STAF COMPUTATION DEPT

PROLEGOMENA TO X-8 ALGOL

(MCR 989)

STICHTING MATHEMATISCH CENTRUM.

1964

T E N G E L E I D E

Dit is een klad-afdruk van een deel van het materiaal voor het rapport R 989, "Prolegomena to X8 ALGOL", dat D.V. najaar '64 als Mathematical Centre Tract zal verschijnen.

Het is tevens een expreriment in "Engels uitlijnen" met extra layout-faciliteiten. Brandt Corstius komt dank toe voor de terzake verleende medewerking.

Dit is geen publicatie. Verspreiding is beperkt tot de Rekenafdeling van het MC en enkele intimi. De bedoeling is critiek, op- en aanmerkingen in te wachten en in een of meer werkbesprekingen te behandelen. Verantwoording naar buiten kan het MC niet dragen zolang de discussie over de hardware niet is afgesloten en de fabrikant in gebreke blijft behoorlijke beschrijvingen van zijn product te distribueren.

De inhoud is een collectief geestesproduct van de Reken-afdeling. De tekst is verzorgd door Kruseman Aretz en Nederkoorn. Aan het Engels is geschaafd door Mailloux.

13-5-64

J.N.

1. Preface

This report contains a preliminary sketch of an ALGOL 60 compiling system for the ELECTROLOGICA X8, a new computer to be released in 1965.

We only describe the running system: the output of the compiler is defined for the constructions allowed in the language. We also describe the administrative and executive subroutines (in our terminology, the "c o m p l e x") that are necessary for the proper functioning of the system. Some of them have now been programmed in full, others will be defined in terms of ALGOL 60 statements or in plain English. Little attention is given to the construction of the compiler itself.

In executing an object program no interpretive system will be used, nor will the program consist mainly of a sequence of subroutine calls. A compiler will produce a machine code program, which will occasionally call for the complex routines.

The X8 will be a digital electronic computer having 16K or 32K of directly accessible core storage. Most installations will be equipped with some kind of backing store, but this aspect will be deliberately neglected here. We only consider situations, wherein the complete object program, selected library programs, complex, working spaces and in- and output buffers are contained in directly accessible core memory.

The X8 may be concisely but incompletely described as being the same computer as the X1 [2], extended in the following ways:

1. a 54 bit register for floating numbers in a modified Grau representation [3] and hardware floating addition, subtraction, multiplication and division;

2. a dynamic addressing facility using up to 58 pseudo index registers. Any sequence of at most 58 consecutive memory locations may at any moment be selected as address modification registers for indirect referencing. Such a set of locations will be called a display. More than one display may exist at a time but only one will be active. A display is selected as active by storing the address of its first location in the display pointer D, which is address 63 of the core memory;

3. stack facilities including automatic stack pointer updating;

4. an instruction DO(n), which will cause the order in location n to be executed.

A software committee for the X8 has been organized, in the spirit of SHARE, under direction of Prof. Dr. Ir. A. van Wijngaarden. Other groups represented on this committee will do the necessary work with respect to assemblers, trace routines, in- and output facilities etc. A group in Eindhoven (Prof. Dr. E.W. Dijkstra and collaborators) will give attention to the multiprogramming problem, restricted however to ALGOL 60 programs.

The object of this report, describing the ideas of a working team of the Mathematical Centre on an ALGOL system for a homogeneous memory, is twofold:

1. It is a proposal to the software committee mentioned above and an offer to complete the system described; that is, to write all programs needed, the compiler, library organisation, in- and output routines, etc.

2. It tries to give potential X8 users at an early stage an insight into the possibilities and problems of an ALGOL system and to elicit discussions and criticisms.

Our team consists of:

1. Prof. Dr. Ir. A. van Wijngaarden
2. Drs. F.J.M. Barning
3. Dr. F.E.J. Kruseman Aretz
4. Drs. P.J.J. van de Laarschot
5. J. Nederkoorn
6. Drs. J.A. Zonneveld

This report has been written by Kruseman Aretz and Nederkoorn, using the working sheets of the group.

2. Notation and terminology

Hardly any knowledge of the machine code of the X8 will be necessary to understand the rest of this report, since we will explain every instruction in ALGOL 60, supposing that the following declarations have been made.

integer A, S, B, T, D; comment the registers A and S are accumulators, used mainly for fixed point operations and address correction and B for addressing the top of the stack // T is the order counter // D is the display pointer, used in dynamic addressing;

Boolean C, LS, LP, OF, NINT; comment condition, last sign, last parity, overflow, non-integer;

integer array M[0: 32767]; comment this array will represent the core memory;

real F; comment the floating register;

integer procedure red(a); comment before using a p o s i t i v e memory word for indirect addressing, this will be reduced as follows;

red := a - a : 2 ↑ 18 × 2 ↑ 18;

integer procedure pha(r,q); comment pha means physical address // the compiler will associate with a variable a dynamic address, consisting of 2 integers r and q// $-256 \leq q \leq 255$ // $0 \leq r \leq 57$;

pha:= red (M[red(D)+r]) + q;

comment dynamic addressing amounts to this: the relative address q is increased by the contents of an index register selected from the display under control of r. If positive values are assigned to D and to the index registers, only the 18 least significant bits of these words will take part in the operation;

We expect a compiler to consist of two distinct parts: an analytical main program, to a great extent machine-independent, and a m a c r o processor. The analytical part produces a string of macros, the action of some of them being specified by a set of addresses, code words, constants etc., which we will call metaparameters, to avoid confusion with the parameters of the source language. The macro processor

transforms the macros and their metaparameters into machine code orders.

