STAATSBEDRIJF DER POSTERIJEN, TELEGRAFIE EN TELEFONIE

REPORT 164 MA

PROCESS FOR AN ALGOL TRANSLATOR
PART ZERO:
INTRODUCTION

PART THREE:
THE TABLES

DR. NEHER LABORATORIUM

REPORT 164 MA

PROCESS FOR AN ALGOL TRANSLATOR
PART ZERO:
INTRODUCTION

PART THREE:
THE TABLES

BY: G. VAN DER MEY
WITH THE CO-OPERATION OF: W.L. VAN DER POEL
P.A. WITMANS
G.G.M. MULDERS

JULY 1962

Preface

The present work is a description of an ALGOL 60 compiler for ZEBRA. However, the scope of the work has been made much wider than a strict description of the action of the compiler for the particular machine code of ZEBRA. We have tried to give the description in ALGOL language itself and but for insignificant details the description is machine free. This means that the system can be coded in any machine language. This was only possible because this ALGOL compiler translates source language into an intermediate interpretive code. This interpretive code is of course again machine free. So is the interpreter.

The compiler is of the load-and-go type. First the translator is put into the store; then the source program can be translated and is directly put into the store. Secondly the interpreter is put in on top of the translator and the program can start working.

A few details have been omitted from the report on purpose. The action of the arithmetics of real and integer is too well known to need description; in some machines it is a built-in function, in some it is not. The conversion from hardware input language to identifiers, numbers and delimiters has also been delegated to a procedure called "input" of which only a flow diagram has been added for the ALCOR hardware conventions. The same is true for "input1",for strings. No descriptions of machine code body procedures have been added, as these differ too much from machine to machine. However, it will be clear from the report how the same procedure heading can be used for machine code body procedures.

As many features of ALGOL 60 have been incorporated as possible with the exception of own dynamic arrays and subscripted controlled variables in for statements.

Even a lot of extra features are incorporated such as intermediate assignment, use of blocks as switch elements, implicit assignment to procedure identifiers in type procedures.

The present work is distributed in the spirit of the ALGOL 60 report and is not copyrighted. In the machine free way it is described it is thought to belong to the realm of pure mathematics. However when used, please state the source explicitly.

W.L. v.d. Poel

0. Process for an ALGOL Translator.

The process described below has been developed for the ZEBRA, which binary machine contains 8192 locations in its store. Each location consists of 33 bits

$b_0, b_1, \ldots . b_{32}$.

b_0 is the sign digit: $b_0 = 1$ means that word is negative.

Within the store, the locations with addresses P0, P0 + 1, P0 + 2, ..., Q0 are the working space of the translator, which itself may occupy the addresses

Q0 + 1, Q0 + 2, ...

In translation time, an ALGOL text being read from the tape, is translated into a binary form which will be called below an object programme, and the successive words of the object programme are stored on the addresses

P0, P0 + 1, P0 + 2, ...

Each word of the object programme is either a programme constant or an instruction. An instruction is either a ZEBRA jump instruction which is always positive, and must be normally executed, or it is negative, being written in a convenient intermediate code to be interpreted. In the operation time of the object programme, the interpreter is supposed to take the place which, in translation time, is occupied by the translator.

0.1 Representations.

Integer representation:

Integral values v, $-2^{32} \leq v < 2^{32}$, can all be represented by words

$b = \overline{b_0 b_1 \ldots b_{32}}$

in which b_0 is the sign digit.

The logical values will be represented by integers:

true = 0, false = -1

The or operation performs the logical product and may also be applied to integers.

real representation:

When i is any fixed integer between 1 and 31, a word b defines, through the relations

m = fraction $\overline{b_0 \cdot b_1 \ldots b_{31-i}}$ and e = integer $\overline{b_{32-i} \ldots b_{32}}$, a value

$v' = m \times 10^e$

However, v' = 0 will be represented by m = e = 0. For the mantissa and exponents of values $v' \neq 0$ the following bounds are observed:

$0{:}1 \leq x < 1, \quad -2^i \leq e < 2^i$

Thus b_{32-i} is the sign digit of e

"Accumulator".

For retaining the value accu obtained last in calculations, the interpreter uses an accumulator consisting of the two variables mant and exp of table 4B. Either mant is the integer representation of accu and exp is = 0, or when real representation is required, mant is the mantissa of accu and exponent = 1 + 2 x the exponent of accu. Thus exp ≠ 0 indicates that accu is given in real representation. Then exp_0 is the sign digit of the exponent. Before storing, the mantissa of accu must be rounded to 31 - i digits behind the binary point and is joined by the exponent which, of course, must be within the bounds mentioned above.

0.2 Working space.

In an object programme, any instruction requiring a constant is immediately followed by that constant. Thus the translator does not assemble a list of programme constants.

Conveniently supposing that arrays with variable bounds are not own, the translator reserves, within the object programme, one or more fixed spaces for the own arrays and own variables declared in the text being translated.

The completed object programme may occupy the addresses P0 to P1 - 1. Then the space

P1, P1 + 1, ... Q0

is still available for the simple variables and arrays which are not own. For an efficient use of relative addresses it is advantageous to isolate, within the working space, the simple variables from the arrays. Thus, at any moment of the operation time, the space P1 ... Q0 is divided into 3 ranges, P1 ... P - 1, P ... Q, and Q + 1 ... Q0, so that the first range and the third range are fully occupied by the arrays respectively the simple variables which are still, or again, in use, while the middle range is not occupied. Of course, the lower pointer P and the upper pointer Q are no constants but variables of the interpreter (cf. table 4B).

0.3 Relative addresses.

After reading a simple variable i to be declared which is not own, the translator associates to i a fixed relative address y which is the value of an address pointer q and the variable q is decreased by 1. q has the initial value Q0 (cf. table 4A).

For enabling an object programme to handle arrays with variable bounds or to retain values of the pointers etc., the translator must provide it with some internal simple variables which are not declared in the text. When introducing an internal variable i, the translator again associates a relative address y = q to i. Thus each of the relative addresses q + 1, q + 2, ... Q0 is occupied by either a declared simple variable being not own, or an internal variable. Internal variables are introduced only when translating a block head or procedure heading. They may be regarded as being local to the block or procedure concerned.

After translating a block B, the translator assigns to variable q the value which is the highest relative address occupied by a simple variable which is local to B. That value of q was resident when B was going to be translated. The variable with the relative address q + 1 is not local to the translated block B.

Before translating a procedure P, the translator assigns to the variable q its initial value Q0. If formal parameters are present, the key (cf. table 3) of the first formal parameter is given the relative addresses Q0 and Q0 - 1, etc. After translating P, the translator again assigns to q the value which was resident immediately before the translation of P was beginning.

0.4 Absolute addresses.

When, in operation time, the object programme of a block or procedure B is going to operate, one location L, with the address p + y, is reserved for each local simple variable i of B, y being the relative address of i with respect to B. Location L is at the disposal of variable i until the operation of B finishes. The pre-value p is a variable of the interpreter which is = 0 when no procedure is operating. Thus, in the case of a block B which is not contained in any procedure text, the relative address y of i is in fact the absolute address of the location occupied by i.

When any procedure B is called, the value Q - 5 - Q0 is assigned to variable p, and this value is restored whenever, after an interruption, the operation of procedure B continues. At the call of B the space P...Q is not yet occupied. As the relative address y of a local variable i is ≦ Q0, the absolute address p + y is ≦ Q - 5 thus not yet occupied when i comes into process. The previous value of p is stored on the address

Q - 3 = p + Q0 + 2

thus being a "local simple variable" of procedure B with the relative address Q0 + 2. Of course, the value of pointer Q must be adjusted.

0.5 Intermediate code.

When the interpreter (cf. label S10L2) extracts an instruction

$$I = \overline{I_0 I_1 I_2 \ldots I_{32}}$$

from the object programme, the next word N of the object programme is also extracted. If N is no programme constant and, in addition, I does not make the interpreter perform a jump in the object programme, the interpreter augments its extraction instruction e by 1.

If the bit $I_0 = 0$, then I is a machine code instruction and is executed normally.

If the bit $I_0 = 1$,
instruction I must be interpreted according to the intermediate code to be described now. The instruction I consists of the following groups of bits:

1 $I_0 \ldots I_6$ - the operation part.

In the intermediate code, every instruction I (adding, storing, jumping, etc.) has a number q, $64 \leq q < 128$. $2^{26} \times q$ is the operation part of I.

When, in table 1A, q is < 96, I is called a <u>calculative</u> instruction and requires, besides a value to be extracted, the value accu contained in the accumulator of the interpreter. In table 1A only the "progressive" version of the calculative instructions is listed, from which the "regressive" version is obtained by inverting the bit I_6. Thus the regr. version of $<$ is equal to the progr. version of $>$.

When $96 \leq q < 104$, (cf. table 1C) I is called an <u>extractive</u> instruction and requires only the value to be extracted.

When $104 \leq q < 128$ (cf. table 1D), I is called a <u>non-extractive</u> instruction.

2 I_7 - the type bit.

$I_7 = 1 \rightarrow$ the value to be extracted or stored by instr. I has the <u>real</u> representation.

$I_7 = 0 \rightarrow$ - has the <u>integer</u> representation or is <u>boolean</u>.

3 I_8

If the address part (cf. 6) of I is 0, there are 2 possibilities (cf. S10L6):

$I_8 = 0$ The word N next to I and already extracted, is a programme constant required by I.

The extraction instruction e is augmented by 2 instead of 1. I is a calculative or extractive instruction.

$I_8 = 1$ Instruction I requires a partial result located on addresses Q + 1 and Q + 2, and augments pointer Q by 2. I is calculative.

4 I_9 - formal bit.

$I_9 = 1$

Instruction I does not refer to a parameter key.

$I_9 = 0$. Then is also: $I_7 = I_8 = 0$.

Instruction I refers to the key (table 3) of a formal parameter. The test on S10L13 succeeds and the parameter key is extracted. If there has been specified a type t for the parameter in the text, there happens on S10L31:

I_7 : = type bit t, and eventually bit I_8 : = 1. If the formal parameter represents a variable and instr. I is calculative or extractive, the bit $I_8 = 1$ indicates, on S10L18, that the representation of the value just extracted from that variable, must be changed before the value is used. If the parameter is a function or expression, the bit I_8 is tested on S10L51.

5 $I_{10} \ldots I_{19}$ - rank part.

In translation time, ranks are introduced:

Own declared identifiers have the rank 0.

A label, an identifier, or an internal variable, defined or introduced at a moment when no procedure of the text is translated, has the rank 0.

When a procedure identifier has the rank r, the procedure concerned has the rank r + 1

A label, an identifier being not own, or an internal variable, which is defined or introduced when the translation of a procedure is running, has the rank r of that procedure.

$R = 2^{13} \times r$

is a variable of the translator, in which r is either the rank of the procedure which is being translated, or 0, when no procedure is being translated.

In the rank part $2^{13} \times r$ of an instruction I, r is the rank of the object (simple variable, label, etc.) to which I refers.

6 $I_{20} \ldots I_{32}$ - address part.

For the address part 0 confer 3 above.

When the address part is an address $y > 0$ and if I is an instruction for performing a jump, then the word $\{y\}$ is the instruction to be executed next.

When I is no jumping instruction and if the rank part of I is = 0, then $\{y\}$ is the value required.

Otherwise the rank part of I enables the interpreter to look up the pre-value p1 to be added to the relative address y contained in I (cf. S10L12). Then $\{p1 + y\}$ is the required value.

0.6 Basic strings.

In ALGOL there occur 3 kinds of basic strings: identifiers, constants, and delimiters.

There may, and will, be supposed that a constant is an unsigned number. The sign which may precede it in the text, is a delimiter.

The basic strings true and false will be regarded as to be constants.

Within the text, the beginning and the end of each identifier or constant is marked by a neighbouring delimiter.

Disregarding comment and the string quotes, the delimiters will now be divided up into 5 groups:

1 the arithmetic, relational and logical operators (table 1A).
2 the declarators and specificators (table 2).
3 the colon (table 1B).
4 the opening symbols.

Within a statement, the first delimiter differing from colon is an opening symbol i.e. one of the delimiters begin for go to if : = ([

(In the first column of table 1B, procedure and switch are also listed as "opening symbols", but these delimiters are regarded here as declarators).

5 the separation- and closing symbols:

comma semi-colon then else step until while do end)]

The input part of the translator reads each time the translator goes to it, either a comment string (which will not be considered here) or one basic string s.

If s is an opening symbol or colon, the input part returns to the entry mentioned for s in table 1B.

Otherwise the input part represents s by a single word f (f is variable of the translator, cf. table 4A) and returns to either S1L1, for operators, separation- and closing symbols,

or S3aL1, for declarators and specificators,

or S4L1, for identifiers,

or S4aL1, for constants.

The delimiter := consists of 2 delimiters. There may, and will, be supposed that the input part is able to be aware of the whole of such a compound delimiter thus does not return to the translator after reading only the first symbol contained in it.

Two delimiters such as = - which immediately follow one the other in the text without forming a compound delimiter, are said to be "separated by the 0-identifier". In this case, the input part performs two returns to the translator.

0.7 Identifiers.

Identifiers consists of the decimal digits, and only one kind of alphabet, which can be numbered 0 to 9, 10 to 35 respectively.

For retaining 6 characters (letters and digits) of an identifier, 32 bits i_1 to i_{32} of a word i are required, while i_0 is supposed to be = 0. This is done in such a way that the characters 0-9, 10-35 are converted as digits a radix 37 system into binary. All overflow above 31 bits is removed and i_1 and i_2 are made = 1 as soon as more than 6 characters are read. Thus an identifier of 6 characters or less can never be the same as another of 6 or less. But two identifiers of more than 6 characters can be the same although the chance is very small.

An integer n, $0 \leq n < 2^{28}$ that occurs as a label, will, in this quality, be represented by

$$i = n + 2^{30} \times 3$$

which, differs from the representation of any identifier.

To the representation i of a label or identifier corresponds the (negative) <u>contra-identifier</u>

$$i' = i + 2^{32}$$

0.8 The declaration pattern.

The declaration pattern

$$D = \overline{D_0 D_1 \ldots D_{32}}$$

(cf. table 2) is a variable of the translator which is positive only when identifiers are being declared or specified.

After reading begin or a semi-colon occuring within a blook or compound statement, there happens on S3L1 resp. S3bL4:

$$D := \overline{1\ 111111111\ 0\ldots0}$$

When the next delimiter should be a declarator, the translator, on S3aL1, replaces D by its logical product with the representation f of the declarator, which is positive. The positive value of D indicates to scheme 4 that, and how, an identifier just read must be declared.

In a procedure declaration, the opening parenthesis of the formal parameter part makes the translator proceed to S5L2, where there happens:

$$D := \overline{0\ 000011110\ 0\ldots0.}$$

This value indicates to scheme 4 that the identifier just read is a formal parameter to be defined (declared). In this case, the parenthesis is in fact a kind of declarator.

Semi-colons occuring in a procedure heading make the translator proceed to S8dL9, where there happens:

$$D := \overline{1\ 000011111\ 0\ldots0}$$

This value is still negative. However, when the next delimiter should be any declarator or specificator, the value of D is modified such as to make the translator, after reading an identifier i , proceed to S4L6 for specifying parameter i in the object programme.

