result since the latter problem having linear constraints is more easy to solve than the original one.

Another nice result is that Kelley's cutting plane method for convex programming (developed independently by Cheney and Goldstein) and Wolfe's decomposition method for convex programming are dual to each other. Since both methods make an essential use of convexity they have little practical value.

For the solution of (28) use is often made of penalty function methods. An auxiliary function is defined by

$$g(x,\rho) = f(x) - h(x,\rho) \tag{30}$$

where ρ is a parameter > 0 and the penalty function $h(x,\rho)$ has to satisfy the requirements:

(1) for $\rho > 0$, $h(x,\rho)$ is differentiable in F_0,

(2) $\forall x \in F_0$: $h(x,\rho) \to 0$ when $\rho \downarrow 0$,

(3) if $x^k \in F_0 \to \bar{x} \in F\text{-}F_0$, then for any $\rho > 0$, $h(x^k,\rho) \to \infty$.

We could say that a ditch has been dug around the boundary, so that once in F_0 we will never leave it when we maximize $g(x,\rho)$.

We now solve a sequence of linearly constrained nonlinear programming problems

$$\max\{g(x,\rho_k) \mid x \in L\} \tag{31}$$

for a decreasing sequence of ρ_k tending to 0. We always take the final solution $\hat{x}^k$ of a subproblem as initial solution of the next subproblem $(x^0)^{k+1} = \hat{x}^k$. It can then be proved that under mild conditions any point of accumulation of the sequence x^k is a local maximum of (25). Fiacco and McCormick who have done a lot of experimentation with penalty function methods have chosen

$$h(x,\rho) = \rho \sum_{i=1}^{m} \frac{1}{b_i - f_i(x)} .$$

In the case of nonlinear equalities also being present, $f_i(x) = b_i$ for $i \in I$ they add another penalty term

$$-\frac{1}{\rho} \sum_{i \in I} z_i^2 \text{ with } z_i = b_i - f_i(x).$$

It is clear that this term has the effect to drive the z_i to 0 when $\rho \downarrow 0$ and $g(x,\rho)$ is maximized.

There have been proposals for many other penalty function methods. Computational experience has been relatively good for problems with up to 60-80 variables.

For larger nearly linear NLP problems a direct method not using a penalty function is to be preferred. Any method of feasible directions has two additional problems. First the steplength procedure is no longer simple since equations of the type $f_i(\bar{x}+\lambda\bar{s}) = b_i$ have to be solved. Secondly the cone of feasible directions is no longer closed since a direction vector satisfying $\nabla f_i(\bar{x})^T s = 0$ (in $\bar{x}$ such that $f_i(\bar{x}) = b_i$) will be in the tangent plane in $\bar{x}$, so that in many cases, for instance for f_i strictly convex, s is not feasible. We must then require $\nabla f_i(\bar{x})^T s < 0$, e.g. ≤ -1. In this way a method of feasible directions can be constructed. Another possibility

is that we make a small move in the tangent plane after which we try to get back into F. This "hemstitching" has been implemented in the so-called generalized reduced gradient method (GRG method). Still another possibility is that we develop a sequence of interior points $x^k \in F_0$, so that $f(x^k) \to \max\{f(x) \mid x \in F \cap L\}$.

At step k we solve a linear subproblem consisting of the linear constraints, linearizations from previous steps and, possibly, conjugacy relations. This gives a solution w^k. The line connecting x^k and w^k intersects $F \cap L$ in z^k. Let $s^k = z^k - x^k$. If from (22) it follows that $\lambda_k < 1$, then $x^{k+1} = x^k + \lambda_k s^k$ and a conjugacy relation is added to the linear subproblem. If, however, $\lambda_k \geq 1$ (as determined by (22)), then $x^{k+1} = x^k + \lambda s^k$ with $0 < \lambda < 1$, say $\lambda = \frac{1}{2}$; the linearizations in z^k, $\nabla f_i(z^k)^T(x-z^k) \leq 0$ for i such that $f_i(z^k) = b_i$ are added to the linear subproblem. Conjugacy relations, if any, are omitted. The process is then repeated. This modified feasible directions method can easily be adapted to nonconvex problems.

For separable programs

$$\text{Max}\{\sum_{j=1}^{n} f_{0j}(x_j) \mid \sum_{j=1}^{n} f_{ij}(x_j) \leq b_i,\ x_j \geq 0\}$$

complete linearization beforehand has been sucessfully applied. This separable programming method has been developed by Miller.

FINAL REMARK

From this incomplete survey it is already clear that there are a great number of methods, algorithms and computational variants that are worth trying out on large practical problems. However, programming and computer experimentation are by no means inexpensive any more. It remains an open question how this financing can be organized. According to the author it is certainly worth trying to organize these experiments. Many of the more important problems man has to solve will ultimately be formulated as large mathematical programming problem: models for population control, for ecological systems, for balanced economic growth in developing countries, etc. Without adequate methods of solution and computer programs there is little reason to try to build the large models required. The next 25 years will be an even larger challenge to the mathematical programming practitioners than the first 25 have been.

LITERATURE

Books on Linear Programming:

S. Gass, Linear Programming (3rd edition), McGraw-Hill, 1969.

G.B. Dantzig, Linear Programming and Extensions, Princeton University Press, 1963.

Books on Nonlinear Programming:

O.L. Mangasarian, Nonlinear Programming, McGraw-Hill, 1969 (theory).

J. Kowalik and M.R. Osborne, Methods for Unconstrained Optimization Problems, Am. Elsevier, 1968.

A.V. Fiacco and G.P. McCormick, Nonlinear Programming, Wiley, 1968 (esp. penalty functions).

W.I. Zangwill, Nonlinear Programming, Prentice Hall, 1969.

E. Polak, Computational Methods in Optimization, Academic Press, 1971, (directed towards control theory).

Books on Integer Programming:

T.C. Hu, Integer Programming and Network Flows, Addison-Wesley, 1969.

Conference Proceedings:

R.L. Graves and P. Wolfe (eds.), Recent Advances in Mathematical Programming, McGraw-Hill, 1963 (Chicago Math. Progr. Symposium, 1962).

J. Abadie (ed.), Nonlinear Programming, North-Holland, 1967 (Nato Summer School, Menton, 1964).

H.W. Kuhn (ed.), Proceedings of the (1967) Princeton Symposium on Mathematical Programming, Princeton Univ. Press, 1970.

R. Fletcher (ed.), Optimization, Academic Press, 1969 (Keele Conference, 1968).

E.M.L. Beale (ed.), Applications of Mathematical Programming Techniques, The English Universities Press, 1970 (Nato Conference at Cambridge, 1968).

J. Abadie (ed.), Integer and Nonlinear Programming, North-Holland, 1970 (Nato Summer School, Bandol, 1969).

J.B. Rosen, O.L. Mangasarian and K. Ritter (eds.) Nonlinear Programming, Academic Press, 1970 (Madison Conference, 1970).

Journal:

Mathematical Programming (first issue October 1971).