0.9 The list I of declarations.

To any identifier i having just been declared in the text, the translator associates an internal equivalent e, stores f = i and e on the addresses T and T - 1 as indicated by its variable T, and decreases T by 2 (cf. procedure S0e as invoked on S4L20). Variable T takes only values Q0, Q0 - 2, Q0 - 4, etc. Thus a list I of declarations is formed.

Mostly, e has the form (cf. table 2):

$$e = y + 2^{13} \times 5 + D$$

in which D is the declaration pattern as left by the combination of declarators preceding the identifier list in which i is contained. y and r are the address and the rank given to i.

If i is an own variable (cf. the failing test on S4L9), then a location with the address y is reserved for i within the space of the object programme, and r = 0. If i is a simple variable not being own, then y is the relative address as indicated by the variable q, and $2^{13} \times r$ is the current value of variable R, r being either the rank of the procedure which is being translated, or = 0 (when no procedure is being translated).

If a variable i is referred to within a statement or expression, the translator proceeds to S4L22 and looks up i and e in the list I. y, r, and the bits e_7, e_8, e_9 are included in the instruction to be formed.

After translating a block or procedure B, on S3bL14 the value T' = e = - 2 + (lowest address where an identifier being not local to B is located in the list I) is assigned to variable T. Thus the identifiers which are local to the translated block or procedure (in the case of a procedure, only its formal parameters and eventual labels are meant), are no longer retained in the list I. In most cases, T' is the value variable T had when B was going to be translated.

When an identifier i is referred to in the text, the translator (cf. S4L22 and the more complicated case of S3bL13 where at first a contra-identifier is extracted) looks for the lowest address where i occurs in the list I, by comparing i successively with the identifiers

{T + 2}, {T + 4}, {T + 6}, ...

The internal equivalent is extracted. Thus the identifiers which

are not local to the - youngest block or procedure B whose translation is not yet completed, are considered only when i is not found among the local identifiers of B. Thus the translator is not confused, when an identifier which is local to B, has outside the text of B another significance. As a local significance of i is not yet, or no longer, listed in I before, or after, translating B, no reference may be made to that significance of i outside the text of B.

The declarations of the h standard functions may be included in the translator programme on the addresses
Q0 + 1, Q0 + 2, ... Q0 + 2xh - 1, Q0 + 2xh.
They are permanent and immediately join the list I so that they are available at any stage of a translation. They refer to fixed programmes of standard functions which are sections of the interpreter programme.

0.10 Opening-, separation- and closing symbols.

The definition of opening symbol as given on page 0.6.1. par. 4 is sufficient for input purposes. For unterstanding the translator a better formulation is required:

Opening symbols are:

1 the delimiters begin for go to if [

2 the parenthesis (unless preceding a list of formal parameters to be defined

3 the declarators procedure and switch unless used in the quality of specifier

4 := unless contained in a for statement or switch declaration.

In the context, an opening symbol is followed by a piece of text which is called its court and is largely characterized by it.

As an array declaration is already characterized by the opening bracket together with the declaration pattern, the declarator array will not be treated below as opening symbol.

The delimiter s' marking the end of the court of an opening symbol s, is the closing symbol of the court of s. Unless s and s' are a pair of bracket-like delimiters, s' does not at all characterize the court of s.

For dividing up courts of opening symbols into pieces, the following separation symbols are provided:

opening symbol:		separation symbol:				
begin		,	;			
for	:=	,		do	step	until	while
go to						
if				then	else	
:=						
(		,				
[		,	:			
procedure		,	;	)		
switch	:=	,				

After begin and procedure, the separation symbol comma occurs only in lists of identifiers to be defined or specified.

As the colon may also occur in the quality of a "declarator" of labels, it has the individual entry S3CL1 for return after reading.

The closing symbol s' of the court of an opening symbol s may combine this quality with that of separation-or closing symbol of the court of another opening symbol.

0.11 The list L.

The variables S0 and S of the translator point to the first and last locations of a list L. When a translation is beginning, there happens in scheme entry:
S0 := L0 S := L0 + 1 {S0} := 0,
L0 being a fixed address somewhere in the mid between P0 and Q0.

When the translator has read an operator or opening symbol f, there happens:
S := S + 2 {S - 2} := f {S - 1} := g := 0
(cf. procedure S0c and label S1L28).

However, in many cases of f being an opening symbol, at first S is increased by 1, 2 or 4 so that in L to the left of f there are 1 + 1 instead of 1 location available for storing information for the opening symbol.

Often, when f is an operator, at first some operators {S - 2}, {S - 4}, ... are translated, before f is listed in L.

After listing a delimiter f in L, input continues and S may be considerably increased. However, when f is on the point of being translated, variable S has again the value 2 + (address where f is listed in L).

Immediately are translating f, variable S is decreased by 2 or 2 + 1 so that f and the information accompanying f are no longer listed in L.

When, in the text, an operator or opening symbol f is followed by a programme constant C, the translator replaces, on S4aL2, the word {S - 1} = 0 following {S - 2} = f by C, while the input part has already replaced the value 0 of g by $\overline{0\ 000000t01\ 0...0}$
in which bit t indicates the representation of C:
t = 1 → C is <u>real</u> t = 0 → C is <u>integer</u> or <u>boolean</u>.

When, in the text, f is followed by an identifier i, there happens, on S4L22 and 23:
{S - 1} : =
if i is a simple variable or formal parameter then the internal equivalent of i else the contra-identifier $\overline{1\ 0...0 + i}$. Unless i is an array identifier which always assigns a negative ar1 instruction as value to variable g, the value 0 of g is not changed.

The additional assignment {S} : = i is useful only when i must be contra-declared.

Thus g > 0 means that the operator or opening symbol {S - 2} = d is followed by a constant, while {S - 1} = g = 0 means that d is followed by the 0-identifier i.e. neither a constant nor an identifier.

Separation- and closing symbols, declarators differing from procedure and switch, specificators, and lists of identifiers to be defined or specified, are not listed in L. Thus, after reading

begin real a: a: = function

the list L contains on the address S - 4 to S: begin, the internal equivalent of a, :=, $\overline{1\ 0...0}$ + function, and function which looks as if a were the identifier immediately following begin in the text.

When translating an actual parameter part, a switch list, or an own array declaration, variable S0 is decreased for temporarily storing a list of constants or instructions in L.

Within the store, the list L resides between the declaration list I in the upper area and the object programme being formed in the lower area. When necessary, shifting L is performed by procedure S0g.

n be the number of locations contained in L on adresses S - 2, S - 4, S - 6, ... Delimiter {S - 2} and the word {S - 1} may be indicated now by d_n and i_n respectively.

0.12 Translation of operators. The rule of precedence.

By applying the traditional rule of precedence to the operators of a text as taken in the order of reading, the order of translating them is obtained.

When assigning ranks from 1 to 9 to the various operators as indicated in table 1A, the rule of precedence implies that an operator d with rank r which, in the text, is preceded by an operator d' with rank r', is earlier translated than d' only when is: $r < r'$. In the table, d is given in the form

$$d = 32 \times r + 2^{26} \times q$$

the latter term being the operation part of the instruction concerned. The instruction resulting from d and the internal equivalent e of a simple variable or formal parameter, has the form (cf. S1L14):

$$2^{26} \times q + e - 2^{26} \times \overline{0\ 111111}$$

Also parentheses and brackets etc. participate in the game:

Each opening symbol is provided with 2 ranks,
the rank $r = 0$ on its left-hand side, and
the rank $r = 10$ on its right-hand side.
Separation- and closing symbols are provided with the rank $r = 10$.

The values listed in table 1B have the form
$32 \times 10 + s$ with $0 \leqq s < 32$.

After reading an operator, separation- or closing symbol, 32 x the rank of it is assigned to variable a on S1L1 by procedure S0a. Opening symbols do not go to the collective entrance S1L1 but have individual entrances which fact is in accordance with their unconditioned precedence in translation.

Definition.

In the text, d be any operator or opening symbol, and f be the next delimiter which is either an operator or opening-, separation- or closing symbol.

If the right-hand rank r_d of d is not greater than the left-hand rank r_f of f, operator or opening symbol d is called <u>progressive</u>.

If r_d is greater than r_f, d is called <u>regressive</u>.

When, for example, f is a separation- or closing symbol, then d is certainly progressive ($r_f = 10 \geqq r_d$).

When f is an opening symbol, then d is regressive ($r_f = 0 < r_d$).

An operator d of the text is never translated immediately after reading it. After reading d, the translator also reads the constant or identifier i next to d (which may also be the 0-identifier) and the delimiter f next to i. When f is being read, d and the appropriate representation of i are already listed in L as

$$\{S - 2\} = d_n \quad \text{and} \quad \{S - 1\} = i_n$$

If f is an opening symbol, then d_n is regressive, f is also listed in L and reading continues (after the so-called pre-action of the opening symbol has been completed).

If f is either an operator or separation- or closing symbol, then, on S1L1, d_n and i_n are extracted from L, and the ranks of d_n and f, each multiplied by 32, are assigned to the variables a and b. On S1L4, the rank r_n of d_n is compared with the rank r_f of f:

If $r_n > r_f$, then operator d_n is regressive, (operator) f is listed too, and reading continues.

If $r_n \leqq r_f$, then operator d_n is progressive and must be translated. When v_j $(j \leqq n)$ is the value which, in operation time, corresponds to the element i_j of the list L, the operation to be translated may be indicated here by

$$\text{accu} := \text{accu } d_n\ v_n$$

which is eventually preceded, in the object programme, by the operation

$$\text{accu} := v_{n-1}$$

for storing a first value in the accumulator of the interpreter. (When i_{n-1} is the 0-identifier preceding d_n = not or minus, no previous assignment to accu takes place).

After translating operator d_n, i_{n-1}, d_n and i_n are no longer retained. Thus, on S1L17, variable S is decreased by 2. However, in the discussion, n is not replaced by n - 1. On S1L18, i_{n-2} and d_{n-1} are extracted from L.

If the (right-hand) rank of d_{n-1} is $\leqq r_f$ and d_{n-1} is an operator $(r_{n-1} < 10)$, that operator must be translated now (compare the minus sign contained in p - q x r + s).
As d_{n-1} has not been translated immediately after reading d_n, it is regressive. There must be translated the operation:

accu := v_{n-2} d_{n-1} accu.

When operator d_{n-1} is commutative, the operation has the same effect as

accu := accu d_{n-1} v_{n-2}.

Then the operation part 2^{26} x q as contained in d_{n-1} = d is used for translation. When d_{n-1} = d is not commutative, the bit d_6 of d is inverted on S1L24. In operation time, on S10L20, the bit I_6 of the instruction to be interpreted is examined by testing J_{32}: If I_6 = 1, then accu and the value v just extracted are interchanged, before the computation is performed.

After translating operator d_{n-1}, S is again decreased by 2, and i_{n-3} and d_{n-2} are extracted from L. If d_{n-2} is an operator and has a rank $\leq r_f$, d_{n-2} is translated as explained for d_{n-1}, etc.

The cycle for translating operators goes on, shortening the list L, until a delimiter d_{m-1} is extracted which is not an operator having a rank $r_{m-1} \leq r_f$. Then there are only 2 possibilities:

either

1 delimiter f is an operator,

or

2 d_{m-1} is an opening symbol, and
f is a separation- or closing symbol.

For, in L there are listed no delimiters but operators and opening symbols, and there has already been found above: $r_f \geq r_n > 0$. Thus the consequence of $r_{m-1} > r_f$ is: $0 < r_f < 10$ and the consequence of $r_{m-1} \leq r_f$ while d_{m-1} is no operator, can only be: d_{m-1} is opening symbol and left-hand rank $r_f = 10$.

In the case 1 there happens the following:

S' := S { S } := f S := S + 2

Thus operator f is listed as the element d_m on the place of the operator translated last, and the index m is retained by the variable S'. Reading begins again and continues, until an operator or opening symbol d_n = { S - 2 } is found to be progressive.

If n = m thus S = S' + 2, operator d_m itself is progressive so that only one constant or identifier i_m and one delimiter f have been read. Then the operation to be translated next is:

accu := accu d_n v_n

If $n > m$ thus $S > S' + 2$, then, in operation time, the value accu as formed by the operator d_m translated last, is required later by the regressive operator d_m still to be translated. Meanwhile a new calculation must be performed. Thus, on S1L7, the translator inserts the machine code instruction partres of table 1E which makes, in operation time, the interpreter go to S10L0 for storing a partial result as follows:

Q := Q - 2 {Q + 1}:= mant {Q + 2}:= exp

In addition, the translator replaces the obsolete element i_{m-1} by

$$i_{m-1} = 2^{23} \times \overline{0\ 111111011}.$$

Thus, when $2^{26} \times q$ is the operation part of the regressive operator d_m, d_m and i_{m-1} will later give rise to the instruction

$$2^{26} \times q + 2^{23} \times 3$$

with the address part 0, which instruction makes, in operation time, the interpreter proceed to S10L7 for extracting the partial result as stored by the corresponding instruction partres. There happens:

N := {Q + 1} E := {Q + 2} Q := Q + 2.

Loss of accuracy by rounding the mantissa has been avoided by occupying 2 locations. Partial results, stored one after the other, are extracted in the opposite order so that, before each extraction, pointer Q has the required value.

When being an operator, d_n is translated now as mentioned above:

accu := v_{n-1}
accu := accu d_n v_n

Example 1.

p and q or r + s x t > u↑2 implies ...

in which p and q are boolean and the other variables are real and/or integer, is read and listed in L, and the text as arranged in L is translated. In each paragraph below, the first line shows the contents of L and is followed by the operations to be translated. Dashes indicate internal equivalents.

$$\varepsilon_i = 2^{23} \times \overline{0\ 111111011}$$

concerns the partial result number i.

p' and q' f = or

accu := p accu := accu and q
  p' or r' + s' x t' f = >
first partres := accu
accu := s accu := accu x t
  ε_1 or r' + s' f = >
accu := r + accu
  ε_1 or r' > u↑2 f = implies
second partres := accu
accu := u accu := accu↑2
  ε_1 or ε_2 > u' f = implies
accu := second partres > accu
  ε_1 or ε_2 f = implies
accu := first partres or accu
  ε_1 implies etc.

Inversion of the "gression" bit takes place only when translating
accu := second partres > accu.

The translator regards the progressive version of the operators < and ≦ as to be the same as the regressive version of > and ≧ (cf. table 1A).

0.13 Pre- and after-actions of opening symbols.

An opening symbol s just read makes the input part go to the entrance of s as indicated in table 1B. The action performed there will be called the pre-action of s. However, the pre-actions of procedure and switch begin on S4L24 and S4L37, when also the procedure- or switch identifier is being declared. The information needed by opening symbol s, and the word f corresponding to s, are listed in L. The pre-action of (depends on (being preceded in the text by the 0-identifier or not.

After the pre-action of any opening symbol s, the translator goes to input. s remains in the list L as long as the translation of its court is not yet completed. Separation symbols are not listed in L. E_1, E_2, ... which are the pieces into which the court of s is divided up by its separation symbols, are read and translated one by one.

There may be, and is, supposed that, after translating each E_i (i = 1, 2, ...), the variable S takes again the value it had immediately before reading and translating E_i to that S - 2 is again the address where opening symbol s is listed in L, and that the separation- or closing symbol which is next to E_i in the text, is still available as the value of variable f. This is exactly the situation as found on page 0.12.3., case 2.

The action performed now by the translator, will be called an after-action of the opening symbol s = {S - 2} and depends also on the separation- or closing symbol f.

If E_i is but a single constant or an identifier referred to, then opening symbol s is called progressive with respect to its after-action with f. The after-action begins immediately after reading E_i and f and the translator proceeds to S1L6 with d = value of s as indicated in table 1B. Then the constant or identifier E_i must still be translated.

Otherwise s is called regressive with respect to its after-action with f and the translator proceeds to S1L21. Then, in most cases, E_i has already been translated.

Usually the after-action itself, of s with f, is called progressive or regressive. Thus, when translating f(x, 100, x + 1). The first and second after-actions of (are progressive, and the last after-action is regressive.

The way an after-action of an opening symbol s = {S - 2} finishes, depends on the other delimiter, f. There are 3 possibilitis:

1 f is one of the separation symbols in the court of s.
There happens:
S' := 8191
and the translator goes to input. As the variable S is not decreased, opening symbol s remains in L. Separation symbol f is obsolete. The value of S' prevents the translator of inserting an instruction partres not required in the object programme (cf. S1L7).

2 s and f form a pair
either () or [] or begin end.
There happens:
S' := S := 2 + address
where delimiter d preceding s in L, is listed in L. and the translator goes to input. Thus s and f are both obsolete. If the right-hand rank r_d of the regressive opening symbol or operator d should not be greater than the left-hand rank r_f of the delimiter f to be read next, d seems to be progressive, as will be shown in example 2. Then it is the test on S1L5 that makes the translator proceed to the treatment of regressive delimiters. (As, in the case of f = end, the input part must be aware of comment, leaving the normal course. It may itself choose the right return so that the value of S' is no more important).

3 f which is the closing symbol of the court of opening symbol s, is superior to s.
There happens:
S := as indicated in case 2
and the translator proceeds to S1L18 for extracting d from L. Thus opening symbol s is obsolete. When the ALGOL text is correct, d cannot be an operator. Thus d is an opening symbol and f is a separation symbol or the closing symbol of the court of d. The after-action of d with f is carried out.

Thus an opening symbol s is retained in the list L untill its court has been translated. s after-acts successively with each separation symbol, and closing symbol of its court.

Example 2.

p - (q + r) + ...

with real variables.

For the notations below confer example 1 on page 0.12.5.

p' - 0-ident. f = (

pre-action of (preceded by 0-identifier.

p' - 0-id (q' + r' f =) with S' = 8191

accu := q accu := accu + r

p' - 0-id (q' f =)

regressive after-action of (with) of the above type 2.

Thus S' := S := address where (has been listed in L.

p' - 0-id f = + S' = S

Operator minus is regressive, but seems to be progressive. Fortunately the test on S1L5 succeeds and there is translated:

accu := p - accu

p' + etc.

Example 3.

p - (q + r) x s + ...

After the after-action of (mentioned in example 2, the translator continues as follows:

p' - 0-id x s' f = + with S = S' + 2

operator x is progressive and there is translated:

accu := accu x s

p' - 0-id f = +

accu := p - accu

p' + etc.

Example 4.

p - (q + r) x s↑2 + ...

After the after-action of (mentioned in example 2, the translator continues as follows:

p' - 0-id x s ↑ 2 f = + with S = S' + 4

thus test in S1L7 succeeds and that in S1L10 fails.

partres := accu accu := s accu := accu↑2

p' - ε x s f = s +

accu := partres x accu

in which x is commutative

p' - ε f = +

accu := p - accu

p' + etc.

When translating

begin real a:

the declarator real is not listed in L. Thus, when semi-colon is read, begin is still listed on address S - 2 and is on the point of after-acting with semi-colon, as happens also in the progressive case. Yet scheme 4 has already declared identifier a so that a need not be considered when begin is after-acting.

Therefore after-actions of begin and procedure with commas and semi-colons occuring in resp. after the lists of identifiers to be defined, will be regarded as being regressive. Therefore, after reading a declarator or specificator, there happens on S3aL4: S' := S . Then the test on S1L5 secures that a regr. after-action is performed, and the corresponding test in S1L2 avoids testing for a 0-identifier.

0.14 If-then-else.

When X is a piece of text, then |X| be the address where the object programme of X begins.

The expression

E ≡ if B then E1 else E2

in which B is a boolean expression, and E1 and E2 are expressions, gives rise to the following object programme:

object programme of B
2^{26} x 111 + 2^{23} + |E2| (cf. test in table 1D)
object programme of E1
2^{26} x 121 + 2^{23} +|etc| (cf. pass in table 1D)
object programme of E2
object programme of etc.

When interpreted,
the pass instruction introduces its address part |etc| into the extraction instruction e of the interpreter (cf. S10L58). The test instruction introduces its address part |E2| into e only, when accu has the value false = - 1 (cf. S10L130).

B, E1, and E2 and their constituents may have the same structure as E above. Then enclosing them within parentheses is not required. The meaning of E is always clear.

Proof. k be the number of triplets if-then-else contained in the text of E. If k = 0, then E has a fixed meaning. Suppose: every expression of the above structure containing less than k triplets, has a fixed meaning being independent of which delimiter with left-hand rank 10 follows it. Then B, E1, and E2 have fixed meanings. When expression B is opened by $\underline{if}_B$, the delimiter then of the outer triplet closes the court of $\underline{if}_B$. The last after-action of $\underline{if}_B$ is of the type 3 of page 0.13.2. Then $\underline{if}_B$ is obsolete and $\underline{if}_E$ starts to after-act with the separation symbol then as intended. Then $\underline{if}_E$ remains in L, f = then is obsolete, and the translation of E1 begins. Etc.

Immediately after its pre-action (scheme S2), opening symbol $\underline{if}_E$ = {S - 2} is preceded in L by {S - 3} = 0.

When $\underline{if}_E$ is after-acting after-acting with f = then and the translator has arrived on S2aL21, expression B has just been translated also in the case of progressive after-action of $\underline{if}_E$.

Then $\{S - 3\} = 0$ is replaced by $-P$, and P is increased by 1. The separation symbol f = then is obsolete.

When $\underline{if}_E$ is after-acting with f = else, thus E1 has been translated, there happens:

$\{-\{S - 3\}\} := 2^{26} \times 111 + 2^{23} + P + 1$,

$\{S - 3\} := +P$, and $P := P + 1$

Thus the test instruction is inserted and a place for the pass instruction is reserved.

f = else is obsolete.

When $\underline{if}_E$ is after-acting with the closing symbol of its court (thus E2 having been translated) there happens:

$\{\{S - 3\}\} := 2^{26} \times 121 + 2^{23} + P$.

Thus the pass instruction is inserted. This time, $\underline{if}_E$ itself is obsolete.

Thus, by testing $\{S - 3\}$, which is either 0 or negative or positive, the translator knows which after-action of if must be performed. The separation- or closing symbol f is useful only through its left-hand rank 10 which has stimulated the translation of the expression preceding delimiter f in the text. Thus, in table 1B, if, then, and else may be represented by the same value 326.

The if statements must still be considered.

Within the text, never a statement

if boolean then statement

in which else is absent, is immediately followed by a delimiter then or else. Thus, when in the second after-action of if delimiter f is found to be different from else, then f is the closing symbol of the above instruction. Then P instead of $P + 1$ is the address part of the test instruction, and P is not increased, and the after-action has the type 3 instead of 1.

0.15 Block, contra-declaration.

A block

B ≡ begin D_1: D_2 ... D_k: <compound tail> end gives rise to the following object programme:

$2^{26} \times 127 + 2^{23} + 2^{13} \times r + q$

(cf. adjust in table 1D), r = rank of, and q + 1 = lowest address occupied by, local simple variable of B

object programme of declaration D_1
object programme of declaration D_2
⋮
object programme of declaration D_k
the instruction retain of table 1E
object programme of the compound tail
$2^{26} \times 119 + 2^{23} + 2^{13} \times r + q'$

(cf. restore in table 1D), q' = highest address occupied by a local simple variable of B.

The dynamic introduction of the block B consists of the adjust instruction, the object programmes of the declarations D_k ($1 \leq i \leq k$) of local arrays being not own, and the instruction retain, which constituents are, in the object programme of B, linked together by appropriate pass instructions for passing by the object programmes of the other declarations D_i

The "object programme" of each declaration D_i of simple variables being not own, is empty. The object programmes of own variables are single locations. The object programmes of own array declarations consists of fixed spaces and fixed series of constants. Of course, all these object programmes may join one the other.

At a moment the object programme of the block B is going to operate, P', Q', and p be the values of the variables P, Q, and p of the interpreter. q' and q as mentioned above, are the highest and lowest (relative) addresses occupied by a local variable of B.

Then Q' is = p + q'

and the dynamic introduction of B must assign the value p + q to Q, which value is the highest address not occupied by a local variable of B.

Therefore the above adjust instruction makes the interpreter act, on S10L100, as follows:

Q := p + q

When there are local arrays being not own, the dynamic introduction of B reserves a space P' ... P" - 1 for them, assigning the value P" to variable P. Then P ... Q is again the space not yet used.

At the end of the dynamic introduction of B, the instruction retain makes the interpreter jump to S10L99, where there happens:

{Q + 1} := P

Thus the relative address q + 1 is occupied by a local internal variable of B which is used for retaining the value of P formed last. Thus this value is still known when P should temporily take another value.

As, with respect to the block (or procedure) in the text of which B occurs, the address q' + 1 has the same destiny as q + 1 has with respect to B,

{p + q' + 1}

is the value P had when the object programme of B was going to operate.

At the end of the object programme of the block B, the restore instruction makes the interpreter, on S10L102, act as follows:

Q := p + q' P := {Q + 1}

Then the values P' and Q' which were resident when the object programme of B was going to operate, are re-assigned to P and Q.

As the dynamic introduction of a block B reserves space for all simple variables and arrays which are local to B, the pointer values P" and Q" introduced by it are resident during the further course of the object programme of B. When S' is the object programme of any statement contained in the compound tail of B, the pointers P and Q may take other values while S' is acting; however, as soon as S' jumps to the object programme of any statement contained in the compound tail of B, the values P" and Q" must be restored. Above this restoration has been secured only in the case of S being a block left through end, or an assignment statement involving evaluations of expressions with storing partial results.

When the opening symbol begin, in front of any block or compound statement B, is pre-acting, there happens in scheme S3:
S := S + 2 {S - 3} := 0 {S - 2} := begin {S - 1} := 0.

When B is a block, the declarator occuring next to begin finds (cf. S3aL2) in the list L

{S - 3} = 0 {S - 2} = begin

indicating that the translation of the block B must still be prepared in L. Then there happens:

S := S + 4

{S - 7} is already = 0

{S - 6} = begin is no more important

{S - 5} := P which is the address P'

where the object programme of B must begin

{S - 4} := q = highest relative address q' to be occupied by a local simple variable of B

{S - 3} := T = highest address T' where an identifier declared in B will be listed in I

{S - 2} := begin

{S - 1} := 0.

As an adjust instruction must be stored later on address P', variable P is increased. The other declarators of the block B find in L

{S - 3} = T' ≠ 0 {S - 2} = begin

indicating that the translation of B has already been prepared in L.

When the opening symbol begin of B is after-acting with end, {S - 3} is assigned to variable c on S1L1. Then, on S3bL6, B is found to be a block and no compound statement. Thus, on S3bL9, the information listed for B by scheme S3a, is extracted from L, and the beginning value q' of q is re-assigned to q on S3bL12. As a similar re-assignment takes place too when the translation of any block contained in the text of B is finishing, the variable q has, on S3bL10, still, or again, the value

- 1 + (lowest address occupied by a local variable of B) so that the adjust- and restore instructions as required for B, can now be inserted in the object programme.

The identifiers defined in B, and the required contra-identifiers as will be discussed below, are listed in I on the addresses

T + 2, T + 4, T + 6, ... T'.

At the end of the after-action, on S3bL14, the value e is assigned to variable T, which value is the highest address where either an identifier defined in B or an obsolete contra-identifier is listed in I. When no contra-identifiers are present, then e = T' is assigned to T.

When declaring the identifier of a local procedure, switch, own variable or own array, the translator has reached an address P which either precedes, or is contained in, an interruption of the dynamic introduction of the block B. Therefore, on S4L 24, 37, 10 and 14, there is tested by procedure SOi if {S - 7} is still = 0 or not. If so, then {S - 7} = 0 is replaced by P and P is increased. Thus, at the beginning of each interruption, a place is reserved for a pass instruction to be inserted later by procedure SOh. Before translating a declaration of local arrays being not own, on S4L17, {S - 7} is tested by procedure SOh. If > 0, then {S - 7} is an address, and the addresses {S - 7} + 1 to P - 1 do not belong to the dynamic introduction of B, and there happens:

$$\{\{S - 7\}\} := 2^{26} \times 121 + 2^{23} + P$$

which is a bridging pass instruction, and {S - 7} is set to 0. When the statement being next to the heading of B is going to be translated, again a bridging pass instruction may be required. Then procedure SOh inserts also the instruction retain and assigns a negative value instead of 0 to {S - 7} (cf. for instance S2L1, S3L2 and S3bL1).

A label x is either an identifier or an integral number being not negative. The latter is supposed to be $< 2^{24}$ and is internally represented by $x + 2^{24} \times 63$. The colon, inserted between x and the statement to be labelled, is a sort of declarator of x.

Label x be local to a block B. After reading the colon declaring x, the input part goes to S3cL1. On S3cL5, x and its internal equivalent

$$e = 2^{23} \times 127 + 2^{13} \times r + P$$

are listed in I. The restore instruction

$$2^{26} \times 119 + 2^{23} + 2^{13} \times r + q$$

explained above, is inserted on address P. When there is a reference to label x, within a designational expression the resulting jump instruction

$$2^{26} \times 122 + 2^{23} + 2^{13} \times r + P$$

jumps in operation time to the above address P, supposing that it also restores the needed pre-value p. Then the restore instruction {P} re-assigns to the pointers P and Q the values as calculated by the dynamic introduction of B.

When a label x is referred to within the text of a designational expression E, then the label x is meant, which is defined in the text of the smallest block containing expression E in its interior. Thus, in begin D1; ...; x: S1; begin D; ...; go to x; ..; x: S end end go to x jumps to the second label x. Thus a reference to a label x may not be translated immediately after reading that reference, but must be delayed until the whole block in which x is defined, has been read.

After reading a reference to a label or a switch identifier x, the translator contra-declares x by listing in I:
the contra-identifier $x' = 2^{32} + x$
and its "internal equivalent" $e' = 2^{23} \times 127 + P$ in which P is the address where to insert a jump instruction later. Of course, P is increased. This happens on S8aL2, S8bL2 and S2aL17, by calling procedure S01.

Making contra-declarations is necessary too, when translating procedure statements. Table 2A shows all types of contra-declarations.

Thus, while translating a block B, identifiers as well as contra-identifiers are listed in I. When the opening symbol begin of B is after-acting with end, the cycle prepared on S3bL13 looks for the addresses where jump instructions etc. must still be inserted in the object programme of B. The cycle looks as follows:

a := T

cycle:

Look up next address a + 2 or a + 4, ... T' where a contra-identifier i' is listed in I.

If not present, then object programme of B is ready.

If i' is found, then a be its address.

$e' = \{a \leftarrow 1\}$ is extracted from I

$i = 2^{32} + i'$ is a normal identifier.

Look up address T + 2 or T + 4, ... T' where i is listed in I.

If i is found, its internal equivalent e is extracted.

When, for example, e' is = 2^{23} x 127 + P', then i may only represent a label or switch, e being either 2^{23} x 127 + 2^{13} x r + P or the internal equivalent of a formal parameter i. Then the contra-declaration i', e' can be <u>satisfied</u> and there happens: {P'} := 2^{26} x 122 + the logical product of e and 2^{26} - 1 so that a jump instruction is inserted on address P'. Going back to cycle, the next contra-identifier is looked up.

If i is not found, contra-declaration i', e' can not yet be satisfied and will thus be satisfied later. Thus i' must be regarded further on as to be not local to the translated block B. Therefore there happens:
{a} := {T'} {a-1} := {T'-1} {T'} := i' {T'-1} := e' a:= a - 2 t' := T' - 2.
Going back to cycle, the next contra-identifier is looked up.

Thus, when B is contained in the block or procedure B1, the contra-declarations which cannot be satisfied while translating B, are adopted as "locals" by B1 when the translation of B is ready.

0.16 Switch.

B be again the above block, and q + 1 be the lowest address occupied by a local variable of B.

A local switch declaration

<u>switch</u> i := F, G, ..., J with k entries

gives rise to the following object programme:

```
object programme of  F
object programme of  G
  :
object programme of  H
2^26 x 121 + 2^23 + |H|
  :        for  || cf. page 0.14.1.: if-then-else.
2^26 x 121 + 2^23 + |G|
2^26 x 121 + 2^23 + |F|
2^26 x 112 + 2^23 + 2^13 x r + q  (switch in t. 1D)
constant  k
```

and a switch designator i[E] gives rise to:

```
object programme of arithmetic expression  E
2^26 x 122 + 2^23 + 2^13 x r + x
```

{x} being the switch instruction.

In operation time, the latter object programme assigns the value of E to accu and goes to the switch instruction.

The switch instruction secures that accu has the integer representation.

There are 2 cases:

1 $0 < accu \leq k$

Then the switch instruction acts like the restore instruction, but jumps in addition to the pass instruction { x - accu}. Variable e2 is not changed.

2 accu is "out of capacity":

The values of chain2, p2, Q2 and {Q2 + 1} as left by the jump instruction on the address z = e2 - E0, are re-assigned to the variables chain, p, Q and P, e2 is cleared, and instr. { z + 1 } is executed.

e2 is cleared also by every restore instruction (cf. S10L101), and in the scheme ENTRY.

Jump instructions lead to either restore- or switch instructions. Variables e2 etc. are changed only when e2 = 0.

Then there happens on S10L56:
chain2 := chain e2 := e p2 := p Q2 := Q

For the translation of a switch declaration confer S4L37 and S8bL1. The list of pass instructions is at first arranged in the list L, decreasing variable S0, and is later stored in the object programme, when all switch list elements have been translated.

The switch list elements may be deliberate compound statements and blocks instead of designational expressions. In operation time, they join one the other, when no jump instructions are included. Thus an ssignment statement may be admitted only when enclosed within statement brackets.

0.17 Procedure chain, rank.

<main particle of a statement S>
≡ <each statement of the compound tail of a block or compound statement S>
| <the statement to be repeated in a for statement S>
| <each of the two or one statements to be selected in a conditional- or if statement S>

Fact 1.

If the statements S and S' have a main particle P in common, then S is equal to S'.
(If, for instance, S is a block, then P is not preceded by a for clause, thus S' is no for statement. Etc.)

<particle of a statement S>
≡ S | <main particle of a particle of S>

Fact 2.

If the statements S and S' have a particle P in common, then either S' is a particle of S, or S is a particle of S'.

Proof: If P = S or P = S', then S or S' is indeed a particle of S' or S. If $P \neq S$ and $P \neq S'$, then P is a main particle of particles Q and Q' of S and S'. According to fact 1 Q is equal to Q' so that S and S' have the particle Q in common. Etc.

Fact 3.

Within a text, each statement P determines a most containing statement M(P) of which P is a particle.

Proof: U, V, W, ..., Z be all statements of which P is a particle. According to fact 2 U and V are particles of the statement UV which is either U or V; thus U, V and W are particles of the statement UVW which is either UV or W; etc. Then UVW...Z is exactly the statement M(P).

Fact 4.

If a statement P occurs within the text of any procedure Q', then the statement M(P) is the body of a procedure Q. Procedure Q is contained in the text of Q' and may be equal to Q'.

Proof: If $M(P) \neq P$, then P is a main particle of a particle S of M(P). Then M(S) is equal to M(P), and it is easily verified that S occurs within the text of Q'; etc.

Thus statement M(P) occurs within the text of Q'. As M(P) is not main particle of any statement, M(P) must be the body of a procedure Q which, of course, is contained in the text of Q'.

A <u>chain</u> of procedures Q_1, Q_2, ..., Q_k ($k \geq 1$) satisfies the following conditions:

I Procedure Q_1 is not contained in the text of any other procedure.

II Each procedure Q_i ($1 < i \leq k$) is local to a block which is any particle of the body of procedure Q_{i-1}.

Fact 5.

Within a text, each procedure Q determines a procedure chain Q_1, ..., $Q_k = Q$ The number k of chain elements be the rank of procedure Q.

Proof: If Q itself satisfies condition I, then the chain of Q has only 1 element. If Q does not satisfy condition I, Q is contained in the text of any procedure Q'. Then the block B to which Q is local, is also contained in the text of Q'. According to fact 4, statement M(B) is the body of a procedure R, and Q and R together satisfy condition II. If R satisfies condition I, a chain has been found. Otherwise there exists a chain element preceding R. Etc. ..., R, Q and ..., R', Q may be two chains which have the element Q in common. The bodies S and S' of R and R' have the above particle B in common. According to fact 2, either S is particle of S', or S' is particle of S; and in addition, neither S nor S' is main particle of any statement. Thus S must be equal to S' thus procedure R = R', etc. Thus there exists only one chain having Q as its last element.

Rank.

<u>own</u> declared identifiers have the rank r = 0.

An identifier i whose definition does not belong to the text of any procedure, has the rank r = 0.

An identifier i which is local to any particle B of the body of a procedure Q, is a <u>body identifier</u> of Q. Unless i is declared <u>own</u>, i is regarded as to have the rank r of Q. When i is no label, particle B is, of course, a block. Though i is called "body identifier", its significance declared in B doea not hold in the body outside B. Labels may occur everywhere in the body, even when that is no block.

When i is a procedure identifier, the procedure concerned has the rank r + 1 which is in accordance with condition II of procedure chains. Otherwise the item represented by I has the rank r of i. The formal parameters of procedure Q are also regarded as to be body identifiers of Q. Thus all identifiers of rank r defined in the text of procedure Q are body identifiers of Q.

Fact 6.

A reference to an identifier i occurs in either a for- or if clause, an assignment-, go to- or procedure statement, or an array-, switch- or procedure declaration which declaration is local to a block, thus corresponds to one fixed statement iref.

When a body identifier i of a procedure Q is referred to, statement iref is, of course, contained in the text of Q. Procedure Q is an element of the chain $Q_1, \ldots, Q_k$ whose last element has the body M(iref).

Proof: According to facts 4 and 5, the procedure Q_k and its chain exist. Q_k is contained in the text of Q. If $Q_k = Q$, the proof is ready. Otherwise j be the smallest index for which the procedure Q_j is contained in the text of Q and is also $\neq Q$. As Q_1 is not contained in the text of another procedure, index j is > 1. Then the block B to which procedure Q_j is local is also contained in Q as is procedure Q_{j-1} whose body is M(B). As Q_{j-1} cannot be $\neq Q$, Q_{j-1} must be $= Q$, q.e.d. Object programmes of procedures and procedure statements.

A procedure
procedure f(x_1, ..., x_k); <specification part>;
<value part>; body $(k > 0)$
gives rise to the object programme:

instruction X of table 1E
D_1
: specification patterns
D_k
-2^{13} x rank of procedure f
object programme of body
instruction extract procedure of table 1E
(is present only in object programme of type proc.)
instruction Y of table 1E

Machine code jump instruction X jumps to label S11L0 of the "big transporter".

Each constant D_j has the form

$$D_j = 2^{23} \times \overline{0\ 0000v1tx0}$$

If bit v = 0, then parameter x_j is value, which is stated in operation time on S11L35. If bit x = 0, then there is specified, in the text, a type t for parameter x_j:
t = 1 → parameter is real
t = 0 → parameter is integer or boolean.
In operation time, bit t is tested on S11L25. When calculating the formal parameter keys of a procedure to be called, the transporter S11 extracts one constant D_j after the other, untill the negative constant $-2^{13} \times r$ is found, at the beginning of the body's object programme.

Machine code instruction Y makes the interpreter jump to S11aL1 and the "restorer" prepares the return from the object programme of the procedure.

For the translation of a procedure confer S4L24 ff and compound statement S 8d. The constants D_j are stored on S4L7 and are eventually modified on S4L32 ff.

When there are no formal parameters thus k is = 0, the object programme of procedure f has the form:

instruction X1 instead of X (cf. S8dL6)
$-2^{13} \times r$ etc. as above.

Machine code instruction X1 makes the interpreter jump to the "small transporter" of scheme S12.

A procedure statement (or function designator)
$f(y_1, \ldots, y_k)$ $(k > 0)$
gives rise to the object programme:

instruction F for calling the procedure
key address s
object programme of parameter y_1
⋮
object programme of parameter y_k
constant 0
key by-word | of actual parameter y_k
key main word |
⋮
key by-word | of actual parameter y_1
key main word |

If actual parameter y_j is an identifier or a constant, the object programme of y_j is empty.

When the procedure, called in, does not return to a label, it returns to the instruction {s} which is next to the object programme of the procedure statement.

The key (table 3) of actual parameter y_j occupies the addresses $s - 2xj$ and $s - 2xj + 1$. The key main word differs from 0. The constant 0 occupies the address $s - 2xk$. Because the keys appear in the reversed order k, k-1, ... 1, the key address s can be easily used for two different purposes.

If identifier f preceding the parenthesis (is a procedure identifier, then F is a machine code jump instruction leading to the object programme of procedure f. In operation time, instruction F and the next word s are extracted on S10L2 and assigned to the variables I and N. Instruction F jumps normally to instruction X which is the first word in the object programme of procedure f, and instruction X jumps normally to the transporter S11. The values of I and N inform about the beginnings of the mentioned lists of specification patterns and actual parameter keys.

Identifier f may also be a formal parameter of any procedure Q'. Then parameter f may, in operation time, only represent procedures. In this case instruction F is a prostate instruction (cf. table 1D) having the form:

$$F = 2^{26} \times 120 + 2^{13} \times r' + y'$$

in which r' and y' are the rank and the relative address of the internal variable which is the key by-word of formal parameter f. When procedure Q' is called, the transporter calculates the key of parameter f which parameter must represent a procedure Q, and stores the key in the working space of Q' according to y' and r'. Whenever the above instruction F occurs in the object programme of Q', the interpreter proceeds via S10L13 and extraction of the key of parameter f to S10L40. The key of f reveals the address where the object programme of procedure Q begins. Compound statement S11 is joined for calling procedure Q.

The above procedure statement is translated under direction of compound statements S5 and S5a. As identifier f may be the identifier of a procedure, which, at that moment of the translation time, has not yet been translated, identifier f is contra-declared, and the instruction F is inserted in the object programme

as late as on S3bL40 ff. As the list of actual parameter keys may not be stored in the object programme, unless the object programmes of all actual parameters have been completed, the list is at first assembled on addresses S0 - 1, S0 - 2, ... of the list L, by which intermediate storing the order of the keys is automatically reversed.

When there are no actual parameters thus k is = 0, the function designator f is not characterized as such by a pair of parentheses in the text. The object programme of f reduces (cf. for instance S1L13) to the machine code jump instruction F leading to the instruction X1 in the object programme of a procedure f having no formal param. No key address s is present, and the object programme of procedure f returns to the word next to F.

0.18 The internal variables of a procedure.

When, in operation time, a procedure f is going to be called, the space P...Q as shown by the pointers is available for storing information.

When procedure f is called directly i.e. not through any formal parameter representing f, a machine code jump instruction F leads to the object programme f' of f.

When procedure f has no formal parameters, f' jumps, by the instruction X1, to scheme S12. A key address s which may, in this case, be any address > 1, is introduced.
e + 1 is assigned to {Q - 2}, being the (negative) extraction instruction for the return to the word which is next to instruction F. The further call is obtained from the description below by taking k = 0.

When procedure f has formal parameters x_1, ..., x_k, object programme f' jumps, by the instruction X, to compound statement S11. The key address s to be introduced is the word next to F in the object programme of the procedure statement $f(y_1, \ldots, y_k)$. F and s have already been extracted from the object programme. The call of procedure f is performed by the transporter. There happens the following:

{Q - 2} := P
{Q - 3} := p
{Q - 4} := chain
{Q - 5} := key main word
{Q - 6} := key by-word
} of formal param. x_1

:

{Q - 2xk - 3} := key main word
{Q - 2xk - 4} := key by-word
} of param. x_k

the value of a value parameter which is no array, is stored as the by-word of its key.

When value arrays are present, they are stored on addresses P, P + 1, P + 2, ... and pointer P must be adjusted.

p := Q - Q0 - 5
chain := s + 2^{13} x r
-2^{13} x r being found in object programme f'
Q := Q - 2xk - 6
{Q + 1} := present value of P

as happens also at the end of the dynamic introduction of a block

the object programme following the constant -2^{13} x r is executed.

The transporter has derivied the keys of the parameters x from those of the corresponding parameters y (cf. table 3). Again P...Q is the free space.

Variable chain does not change its value, unless a procedure is called or returning. When no procedure is operating, the value of chain is not important.

As Q0 + p is the address where the key main word of the above parameter x_1 has been listed in the working space of procedure f, the relative addresses

$$Q0 - 2xj + 1 \quad \text{and} \quad Q0 - 2xj + 2$$

have been reserved for parameter x_j in translation time (cf. S4L29 and 8). An instruction referring to parameter x_j has the form

$$I = 2^{26} \text{ x } \ldots + 2^{13} \text{ x r} + Q0 - 2xj + 1,$$

thus pointing to the key by-word. r is the rank of procedure f. The bits I_7 to I_9 are 0. When I is interpreted, the test on S10L13 succeeds and the key of parameter x_j is extracted from the working space of procedure f.

At the call of procedure f, the previous value of variable chain is stored on address Q0 + 1 + p and may thus be regarded as an internal variable with relative address Q0 + 1 and rank r of procedure f.

The internal variables with relative addresses Q0 + 4 and Q0 + 5 are occupied only in the case of a type procedure f, for retaining the values of variables mant and exp when assignment to the function name f is required. In the object programme f' the assignment f := accu is represented by the store procedure instruction

$$2^{26} \text{ x } 113 + 2^{23} \text{ x } \overline{t01} + 2^{13} \text{ x r} + Q0 + 3$$

which joins the partres instruction, therefore having the address part Q0 + 3 instead of Q0 + 4. In the object programme of a type procedure f the second last word is the instruction extract procedure of table 1E. Before the return, it assigns to accu the value earlier assigned to the function name. There happens:
mant : = {p + Q0 + 4} exp := {p + Q0 + 5}.

When a procedure f does not return to a label, it returns normally through the instruction Y which is the last word in the object programme f' and jumps to the restorer S11a.

There the return is arranged. When procedure f has formal parameters, there happens:

e := E0 + address s taken from the value of variable chain

Q := p + Q0 + 5

P := { Q - 2 }

p := { Q - 3 }

chain := {Q - 4 }

and the instruction {s} is executed. When procedure f has no formal parameters, the value of {Q - 2} is assigned to e for the return, while the value of P is not changed.

In both cases, after a normal return from a procedure f the variables P, Q, p and chain have again the values they had immediately before the call of f. The whole range P ...Q is again free.

0.19 Looking up deliberate variables.
Formal parameters representing procedures and expressions.

The base of a procedure Q be the remainder (body of Q minus procedures declared in the body).

Basis instruction of Q be each instruction contained in the object programme of the base of Q.

During the operation of a procedure Q the values, assigned to the variables p and chain at the call of Q, must still, or again, be present whenever a base instruction I of Q is on the point of being executed.
- How otherwise could the body variables of Q be quickly accessible? (If those values of p and chain are present before executing a base instruction I which is a direct call of a procedure R, and R normally returns to the base of Q, then the required restoration is indeed performed by the restorer).

i be any item having the rank r' and being no procedure. Then an instruction I which refers to i has the form:

$$I = y + 2^{13} \times r' + \ldots$$

If r' = 0, then the pre-value 0 is required for i.

If r' > 0, the interpreter finds the values of p and chain as required for the item i as follows (cf. S10L9 to 11):

```
        s1 := chain
        p1 := p
cycle:  if rank r' = rank r contained in
      s1 = 2^13 x r + key address
      then the cycle is ready.
      otherwise there happens
        s1 := {p1 + Q0 + 1}
        p1 := {p1 + Q0 + 2}
        and the cycle is repeated.
```

The values of p1 and s1 as left by the cycle are the required values.

In compound statement S10 the cycle is still simplified by the fact that never r' is greater than r.

When the above item i is a simple variable, it occupies the absolute address p1 + y.

When i is a label thus I is a jump instruction, the values of p1 and s1 are assigned to p and chain on S10L57.

Then instruction {y} to be executed next is a restore- or switch instruction and assigns to pointers P and Q the values required.

Proof of the above tracing process.

When a procedure Q is operating, an instruction I1 must have called Q for that operation. If I1 is contained in the object programme of any procedure, I1 is a base instruction of one procedure P2 and P2 was operating before instruction I1 called procedure P1 = Q. An instruction I2 has called P2 for that operation. Etc. Thus procedure Q and its operation define a chain of calling instructions and called procedures:

$$I_m, P_m; \ldots; I_2, P_2; I_1, P_1 = Q$$

Instruction I_m is not contained in the object programme of a procedure. Each instruction I_j calls procedure P_j and is, when $j < m$, a base instruction of procedure P_{j+1}. At the call I_j there happens:

$$\text{chain} := c_j = s_j + 2^{13} \times \text{rank } r_j \text{ of procedure } P_j$$
$$p := p_j \qquad (m \geq j \geq 1)$$

At first there be supposed that each procedure P_j is called <u>directly</u> i.e. called without using a formal parameter representing P_j. Then all I_j are machine code jump instructions, and the pairs c_j, p_j ($j = m+1\ (-1)\ 1$, $p_{m+1} = 0$) form a chain, defined by the relations

$$c_{j+1} = \{ p_j + Q0 + 1 \}$$
$$p_{j+1} = \{ p_j + Q0 + 2 \} \qquad (1 \leq j \leq m)$$

As procedure Q is supposed to be operating, chain and p have the values c1 and p1.

According to fact 5 on page 0.17.2. procedure Q defines a procedure chain

$$Q_1, \ldots, Q_k = Q$$

which, as will be shown, is contained in the row

$$P_m, \ldots, P_1 = Q$$

but may differ from that row.

As procedure Q may not be directly called from outside the text of Q_{k-1}, instruction I_1 is contained in the object programme of Q_{k-1} thus procedure P_2 is contained in the text of Q_{k-1}. If $P_2 \neq Q_{k-1}$, then P_2 will be indicated now by $Q_{k,1}$ and procedure P_3 is contained in the text of Q_{k-1}. If $P_3 \neq Q_{k-1}$, then P_3 will be indicated by $Q_{k,2}$ and P_4 is contained in the text of Q_{k-1}. Etc. Thus the row $P_1, \ldots, P_m$ takes the form:

$Q_k, Q_{k,1}, Q_{k,2}, \ldots; Q_{k-1,1}, \ldots; \ldots; Q_1$

in which the double subcripted symbols may be absent. .

Each procedure $Q_{j+1,h}$ $(0 < j < k,\ 0 < h)$ is contained in the text of procedure Q_j, and its rank $r_{j+1,h}$ is greater than the rank j of Q_j.

I be a base instruction of procedure Q referring to an item i which is no procedure and has the rank r'. According to fact 6 on page 0.17.3. identifier i is body identifier of element Q_r, of the procedure chain defined by Q. Within the row

$$k,\ r_{k,1},\ r_{k,2},\ \ldots,\ k-1,\ r_{k-1,1},\ \ldots$$

of the ranks, r' is indeed the first element which is not greater than r', and it is extracted from the chain together with the values c_r, and p_r, corresponding to the considered operation of procedure Q_r,.

Up to now each I_j has been supposed to be direct call of procedure P_j.

In the general case, an instruction I_j may also, like the prostate instruction, refer to a formal parameter of a procedure Q', which parameter represents, from the moment Q' was called, procedure P_j.

However, the above chain relations

$c_{j+1} = \{p_j + Q0 + 1\}$ and $p_{j+1} = \{p_j + Q0 + 2\}$

may be applied only, when I_j is a direct call of procedure P_j.

Proof.

Q1, Q2, Q3 be a procedure chain. The procedures Q1 and Q01 be local to the same block, f be the formal parameter of procedure Q01. A body variable v1 of Q1 be referred to in procedure Q2. A procedure statement Q01(Q2) may occur in the text of Q3, giving rise to a machine code jump instruction I01. A procedure statement f(...) in the text of Q01 gives rise to a prostate instruction I02 which calls procedure Q2. Now the following calls are considered:

I1, Q1, I2, Q2, I3, Q3, I01, Q01, I02, Q2

in which each subsequent I is a base instruction of the procedure preceding it. Only I02 is no direct call. When now the above relations would be realized also for the call I02, a wrong address would be formed. For the variable $v^{Q}_{=}$ is referred to also during the "youngest" operation of Q2 thus after the call I02, v1 has the rank 1 of procedure Q1, and in the chain value c01 extracted first, the rank 1 of procedure Q01 is found. Thus the tracing process stops, delivering the pre-value p01 corresponding to the operation of Q01 instead of the required pre-value p1 corresponding to the operation of Q1.

Thus the above relations may not be applied to the call I02, q.e.d.

The following prescriptions serve for avoiding mistakes of the above type and also for speeding up the tracing process:

1 In formal parameter keys may only occur absolute addresses. Thus, when calling a procedure, the transporter must bring the offered actual parameter keys to an "absolute" form.

2 When a formal parameter f of a procedure represents any procedure Q_r which, of course, defines a fixed procedure chain $Q_1, \ldots Q_r$, the reference of f to Q_r is possible only because, at any time, procedure Q_{r-1} has already operated. When that operation was interrupted, the corresponding values c_{r-1} and p_{r-1} were stored on any addresses y and y + 1, and there is supposed now that the address y is contained in the key main word of parameter f. When r = 1, then y is = Q0 + 1 while {Q0 + 2} is understood to be a constant 0 of the interpreter.

3 When formal parameter f represents an expression E, then E is contained in the base of a fixed procedure Q_{r-1}. When the operation of Q_{r-1} is interrupted by the designator of which E is an actual parameter, the same happens as above in 2, and again address y is contained in the key main word.

4 The formal parameter f occuring in a designator f(...) represents nothing but procedures having formal parameters. The prostate instruction concerned makes the interpreter proceed to S10L40, where there happens:

{Q - 1} := the key address of the procedure stat.
{Q - 2} := P
{Q - 3} := p
{Q - 4} := chain
{Q - 8} := {y + 1}
{Q - 9} := {y}

which is joined by the formal parameter keys as described on page

p := Q - Q0 - 10 instead of 5
chain := 2^{13} x r + the key address 0 etc.

Thus, when procedure Q_r is operating, {p + Q0 + 1} is the value c_{r-1} mentioned in 2 above, while the items of rank r - 1, referred to in Q_r, are precisely the body items of the mentioned procedure Q_{r-1}.

When procedure Q_r returns, the key address 0 contained in the value of chain, makes compound statement S11a assign the value p + Q0 + 10 to variable Q. chain, p, and P are restored, and {Q - 1} is the key address for the return.

5 When an instruction I refers to a formal parameter f which represents either an expression or a procedure having no formal parameters, then, on S10L42, the information for the return, including the instruction I itself, is listed in 6 locations, and Q is replaced by Q - 6.

When f represents an expression, there happens also:

chain := {y}

p := {y + 1}

in accordance with 3 above. The expression object programme is executed which, when it is not designational and the instruction return of table 1E is the last word, jumps to S10L46 for the return.

When f represents a function, there happens:

{Q - 3} := {y + 1}

Q - 4 := {y}

Q := Q - 5 p := Q - Q0 chain := 2^{13} x r + 1 etc.

When returning, the key address 1 makes compound statement S11a jump to S10L46.

0.20 Arrays.

In an array declaration

array a, b, ..., c [f_k: g_k, f_{k-1}: g_{k-1}, ..., f_1: g_1], d, e, ... [...], ...

the values of all bound expressions f and g are either integers, or must be rounded to integers. For each suffix $i \leq k$ f_i must be $\leq g_i$ In operation time, the dynamic introduction of a block B reserves space for each local array of B which is not own. The bound values as calculated then are not changed during the further operation of the object programme of B.

Each of the above arrays a, b, ..., c contains

$$H_k = h_1 \times h_2 \times \ldots \times h_k$$

elements, in which each h_j is equal to

$$h_j = g_j - f_j + 1.$$

When the dynamic introduction of B is going to reserve space for array a, the space P...Q is not yet occupied. The first and last elements of a may occupy the addresses P and $P + H_k - 1$. In general, for the subcripted variable

$$a[x_k, x_{k-1}, \ldots, x_1]$$

the following address is reserved:

$$[a[x_k, x_{k-1}, \ldots, x_1]] = P + \sum_{i=1}^{k} (x_i - f_i) \times \prod_{j=1}^{i-1} h_j$$

Then is: $[a[f_k, f_{k-1}, \ldots, f_1]] = P$,

$$[a[g_k, g_{k-1}, \ldots, g_1]] = P + H_k - 1$$

With $u = \sum_{i=1}^{k} f_i \times \prod_{j=1}^{i-1} h_j$ and $[a] = P - u$

the above address formula takes the form

$$[a[x_k, x_{k-1}, \ldots, x_1]] = [a] + \sum_{i=1}^{k} x_i \times \prod_{j=1}^{i-1} h_j$$

$$= \{ \ldots [(x_k \times h_{k-1} + x_{k-1}) \times h_{k-2} + x_{k-2}] \times \ldots \vdots \} \times h_1 + x_1 + [\alpha]$$

and a similar formula exists for u.

[a] will be called the pre-value of array a. If the array contains the element a[0, 0, ..., 0], then [a] is the address reserved for that element.

[a], [b], ..., [c],
H_k, u

and the k - 1, subscript factors

h_{k-1}, ..., h_1

as calculated by the dynamic introduction of B are regarded now as the values of the internal local integer variables of B having the relative addresses

q, q - 1, ..., q' + 2,
q' + 1, q',
q' - 1, ..., q' - k + 1

If k = 1, then the subscript factors are absent.

v be an auxiliary variable.

When applying the notation of a for statement, the calculations to be performed by the dynamic introduction of B take the form:

```
u := f_k
H_k := 1 + g_k - u
for i := k - 1 step - 1 until 1 do
begin
  v := f_i
  h_i := 1 + g_i - v
  H_k := H_k x h_i
  u := u x h_i + v
end
  [a] := P - u
  P := P + H_k
  [b] := P - u
  P := P + H_k
    :
  [c] := P - u
  P := P + H_k
```

f_i and g_i are no subscripted variables but expressions, and the object programme concerned is no cycle but a stretched programme, in which the piece of text obtained for j = 1 precedes the one obtained for j = 2, etc. The variable v has been replaced by variable [c]:

object programme of expression f_k
$2^{26} \times 117 + 2^{23} + 2^{13} \times r + q' + 1$
i.e. instruction u := accu
object programme of expression $g_k - u$
$2^{26} \times 115 + 2^{23} + 2^{13} \times r + q'$
i.e. store factor H_k
constant 0
object programme of expression f_{k-j}
$2^{26} \times 117 + 2^{23} + 2^{13} \times r + q' + 2$
i.e. instruction [c] := accu
object programme of expression $g_{k-j} - [c]$
$2^{26} \times 115 + 2^{23} + 2^{13} \times r + q' - j$
i.e. store factor h_{k-j}
constant j + 2
$2^{26} \times 114 + 2^{23} + 2^{13} \times r + q$
i.e. store pre-value [a]
constant q' - q + 1

A store factor instruction referring to the absolute address y, operates as follows (cf. S10L91):

if accu is not yet an integer, then accu is rounded and integer representation introduced.
{y} := accu [= mant]
if N = next constant in the object programme differs from 0, then there happens in addition:
y := y + N which is the address of [c]
{y - 2} := {y - 2} x mant
which is the next value of H_k
{y - 1} := {y - 1} x mant + {y}
which is the next value of u

and when y and N have the same meaning as above, the store pre-value instruction operates as follows (cf. S10L96):

N := N + y which is the address of u
I := {N - 1} which is H_k
J := {N} which is u
cycle: {y} := P - J
P := P + I
y := y - 1
if y $\neq$ N, then go back to cycle.

In the list I, the above array declaration has the form

identifier a
$2^{23} \times \overline{0\ 011111t01} + 2^{13} \times r + q$
identifier b
$2^{23} \times \overline{0\ 011111t01} + 2^{13} \times r + q - 1$
⋮
identifier c
$2^{23} \times \overline{0\ 011111t01} + 2^{13} \times r + q' + 2$
factor identifier $2^{24} \times 62$
$2^{26} \times 126 + 2^{23} + 2^{13} \times r + q' - 1$
(cf. ar1 in table 1D)
identifier d
$2^{23} \times \overline{0\ 011111t01} + 2^{13} \times r + q' - k$
etc.

The factor identifier may be any other positive number differing from identifiers and numerical labels so that procedure SOf cannot be disturbed by it when looking for an identifier. In the above set of internal equivalents, the ar1 instruction can be found by procedure SOf as the first negative word.

For the translation of an array declaration confer S4L11 to 20, S6L3 to 9, and S6aL3 to 42. own arrays are supposed to have constant bounds. As the translator calculates the pre-values and subscript factors concerned, treating them as programme constants, these constants need not be calculated in operation time.

The address formula of a subscripted variable

$$a[x_k, x_{k-1}, \ldots, x_1]$$

gives rise to the following object programme (e and ef be the internal equivalents of array identifier a and the corresponding factor identifier):

object programme of expression x_k
ef which ar1 instr. refers to first subscript factor
appropriate object programme of x_{k-1}
ef - 1 — second subscript factor
⋮
ef - k + 2 — last subscript factor
appropriate object programme of expression x_1
$2^{26} \times 125 + 2^{23} \times \overline{t01} + 2^{13} \times r + q = 2^{26} \times 94 + e$

which ar2 instruction refers to pre-value [a] and contains the type indication of array a next word N. If k = 1, this

object programme reduces to the object programme of x_k and the ar2 instruction.

If an expression x_i $(i < k)$ is a simple variable, a formal parameter, or a constant, then the appropriate object programme of x_i is the single instruction

accu := accu + x_i

otherwise that appropriate object programme has the form:

instruction partres of table 1E
normal object programme of expression x_i
accu := accu + partial result.

Though ar1- and ar2 instructions are in fact calculative, they are yet listed in table 1D which corresponds to the switch on S10L15. y be the absolute address of the required subscript factor or pre-value. Then these instructions make the interpreter act as follows:

ar1 instruction (cf. S10L62):

if accu has not yet the integer representation, it is rounded to an integer and transferred.
mant := mant x {y}
in which {y} is the required subscript factor.

ar2 instruction (cf. S10L65):

the same rounding and transfer as above.
y := accu + {y}
In which {y} is the required pre-value.
now y is address where subscripted variable is located.
If N = 0
then return from the object programme of an actual parameter being a subscripted variable is arranged (cf. S10L74).
If $N \neq 0$ and $N \neq -1$, then accu := {y} (cf. S10L68).
In this case N is no constant but an instruction and requires the value of the subscripted variable.
if N = - 1,
then the store accu instruction
2^{26} x 117 + 2^{23} x $\overline{t01}$ + y is formed and stored as a "partial result". type indication t is copied from the ar2 instruction. (cf. S10L69) thus assignment is prepared.
The object programme following the constant - 1 calculates the value to be assigned to the subscripted variable (and may also perform assignments).

The next instruction, extract address of table 1E, extracts the stored store accu instruction for execution (cf. S10L72).

In a subscripted variable

$$a[x_k, x_{k-1}, \ldots, x_1]$$

identifier a may also be a formal parameter of any procedure Q. Then the key by-word of parameter a is an internal body variable of procedure Q with a relative address χ and rank r. Of course, when Q is operating, parameter a represents an array. When E is the key by-word and y is the address part of the key main word, then $\{y\}$ and $\{E + 1\}$ are the pre-value and the number of elements of that array (cf. table 3). In this case, the object programme has the form:

object programme of expression x_k
$2^{26} \times 126 + 2^{13} \times r + x$
constant 0
appropriate object programme of x_{k-1} as above
$2^{26} \times 126 + 2^{13} \times r + x$
constant 1
⋮
$2^{26} \times 126 + 2^{13} \times r + x$
constant k - 2
appropriate object programme of x_1
$2^{26} \times 125 + 2^{13} \times r + x$

Thus all ar instructions refer to the key by-word of parameter a. On S10L30 to 33 the required addresses y and E are obtained. The test on S10L36 succeeds. In the case of the ar1 instruction, E minus the programme constant N is the address where the required subscript factor is listed.

0.21 for statement.

When X is a piece of text, then |X| be again the address where the object programme of X begins.

The for statement

for v := ε_1, ..., ε_h do T

in which ε represents the for list elements, gives rise to the object programme:

object programme of ε_1
2^{26} x 124 + 2^{23} + |T|
which is a for instruction of table 1D
object programme of ε_2
same for instruction as above
⋮
object programme of ε_h
same for instruction as above
2^{26} x 121 + 2^{23} + |etc|
object programme of T
the instruction for0 of table 1E
object programme of etc

The object programme of a for list element

ε = F = expression:

object programme of v := F
the instruction for2 of table 1E

The object programme of a for list element

ε = F while G:

the instruction for2 of table 1E
object programme of v := F
object programme of G

The object programme of a for list element

ε = F step G until H :

object programme of F
2^{26} x 123 + the logical product
of 2^{26} - 1 and the internal equivalent of v (cf. for1 in table 1D)
object word of v:= accu
object programme of G
the instruction for3 of table 1E
object programme of H - v

Interpretation of the above instructions:

A for1 instruction
prepares a cycle governed by a step element (cf. S10L106):

{P} := e

by which extraction instruction the for0 instruction can return to the object word of v := accu next to the for1 instruction.

{P + 1} := object word of accu := accu + v to be derived from the for1 instruction itself. This is negative.

P := {Q + 1} := P + 4

thus cycle occupies 4 places in the working space.

The instruction for2
prepares a cycle governed by either a while- or expression element (cf. S10L105):

mant := any positive number

{P} := e (cf. for1 above)

{P + 1} := {P + 2} := instruction for2

but may also be another positive number

P := {Q + 1} := P + 4

The instruction for3
stores increment of count (cf. S10L111):

{P - 2} := mant

{P - 1} := exp

A for instruction
decides if the cycle concerned must still go on or not (cf. S10L108):

if {P - 2} < 0, then mant := - mant

this test fails when for instruction is preceded by object programme of while- or expression element.

If mant > 0, then object programme of statement T is executed. When for instruction is preceded by object progr. of an expression element, the test succeeds only when for instruction is executed first time.

P := {Q + 1} := P - 4

and instruction next to for instr. is executed. Addresses P to P + 3 are no longer occupied.

The instruction for0
returns to the word succeeding either the for1 instruction or instruction for2 which has prepared the operating cycle (cf. S10L112):

e := {P - 4}

I := {P - 3}

mant := {P - 2}
exp := {P - 1}
if I is negative:
then the cycle of a step element is operating. Then at first the object word of accu := accu + v is executed for adding the increment to the count and extraction instruction e + 1 is introduced for extracting the object word of v:= accu.
If I is positive:
then mant := any negative number which makes, in the case of an expression element, the for instruction finish the cycle. Extraction instruction e + 1 is introduced.

The translation of a for statement is directed by information occupying the addresses S - 5 to S - 3 of the list L, which precede the opening symbol {S - 2} (cf. compound statements S7 and S7a). These addresses are used as follows:

S - 5:

Pre-action of for:

{S - 5} := 0 say A_0

After-action of for with comma:

{S - 5} := A_j

being the first address after the object programme of for list element $\varepsilon_j = F_j \ldots$ (j = 1, 2, ...) just read.

$\{A_j\} := A_{j-1}$

thus the locations where to insert the required for instruction later, are linked together by an address chain, of which $A_0 = 0$ is the last element.

After-action of for with do:

address |T| = P + 2 is known now. The for instruction $2^{26} \times 124 + 2^{23} + |T|$ is inserted on the addresses A_j as well as |T| - 2.

{S - 5} := - |T| + 1

so that {S - 5} is negative at the after-action of for with the closing symbol.

A place is reserved for the pass instruction.

After-action of for with the closing symbol:

instruction for0 is added to the object programme.

The pass instruction $2^{26} \times 121 + 2^{23} + P$ is inserted on the address -{S - 5}.

S - 4:

Pre-action of for:

{S - 4} := 0

After-action of for with separation symbol := (cf. S9L4):

{S - 4} := internal equivalent of controlled variable v for translating the store accu- and for1 instructions.

S - 3:

Pre-action of for, and after-action of for with comma:

{S - 3} := - B_j

which is the complement of the address where the object programme of element ε_j to be read next must begin. Then is: $B_j = A_{j-1} + 1 \quad (j > 1)$.

After-action of for with step or while:

then {S - 3} = - B_j is negative.

{S - 3} := + C_j

which is the first address after the object programme of expression F_j just read.

{C_j} := B_j

{C_j}+ 1 := 2^{26} x (117 - 63) + {S - 4}

= instruction v := accu

After-action of for with until:

then {S - 3} = C_j is positive.

{S - 3} := 0

{C_j} := 2^{26} x (123 - 63) + {S - 4}

which is the required for1 instruction. Instruction for3 is added to the object progr.

After-action of for with comma or do:

for the element ε_j just read there are 3 possibilities:

1 {S - 3} = - B_j is negative

element ε_j is an expression.

The object word of v:= accu, and the instruction for2 of table 1E, are added to the object programme.

2 {S - 3} = + C_j > 0:

element $\varepsilon_j = F_j$ while G_j.

The object programme of expression F_j, occupying the addresses {C_j} = B_j ... C_j - 1, is shifted over one place, and the instruction for2 of table 1E is stored in front of it on the address B_j.

3 {S - 3} = 0.

element $\varepsilon_j = F_j$ <u>step</u> G_j <u>until</u> H_j.

The object word of accu := accu - v is added to the object programme of expression H_j

From the above sign indications it is clear that, in table 1B, the delimiters

<u>for</u> <u>step</u> <u>until</u> <u>while</u>

may be represented by the same value, for instance 324, without disturbing the translator. The delimiters <u>do</u> and := have the value 323.

0.22 Verify instructions. Contra-declarations, made when actual parameters are translated.

An actual parameter i which is a single identifier, may have the significance of a procedure, switch or label, and that item may be defined or declared later in the text to be translated. Thus, when i is read in the actual parameter list, the significance of i need not yet be contained in the declaration list I. Thus, instead of trying to form the parameter key, the translator makes an appropriate contra-declaration of parameter i according to the value $p = \overline{0\ 000000000}$ listed in table 2A together with detailed information. On the address x thus indicated, the key main word is inserted later when the significance of parameter i has been found.

A constant actual parameter i' with integer representation and being within the bounds $0 \leq i' < 2^{24}$, may have the significance of a label, and again that label may be defined later. As i' can also be a constant for use in arithmetics, the translator forms, and stores, the appropriate parameter key which contains constant i' as its by-word, but makes also a contra-declaration of a label i' to be expected according to the value $p = \overline{1\ 000000000}$ listed in table 2A. When no label i' is defined in the text, the key of the arithmetic constant parameter i' is maintained in the object programme, and the contra-declaration of i' , it has a negative internal equivalent , is never satisfied. Otherwise the key main word of parameter i' is replaced by a special key main word

$$2^{23} \times \overline{0\ 001011001} + \ldots$$

referring to the occurred label i' (cf. tabel 3).

In operation time, a formal parameter f of any procedure Q may represent an actual constant parameter i' which might be a label. When being executed, an instruction I referring to parameter f, makes the interpreter proceed to S10L83 and is examined there:
If I is found to be a jump instruction, apparently a jump to label i' must be performed. Otherwise instruction I needs constant i' for an arithmetic purpose. The interpreter can also adjust the key of actual parameter i' in the object programme, which has a time saving effect.

An actual parameter may have the form

E ≡ if boolean then i else j

E' be the object programme of expression E.

If i is an identifier, then again the significance of i need not yet be contained in I, when actual parameter E is being translated. However, when i is a simple variable or a formal parameter, its significance has already been recorded in I. x be the address next to the test instruction contained in E'. i is looked up. Depending on i having either the significance of a simple variable or formal parameter, or not, the translator stores, on address x, either the extract normally- or verify instruction referring to i, or 0. A contra-declaration of i must refer to the address x, and is made according to the value $p = \overline{0\ 111111111}$ listed in table 2A. When later i is found to be a local label or procedure, the appropriate jump- or machine code instruction is inserted on the address x as mark from the contra-declaration of i, and may, of course, destroy an extract normally- or verify instruction referring to a non-local simple variable or formal parameter i. Thus, ultimately, {x} is either an extract normally- or jump- or machine code- or verify instruction.

The instruction "verify formal parameter i" has the form $v = 2^{26} \times 110 + 2^{23} + 2^{13} \times r + y$, containing the bit $v_9 = 1$, though y and r are the address and rank of a formal parameter i. When being executed, instruction v makes the interpreter proceed to S10L140, before the key of parameter i is extracted. In operation time, a formal parameter f of any procedure Q may, from the moment Q is going to operate, represent the above expression E. When being executed, an instruction I which refers to parameter f, makes the interpreter proceed to S10L42. Then the instruction I is temporarily stored together with other information, and the object programme E' is executed. At the time the operation of E' proceeds to instruction v, E' is nearly on the point of returning, and

6 + value of pointer Q

is exactly the address where instruction I has been stored. Thus the instruction I can be examined on S10L140:
If I is found to be a jump instruction, then the formal parameter i must apparently represent any label, and instruction v must be interpreted as the jump instruction referring to formal parameter i. In addition, that jump instruction can be

inserted in the object programme in the place of the slow verify instruction v. If I is no jump instruction, expression E is apparently supposed to supply a value for an arithmetic purpose. Then, however, that value must be supplied by the formal i, and instruction v must be interpreted (and can, in the object programme, be replaced by) the extract normally instruction referring to formal parameter i.

When, in the above expression E, i is a constant which is suitable to be used as a label, the translator makes according to the value $p = \overline{1\ 100000000}$, listed in table 2A, a contra-declaration of the label i to be expected for reference to the above address x. And there happens also:

$\{x\} := 2^{26} \times 98 + 2^{23}$

$\{x+1\}$:= the arithmetic constant i

When no label i is defined in the text, the instruction $\{x\}$ for extracting the constant i is maintained, and the contra-declaration of i, having again a negative internal equivalent, is never satisfied. Otherwise the extract normally instruction is replaced later by the Verify instruction

$2^{26} \times 109 + 2^{23} + \ldots$

which refers to the occurred label i, so that, in operation time, there can be decided if i is an arithmetic constant or a numerical label.

Table 2A contains two types of contra-declaration with negative internal equivalent. Both refer to label-like constants occurring in actual parameters, and they need not be satisfied. When, on the other hand, a contra-declaration with positive internal equivalent can not be satisfied, the translated text is wrong.

When, in the above expression E, j is an identifier or label-like constant, j is treated quite similarly. Then x is the address next to the pass instruction contained in the object programme E'.

An actual parameter may have the form

i[E] or <u>if</u> boolean <u>then</u> i[E] <u>else</u> etc.

x be the address next to the object programme of the subscript expression E. While translating i[E], i is again looked up: Depending on i having either the significance of an array or a formal parameter, or not, either the ar2- or VERIFY instruction referring to i, or 0, is stored on the address x.

A contra-declaration of i must refer to the address x, and is made according to the value p = $\overline{0\ 011111111}$ listed in table 2A. When later i is found to be a switch identifier, the appropriate jump instruction is inserted on the address x.

In operating time, the VERIFY instruction

$$2^{26} \times 108 + 2^{23} + \ldots$$

referring to a formal parameter i, examines if parameter i is representing an array or a switch.

An actual parameter E which has the structure of a designational expression, may yet be arithmetic or boolean. Within an expression E, labels and switch identifiers may occur only in certain "key positions". When the translation of E proceeds to such a key position, the value 1 is assigned to the signal mark, as is shown in table 4C. Each identifier i occupying a key position in E, is contra-declared and is definitively interpreted later, when the translation of the block or procedure to which i is local, is finishing.

Table 1.

Operators.

When reading an operator, input goes to S1L1 after assigning the operator's value to variable f of table 4A.
That value has the form:
$32 \times r + 2^{26} \times q$ $(0 < r < 10,\ 64 \leq q < 96$ or $q = 97)$.
(For the value $q = 96$ of minus cf. S1L3).
r is the rank of the operator, and $2^{26} \times q$ is the operational part of the instruction resulting from the operator.

Below q is tabled only for the progressive version of operators, from which the regressive version is obtained by inverting the right-most bit of q. However, the progressive version $2^{26} \times 97$ of not ($2^{26} \times 96$ of - in the extractive sense of S1L3) corresponds to the regressive form take inversion of S1L22 (take complement of S1L23) (cf. tables 1C and 1E).

If $q < 96$, then the operator is calculative, requiring, besides the value of a variable or constant extracted by the instruction, also the value of accu.

The variable J of the interpreter (table 4B) takes the values $2 \times (q \div 2) - 128$ (cf. S10L 23, 26, 19, 15).

operator:	r	q
↑	1	64
x	2	66
/	2	68
÷	2	70
+	3	72
-	3	74
or (cf. table 1C: extract complement)	3	96
=	4	76
≠	4	48
>	4	80
<	4	81
≥	4	82
≤	4	83
not (cf. table 1C: extract inversion)	5	97
and	6	84
or	7	86
implies	8	88
equivalent	9	90

Table 1B.

Opening-, separation- and closing symbols.

The delimiters occurring below in the first column are opening symbols, having the ranks 0 and 10 on their left- and right-hand sides respectively. The other delimiters listed below may have the significance of both separation- and closing symbols, having only the rank 10.

In each line below, the number on the right-hand side is the value of the delimiters occurring in the line.
It can be written as follows:
S + 32 x rank 10 $(0 \leq s < 32)$.

After reading an opening symbol s, input goes to the compound statement metioned below to the right of s, for performing the pre-action of s.
For progressive and regressive after-actions of opening symbols confer S1L6 and S1L21 respectively.

After reading a colon, input goes to compound statement S3c.

After reading any other delimiters listed below, input goes to S1L1 after assigning the value of s to variable f, as happens also in the case of an operator s.

Opening symbols:			Separation- and closing symbols:	value:
(	S5		)	320
[	S6		]	321
begin	S3		end	322
:=	S9		do	323
for	S7		step until while	324
go to	S8		. . . . .	325
if	S2		then else	326
procedure	S3a	*	. . . . .	327
switch	S3a	*	. . . . .	328
lsq	S5b	**	. . . . .	
			, (comma)	330
			: (colon)	331
			; (semi-colon)	332

* After reading procedure or switch, input goes to compound S3a, as happens also after reading any declarator (cf. table 2). The pre-action of procedure (switch) in its quality of opening symbol begins on S4L24 (S4L37) after reading the procedure (switch) identifier.

** No value. Del. lsq is read and skipped by input 1.

Table 1C.

Extractive instructions

These instructions have an operation part 2^{26} x q with $96 \leq q < 104$.
For them, in operation time, the switch on S10L19 succeeds, using the variable J = q - 128 (cf. table 1A).

To the right of each tabled name, there is mentioned the jump performed on S10L19 for the instruction concerned, and the value of q.

instruction	goes to	q
extract complement	S10L88	96
extract inversion	S10L87	97
extract normally	S10L81	98

q = 100 is reserved for transport of arrays (cf. S11L43). Then the switch on S10L19 goes to S10L126.

Table 1D.

Non-extractive instructions.

These instructions have an operation part 2^{26} x q with $104 \leq q < 128$.
For them, in operation time, the switch on S10L15 succeeds, using the variable J = q - 128 (cf. table 1A).

To the right of each tabled name, there is mentioned the jump performed on S10L15 for the instruction concerned, and the value of q.

instruction	goes to	q
adjust	S10L100	127
ar1	S10L 62	126
ar2	S10L 65	125
for	S10L108	124
for1	S10L106	123
jump	S10L 55	122
pass	S10L 58	121
pro(cedure) state(ment)	*	120
restore	S10L101	119
		118
store accu	S10L121	117
store address	S10L 70	116
store factor	S10L 91	115
store pre-value	S10L 96	114
store procedure	S10L120	113
switch	S10L128	112
test	S10L130	111
verify	S10L140	110
Verify	S10L142	109
VERIFY	S10L135	108

* a prostate instruction occurs only in connection with a formal parameter which represents a function or procedure having parameters. The interpreter does not proceed to the switch on S10L15, but goes to S10L39.

Special constants.

The identifiers listed below such as extract address are convenient names of constants depending on the hardware programmes of the interpreter and translator. Whenever such an identifier occurs in the translator text above, the mark + at the end of the line indicates that, in praxis, the identifier is of course to be replaced by the corresponding constant.

The following jump instructions which are written in machine code, may occur in translated texts = object programmes. When extracted in operation time, they are executed normally on S1OL3 as indicated below.

instruction	goes to
extract address	S1OL 72
extract procedure	S1OL131
for0	S1OL112
for2	S1OL105
for3	S1OL111
part(ial) res(ult)	S1OL 0
retain	S1OL 99
return	S1OL 46
take complement	S1OL 90
take inversion	S1OL 89
X	S11L 0
X1	S12L 0
Y	S11aL 1

JO = jump instr. referring to abs. address 0, and
EO = extraction instr. referring to the abs. address 0.

The following values are assigned to the address variables of table 4A (cf. compound entry).

PO = lowest address of the working space
QO = highest address of the working space
LO = beginning address of the list L

for example: LO = (PO + QO) ÷ 2

QOO = QO + 2 x h

in which h is the number of the standard functions and -procedures whose object programmes are included in the object programme of the interpreter. These functions are permanently

Table 1E.

Special constants.

declared; their identifiers occur as constants of the transl. on addresses Q0+2, Q0+4,....Q00.

Table 2.

Declarations.

Declaration pattern D = 2^{23} x . . . . . p

neutral (cf. S3L1 and S3bL4):	1 111111111
before input of formal param.list (cf. S5L2):	0 000011110
when value- or specif. list may occur (S8dL9):	1 000011111

Declarator or specificator = 2^{23} x . . . . p

real	0 111111101
own	0 110111101
boolean, integer	0 111111001
array	0 011111101
label, string, switch	0 001111111
procedure	0 000111111
value	0 000001111

After reading a declarator or specificator, input, having assigned the required value 2^{23} x p to variable f of table 4A, goes to S3aL1 where D is modified:
D : = logical product D or f
which is positive i.e. bit D_0 = 0.
When no identifiers are declared or specified, D is negative so that the test on S4L1 succeeds.

Internal equivalent:

In general, the internal equivalent e of a declared identifier i has the form:

e = y + R + D

in which y is an address, while D and R = 2^{13} x r are the values of the variables D and R of table 4A which were resident when i was being declared.

When i has the significance of a simple variable or array or type procedure, then the bit e_8 is 0, e_7 being = 1 for the type real, = 0 for the type integer or boolean.

When i is an own variable or - array, the bit e_3 is yet = 1, though the pattern bit D_3 is = 0.

When i is a formal parameter, then

e = y + R + 2^{23} x 0 111111000

which is not derived from D = 2^{23} x 0 000011110.
The bits e_7 = e_8 = 0 do not refer to a type. The bit e_9 = 0 is characteristic for i = formal parameter.

Table 2.

Declarations.

When i is a procedure then r = - 1 + rank of that procedure, and y is the address where the procedure has its object programme beginning.

When i is a switch, y is the address where the switch instruction occurs in the object programme of the switch.

When i is a label, then

$$e = y + R + 2^{23} \times \overline{0\ 001111111}$$

(cf. compound S3c). (In which y is the address where a restore instruction occurs in the object programme).

When i is either a simple variable or array or formal parameter, then y is the (relative or absolute) address of either the variable or the array pre-value or the key by-word of the parameter.

When i is a simple variable or formal parameter, instructions are formed as follows:

$$e + 2^{26} \times (q - 63),$$

the number q being taken from table 1 A or C or D.

The translator still accepts formal parameter lists having the following form:
i, real i, real i, integer i, real procedure i, i, value i, integer value i, ... in which a parameter is either individually specified or unspecified. The corresponding pattern values

$$D = 2^{23} \times \overline{0\ 0000v1tx0}$$

are listed in the procedure's object programme and can still be modified according to specifications occurring in the text after the formal parameter part.

Only type- and value specifications have internal effects. When the bit v is = 0, the parameter i concerned is value. When the bit x is 0, then the bit t indicates a type specified for i.

Table 2A.

Contra-declarations.

The internal equivalent

$$e = x + 2^{23} \times p$$

of a contra-identifier

$$i + 2^{32}$$

points to the word st[x] of the object programme which must be adjusted later.

There are the following possibilities:

$p = \overline{1\ 100000000}$

S2aL11: A constant i' has been read which, because of its position within an actual parameter which is a non-trivial expression, might be a reference to a label.
There happens:
st[x] := $2^{23} \times \overline{t1} + 2^{26} \times 98$; st[x+1] :=i'.
Thus, in operation time, i' is extracted by the extract normally instruction preceding i' in the object programme.
If i' is suitable to be a label, which implies that the representation bit t = 0, the contra-declaration of i = i' + $2^{24} \times 63$ is made according to x and p.
If i' really occurs as a label, the translator will proceed later to S3bL48 where the word st[x] is replaced by the Verify instruction of the label.

In operation time, the Verify instruction is interpreted on S1OL142 according to the character of the instruction whose formal parameter represents the actual parameter containing i'.

$p = \overline{1\ 000000000}$

S5aL10: When a constant actual parameter with integer representation has been read, then
st[x-1] :=i'; st[x] := x - 1 + $2^{23} \times \overline{0\ 111111001}$,
the key main word being similar to the internal equivalent of an integer variable located on abs. address x-1.
If i' is suitable to be a label, the contra-declaration of i = i' + $2^{24} \times 63$ is made according to x and p (cf. S5aL 18-21).

If i' really occurs as a label, the translator will proceed later to S3bL48 where the word st[x] is replaced by the key main word of a constant actual parameter which occurs also as a label (cf. table 3).

In operation time, on S1OL82 the interpreter discovers, by examining the instruction whose formal parameter represents the

Table 2A.

Contra-declarations.

actual parameter concerned, if the actual parameter is a label or not, and changes the key main word accordingly.

p = 0 111111111

S2aL6: An identifier has been read which, because of its position within an actual parameter which is a non-trivial expression, may be a reference to a label. i has been looked up in the list I, and there happens: st[x] := either the verify instruction of i being a formal parameter, or the extract normally instruction of i being a simple variable, or 0, i being of another kind.

The contra-declaration of i is made according to x and p. Later the contra-declaration of i is found in the list I, and the declaration of i is looked up again. The translator proceeds to S3bL43.

If i is again found to be a formal parameter or simple variable, then the word st[x] need not be changed.
If i is found to be a label, the appropriate jump instruction is inserted on address x.
If i is found to be a function or procedure (which, in this case, may not have formal parameters), then the appropriate code instruction is inserted.

In operation time, verify instructions are interpreted on S10L140 depending on the character of the instruction whose formal parameter represents the actual parameter which contains the identifier i.

p = 0 011111111

S2aL19, S5aL27; i[...] has been read which, because of its position within an actual parameter, may be a switch designator.

If i is a formal parameter, then st[x] := the VERIFY instruction of i. Otherwise st[x] is already either the ar2 instruction of i being an array identifier, or 0, i being of another kind (cf. S6aL61). The contra-declaration of i is made according to x and p. Later the contra-declaration of i is found in the list I and the declaration of i is looked up again. The translator proceeds to S3bL42. If i is again found to be a formal parameter or array identifier, the word st[x] need not be changed.

Table 2A.

Contra-declarations.

If i is found to be a switch identifier, the appropriate jump instruction is inserted on address x. In operation time, VERIFY instructions are interpreted similarly on s1OL135 as are verify instructions on s1OL140.

$p = \overline{0\ 001111111}$

S8aL2, S8bL2, S2aL17

There has been read i or i [...] in a position in which i (which may also be a formal parameter) can only be a reference to a label or switch. Procedure SOL makes the contra-declaration of i according to p and pointing to the address x where the required jump instruction must be inserted later (cf. S3bL45)

$p = \overline{0\ 000111111}$

S5L6; There has been read i , which is the beginning of a function designator or procedure statement having actual parameters. Thus i indicates a function or procedure having formal parameters and is eventually not yet declared. Of course i may be a formal parameter too. The contra-declaration of i is made according to x and p. Later the contra-declaration of i is found in the list I and the declaration of i is looked up again. The translator proceeds to S3bL40.

If i is found to be a function or procedure, the appropriate code instruction is inserted on address x. If i is a formal parameter, the prostateinstruction of i is inserted.

$p = \overline{0\ 000011111}$

S1L12, S1L13, S6aL52 and procedure SOk

The identifier i considered is found to be no formal parameter and no simple variable. Because of its position, i can then only be a function having no formal parameters. The contra-declaration is made for i according to p and the address x where the appropriate code instruction must be inserted later on S3bL40.

On S3bL1 the identifier i considered can, because of its position, only refer to a procedure having no parameters, but may be a formal parameter. The same kind of contra-declaration is made. When, on S3bL40, i is found to be a formal parameter, the extract normally instruction of i is inserted on address x, which might as well be any other extractive or calculative

Table 2A.

Contra-declarations.

instruction carrying the formal parameter i.

The case of a function designator implies a slight restriction in the use of identifiers (which can be removed only by general use of contra-declaration for all identifiers): Given: any formal parameter or simple variable i being non-local to a block B. When there is no reference in the text of B to i, yet a local function having no formal parameters may not have the same name i unless its declaration occurs earlier than any reference to the function.

P = 0 000000000

S5aL8 The identifier i considered is an actual parameter whose significance may, of course, be declared later in the text. The contra-declaration of i is made according to p and the address x reserved for the key main word of parameter i (cf. S5aL 18-21).
Later the contra-declaration is found in the list I and the declaration of i is looked up.
The translator proceeds to S3bL24 for inserting the parameter key on addresses x-1 and x according to table 3.

After translating a text, the contra-declarations listed in I must have been satisfied with the only exception of those containing negative internal equivalents.
The latter contra-declarations which refer to constants which might be labels, have no imperative character (cf. S3bL18).

In a text which is a compound statement instead of a block, labels may yet be used freely.

Table 3.

The keys of parameters.

Below the keys of actual and formal parameters are listed on the left-hand side and right-hand side respectively. Any key consists of a main word, preceded by a by-word. These words are programme constants or, when the parameter is formal, local internal variables of the procedure, to which the parameter belongs.

When a procedure (or function) is invoked, compound S11, the transporter transforms the keys of the presented actual parameters into those of the corresponding formal parameters, the keys of the latter containing only absolute addresses and stores the results.

parameter =

non-trivial expression:

Y + E0 (cf. table 1E)	Y + E0 (simply copied)
$2^{23} \times \overline{1\ \overline{111111001}}$	$2^{23} \times \overline{1\ 111111tx1} + Z$

Object programme of expression begins on address Y.

Each time an evaluation of the expression through the formal parameter is going to be performed, the interpreter at first restores the values st[Z] and st[Z+1] of the variables chain and p of table 4B which were resident while the object programme of the block to which the expression belongs was operating.

For t and x confer "procedure" below.

simple variable:

by-word not important	by-word not important
$2^{23} \times \overline{0\ 111111t01} + y + 2^{13} \times r$	$2^{23} \times \overline{0\ 111111tx1} + Y$

Y is the absolute address where the variable with relative address y and rank r is located.

The bit t indicates the type of the variable (table 2), t = 1 → real, 0 → integer or boolean.

Mostly the bit x is 0. However, when there has been specified a type for the formal parameter and, in addition, that type differs from the type t, the transporter sets x to 1. Then, after any extraction of the parameter, the type t is transferred on S10L18.

array:

B	Z
$2^{23} \times \overline{0\ 011111t01} + y + 2^{13} \times r$	$2^{23} \times \overline{0\ 011111tx1} + Y$

Table 3.

The keys of parameters

The pre-value of the array which is either a programme constant or an internal variable of the object programme, has the relative address y and rank r and is located on the absolute address Y.

If there are k dimensions, k > 1, the factor called h_{k-1} in the explanation is located on the absolute address Z = Y + B. The number H_k of the array elements is located on address Z+1. The translator calculates B as a difference of relative addresses.

For the bits and x confer "simple variable" above.

label and switch:

by-word not important	Z
$2^{23} \times \overline{0\ 001111111} + Y + 2^{13} \times r$	$2^{23} \times \overline{0\ 001111111} + Y$

Y is the address corresponding to the label, or is the address where the switch [illegible] is located in the object programme of the switch. r is the rank of the label or switch identifier. When any jump to the address Y is performed, the interpreter restores the values st[Z] and st[Z+1] of the variables chain and p of table 4B, which were resident while the block to which the label or switch is local was operating.

Constant occurring in the text
as both actual parameter and label:

B = constant actual parameter	Z
$2^{23} \times \overline{0\ 001011001} + Y + 2^{13} \times r$	$2^{23} \times \overline{0\ 001011001} + U$

Y, r, and Z have the significances mentioned above for labels. The by-word B is located on absolute address U.

In operation time, on S10L82, the actual parameter is found to be either the label B or the "simple variable" located on address U, and the parameter key is adjusted accordingly.

A constant actual parameter B which does not occur as a label, is treated as the "simple variable" with pre-given value B which is located on the address U reserved for the key by-word.

Consequence: the value of a constant actual parameter can be changed in operation time by assignments.

	procedure:
Y + 1	Y + 1
$2^{23} \times \overline{0\ 000111001} + 2^{13} \times r$	$2^{23} \times \overline{0\ 000111tx1} + Z$

Table 3.

The keys of parameters.

The procedure indicated by the parameter has its object programme beginning on address Y, and has the rank r+1. Before any call of the procedure through the parameter is performed, the interpreter at first restores the values st[Z] and st[Z+1] of the variables chain and p of table 4B, which were resident while the block to which the procedure is local was operating. The transporter obtains absolute address Z by aid of r.

When no type is specified for the formal parameter, the transporter sets the bits t and x to 0.

When a type is specified in the text, the transporter sets x to 1 and has t indicates the type specified.
When, in operation time, the type t should differ from the representation of the value obtained through evaluation of the function (or expression, see above), transfer is arranged on S10L51.

	string:
address B	address B
address M	address M

In the object programme, the string consists of the words st[i], $M \leq i < B$

actual parameter being an identifier i
which is a formal parameters:

by-word not important	st[Y]
2^{23} x 0 000011000 + y + 2^{13} x r	st[Y + 1]

The by-word of the formal parameter i involved is located on the absolute address Y obtained by aid of the relative address y and rank r of i. The transporter simply copies the key of parameter i, though a type specified for the formal parameter and differing from that of i, is observed.

The value of a formal parameter f which is called by value in the text, is calculated and stored in the working space of the procedure to be called by the translator.
When f is no array, its key takes the form:

st[Y] = value of f
st[Y+1]
= 2^{23} x $\overline{0\ 111111t01}$ + Y

If there has not been specified a type for parameter f, the bit t indicates the representation of the value as obtained by

Table 3.

The keys of parameters.

the transporter, which is then consequently considered to be the type of the value parameter.

Table 4A.

The variable of the translator.

In compound entry the address variables are initialized. For the constants L0, P0, Q0, Q00 (cf. table 1E).

D, the declaration pattern of table 2.

mark, the signal of table 4c.

P = P0 (1)

The object programme which is the result of translating a text is beginning on address P0.

P is the address where to store the next word.

q = Q0 (-1)

is the relative address reserved for the non-own simple variable which the translator will next introduce into the organization of the object programme. After translating a block or procedure, the value of q resident before the translation of the block or procedure, is restored.

R = 2^{13} x resident rank = rank

of the procedure whose translation is running.

R, initially being 0, is increased resp. decreased by 2^{13} on S4L26 and S8dL3.

S and S0

In the list L, occupying the addresses S0, S0+1, ... S, opening symbols and operators are listed together with additional information. Initially there happens: S0 := constant L0, S := L0 + 1. Procedures S0, S0c, S0e and S0m observe the inequalities P + 1 < S0 and S < T (for T see below). When, on S3bL7, S is found to be = S0 + 3, then translation is ready.

During translation, st[S-2] is the delimiter (either an operator or opening symbol (tables 1A and B)) listed last in L. It has been stored by procedure S0c.

st[S-1] is either 0, or a programme constant i, or the internal equivalent (table 2) of a simple variable or formal parameter i, or the contra-identifier 1 + 2^{32} of an identifier i which is no simple variable and no formal parameter. Mostly, i is the constant or identifier which is, in the text, subsequent to delimiter st[S-2]; when absent, then st[S-1] is = 0. When, on S4L23, the internal equivalent or contra-identifier of an identifier i is listed, i itself is stored on address S which is occasionally required for making a contra-declaration.

Table 4A.

The variable of the translator.

Sometimes the list L is extended by decreasing variable. So for assembling a list of constants to be transplanted later to the object programme.

S' = S accent

When st[S-2] is an operator which has just been translated, then, on S1L17, there happens:
S' := S := S - 2.
When the test on S1L19 succeeds, the translator proceeds to input.

When st[S-2] is an opening parenthesis or - bracket p which is after-acting with the corresponding closing symbol, then, on S5aL5 or S6aL60, there happens:
S' := S := 2 + a[illegible] where the delimiter precedent to p in L is listed in L.
Then the translator proceeds to input.

After reading a separation symbol witnin a list of identifiers to be declared, there happens on S3aL4: S' := S, and the translator proceeds to input.

When, in any other case of an after-acting opening symbol, the translator proceeds to input, there happens, on S6aL11:
S' := 8191.

During input, S' is not changed (with the only exception that procedure Sog must shift the list L, in which case the difference S' - S is not changed).

On S1L7, S5L4 and S6aL44, comparing S with S'shows if the object programme must be equipped with storing and subsequent extracting a partial result, or not.

On S1L6, S' = S indicates regressivity.

On S1L2, S' = S denotes that st[S-1] may not be examined.

T = Q0 (-2) is the address where the identifier i declared next must be stored in the list I of declarations. The internal equivalent (table 2) of i is listed on address T-1. After translating a block or procedure, the value of T president before the translation of the block resp. after the declaration of the procedure identifier, is restored.
On the addresses T1 + 2 (2) T2, procedure SOf looks up the next identifier being equal to the value of variable f below. Mostly T1 is = T and T2 = Q00.

Table 4A.

The varable of the translator.

u and v

cf. own arrays, S6aL29.

a, b, c, d, e, f, g

are used intensively.

Compound statement input assigns the value of the identifier, constant, or delimiter, just read, to variable f (which is not necessary in the case of a delimiter : :=
([begin for go to if lsq).

If f is a constant, then g := 2^{23} x 0 000000t01
in which the bit t indicates the representation of the constant f.

If f is an array identifier read within an expression or actual parameter, then g := ari instruction taken from the list I (cf. S4L22).

If f is an identifier being of another kind, then g := 0.

Procedure S0a shows the main use of variables a,...e during the translation of operators.

Table 4B.

The variables of the interpreter.

In compound ENTRY a number of variables are initialized.

In the object programme of any problem, the first word is the beginning address P1 of the working space P1 ... Q0 of the object programme (cf.S3bL8).

Chain has the form $s + 2^{13} \times r$
in which r is the rank of the operating procedure or function f, and s is the key address presented when f was invoked. When no procedure or function is operating, the value of chain is not important though, in theory, r has the value 0 in this case.

Chain2
retains the value of chain (cf. e2).

e, the ex[illegible]tion of the interpreter, has the form: current address + E0 (cf. table 1E) which is supposed to be negative, but is occasionally replaced by a code instruction which is positive and is carried out normally on S10L2.

e2
retains the valu[illegible] of e for return in the case that the subscript of a switch designator is outof capacity (cf. S10L56, S10L101 and S10L129).

exp, mant
When the contents accu of the phantom accumulator is an integer, then mant = accu, exp = 0.
When accu is real, then mant = mantissa, and exp = 1 + 2 x exponent of accu.

p, the current pre-value.
When no procedure is operating, then p = 0.
When a procedure (or function) is invoked, compound S11 or S12 assigns the value Q - Q0 - 5 to variable p, also changing the values of chain, P, and Q (see below), retaining the previous values of chain, p, and P on the addresses p + Q0 + i (i = 1, 2, 3).
When a simple variable with relative address y has the rank r of the operating procedure, f, contained in chain during the operation of f, then the variable is located on the absolute address p + y.

p2
[illegible]tains the value of p (cf.e2)

Table 4B.

The variables of the interpreter.

P = lower bound, and

Q = upper bound

divide the working space P1 ... Q0, at any moment of the operation time of the problem, into 3 ranges:

The arrays which are in use at the moment, form, together, the range P1 ... P-1.

The simple variables which are used at the moment form, together, the range P1 ... P-1.

The simple variables which are used at the moment form, together, the range Q + 1 ... Q0.

The locations of the range P ... Q are not occupied at the moment.

Q2

retains the value of Q (cf. 62).

E, I, J, N, p1, s1, y

are intensively used in various significances, for example:

I ~~instr... extracted through~~

N = word extracted through e + 1,

$J = i \times 2^{-26}$.

to be used on S10L 15, 19, 23, and 26.

s1 = required (current or previous) value of chain,

p1 = required value of p connected with S1 (cf. S10L 9 -11).

y = address part of I, a relative address· then

y = corresponding absolute address.

N = value of required variable or constant, (cf. S10L16), which is broken up into N and E.

J = key main word, and E = key by-word of the formal parameter, to which instruction, to which instruction I refers.

Table 4C.

Mark.

There are 2 possibilities:

1 mark = 2:

Then has been read a switch designator i[...]. The subscript has already been translated. A special contra-declaration of i = st[S] is made according to the value p = $\overline{0\ 001111111}$ listed in table 2A.

2 mark = 0

There has been read a conditional designational expression. The identifiers contained in it which refer to labels and switches, have already been contra-declared, as will be shown below.

When the symbol (of an actual parameter part) is after-acting regressively, mark is tested on S5aL24.

There are again 2 possibilities:

3 mark = 1

There has been read a subscripted variable or switch designator i[...]. As the kind of i is to be regarded as not yet known, now a [illegible] to the constant p = $\overline{0\ 011111111}$ of table 2A.

4 mark = 0

There has been read an actual parameter, being a non-trivi[illegible] expression differing from i[...].

Only the instruction return of table 1E is added to [illegible] programme.

When the opening symbol if of an expression [illegible] else E2 is regressively after-acting with else o[illegible] closing symbol, there are 3 possibilities for t[illegible] aL14:

5 mark = 2

Then E1 or E2 is a switch designator and is tre[illegible] indicated in the case 1.

6 mark = 1

Then E1 or E2 is either a subscripted variable or a switch designator and is treated as indicated in the case 3.

7 mark = 0

Then the object programme of E1 or E2 has already been translated with involved contra-declarations.

For progressive after-actions of if cf. S2aL1. Then, in [illegible] discussion, labels take the place of switch identifiers.

Table 4C.

Mark.

During progressive after-actions of (, go to or switch, the signal mark need not be consulted.

Table 5.

Labels of the translator corresponding to mistakes.

S0gL Translator is short of working space.
S0LL A label is referred to by an unsuitable constant.
S1L2a Operator not is preceded by a constant or identifier.
S1L3a To the right of an arithmetic operator which do[illegible] not precede an opening parenthesis, a constant or [illegible]ntifier has been omitted.
S1L3b Identifier or constant has been omitted.
S3aL2a Declarator occurs in the wrong place.
S3bL2a In the compound tail of a block, or compound statement, a statement is followed by comma.
S3b[illegible]5a A declaration or statement , contained in a compound statement or block, is followed by a delimiter differing from semi-colon and end.
S3b18a Object programme is ready, wrong bei[illegible]
S3bL18a One of the identifiers has not been de[illegible]
S3bL33a An identifier has been defined twice within the same [illegible]side the bloc[illegible] contained in it.
S3bL41a Nonsense, invoked by a function des[illegible]cor or proce[illegible] statement.
S3bL41b A function designator or procedure statement having actual parameters, invokes a function having no formal parameters.
S3bL42a A subscript, attached to a simple var[illegible]
S3bL42b A subscript, attached to a procedure [illegible]
S3bL43a An array identifier to which no subsc[illegible] occurs within an expression.
S3bL44a A function designator or procedure s[illegible] no actual parameters, invokes a funct[illegible] having formal parameters.
S3bL45a Nonsense, referred to in a designati[illegible]xpression.
S3cL3a A label has been omitted.
S3cL7a A label, represented by an unsuitable constant.
S4L33 One of the specified identifiers does not occur within the formal parameter list concerned.
S4aL1a In a list of identifiers to be declared there occurs a constant.
S5L1a (Occurs within an identifier list of a block hea[illegible]

Table 5A.

Labels of the interpreter, corresponding to mistakes.

S10L0 No place available for storing a partial result, obtained while evaluating an expression or execu an assignment statement.

S10L35 A formal parameter with the apparent significa of a label or switch is found to represent a sir variable or a value parameter or a constant ich is no label.

S10L38 F formal parameter representing a string is sed ir an ALGOL-coded act.

S10L40 A designator f(...), in which f is a formal aramet has no space enough for calling the procedure represented by parameter f.

S10L42 No space available for obtaining he value of a parameter f which represents eith ocedure ha no formal parameters or an expressi

S10L50 A formal parameter which, in the contex , s significance of either an array or a label or to or a left part element of an ass ent statm is no function name, is found to ither a fi having no formal parameter, or an ss on di from a variable.

S10L52 A formal parameter representing used in the text in an undesigna

S10L64 A formal parameter representing that significance in the context.

S10L92 Within a bound pair, the upper boun value than the lower bound.

S10L98 No place can be reserved for the r arrays of the block which must be e

S10L100 No place can be reserved for the l well as internal) simple variable must be executed.

S10L107 There is no space enough to execute the next for statement.

S10L126a Exponent of real value to be stored is too great positive.

S10L126b Exponent of real value to be stored is too great negative.

Table 5A.

s of the interpreter, corresponding to mistakes.

144 Dummy elements in switch list: this case will never occur.

S1 145 A formal parameter f, contained in a designator f(...) is found to represent anything else than a procedure having formal parameters.

S10L1 Incorrect use of real variables.

S11L0 A designator f(...), in which f is a procedure name thus no formal parameter, has no space enough for calling the procedure.

1L1 No place available for storing the formal parameter keys of the procedure which must be invoked.

1L5 Actual parameter list of calling designator contains les[illegible] than does formal parameter list of [illegible] to be invoked.

S1 [illegible] parameter list of calling designator contains more elements than does formal parameter list of procedu[illegible] invoked.

0 A type [illegible] been specified for a formal parameter repr[illegible]ng a string.

[illegible]ty[illegible]l, integer or boolean) has been specified [illegible]l parameter representing a label or switch.

[illegible]ailable for storing the next value array [illegible]edure which must be invoked.

[illegible]or f having no actual parameters has no space [illegible]o call the procedure concerned.