Macros will have names, e.g. RET (return), TAV (take arithmetical value), TA (take address) etc., and code numbers for machine representation.

From the term macro it should not be inferred that the machine code translation of a macro will require, as a rule, more than one order. Some macros will, in fact, correspond to part of a machine order. We will also define macros, which are not produced by the analytical part of the compiler, to be used purely for explanatory purposes. A macro, in short, is a logical unit.

If no confusion threatens we will use the macro name for the macro itself, for the piece of object code translating it and, sometimes, for the routine in the complex called for by this code.

In the following sections a set of ALGOL statements printed on one line will correspond to one hardware instruction. In Appendix 1 a (generally more concise) version of each piece of program in the assembler's code can be found.

Since T := T + 1 is part of the function of most orders, this statement will generally be omitted from the order description.

Jumps to subroutines in the complex will be described as "procedure (< name of macro or subroutine >)".

3. General principles

Our first aim is to implement high fidelity ALGOL 60 as defined in [1], except for cases of downright impossibility as e.g. the dummy switch jumps (4.3.5. of [1]). Some quantitative restrictions due to hardware limitations will have to be accepted. The only important ones are these:

1. No numbers larger in absolute value than 2 ↑ 26 - 1 may be assigned to integer variables and integer procedure identifiers (the word length of the X8 being 27 bits).

2. A block must not be textually embraced by more than 57 other blocks.

3. Of course an object program plus working spaces must not exceed the storage capacity.

No restrictions wil be introduced as to side effects, size of floating numbers, length of names and expressions, depth of recursion, number of parameters and dimensions, for list elements, switch elements etc. Integer labels, unspecified formals called by name, value designational expressions and own arrays with dynamic bounds will be included. (The own concept, as defined in [1], is not wholly unambiguous. We chose the so-called static interpretation, which means that own variables, declared in a procedure, whether recursive or not, will have at most one identity. Except for own arrays with dynamic bounds and for clash of names, this amounts to handling own variables as if declared in the outermost block).

On the other hand no innovations, extensions or deviations from ALGOL 60 will be implemented on purpose, with one exception; namely, string as a declarator and type. The wide range of possibilities opened by this single extension will be discussed at length in sections 5.7. and 15.

Since the system caters for value designational expressions, it will implicitly cater for designational variables and assignments to them. So a compiler might accept these features.

We also consider optimization to be largely a matter concerning the compiler alone. So in describing the object program for the constructions of ALGOL 60, we will as a rule restrict ourselves to the most general case, sometimes indicating briefly some possible opportunities for optimization.

If, for a particular implementation problem, more solutions present themselves, some may be preferable as to speed, whereas others may lead to compacter object programs. In most cases we will then give preference to speed. In other cases we will leave things open, since compilers might, before compiling, give users the right to choose between fast-and-long and slow-and-compact. The latter choice would then imply that even short fixed sequences of machine code would be relegated to the complex.

4. Monitors, in - and output

Monitoring systems will of course depend on hardware configurations. We will consider only the barest possible outfit here: a basic computer operated by means of a teleprinter, one or two core modules of 16K each, a punched tape reader (1000 heptads/sec), a tape punch (150 heptads/ sec).

In this situation we expect that multiprogramming will offer little benefit. So we will restrict ourselves to sequential execution of ALGOL programs. The compiler will normally be operated in load- and-go fashion. Object programs produced need not be relocatable, therefore. It seems attractive, however, to be able to use the compiler for the construction of library procedures. The compiling program should therefore, at the user's request, produce (and print or punch in a readable code) object programs that can be located freely by a library selecting program.

Some object programs, indeed, will need completion by library procedures. In scanning the library tape for this purpose, and completing the object program with selected items from this library, or during the execution of the program, the compiler may be destroyed.

On the other hand, if the compiler should happen to be left intact, compilation and execution of a program may begin immediately, even before the output of the foregoing program has been completed.

Thus, for a monitor the following functions emerge:

1. Instructing the operator which kind of tape must be laid into the tape reader: compiler tape; library tapes; next ALGOL source program tape; or input tape of object program under execution;

2. Housekeeping of in- and output buffers;

3. Producing a complete logbook reporting operator actions, program identifications, program errors, execution times, etc.

These considerations (and the general necessities of our implementation system) almost force the partitioning of the entire storage into the following main sections:

1. the part of fixed extension during run time, including monitor program, complex, object program, selected library routines, etc.;

2. stack (for most variables, arrays, link data, etc.);

3. no man's land (immediately at the disposal of the main program for extending the range of the stack, counter-stack or buffer);.

4. counter-stack (for own arrays and strings);

5. in- and output buffer;

6. compiler.

Since the counter-stack will have to be shifted aside if an in- or output congestion occurs, all references to its contents including its internal references will be relative to a variable, viz. location of counter-stack (lcs), common to monitor and complex.

5. Expressions

5.1. Arithmetic expressions.

Initially, we will consider only simple arithmetic expressions involving no if clauses. Of these we give a general treatment first, covering the most awkward cases. In section 5.2. we will show how much may be gained by optimization in this area.

In calculating arithmetic expressions the system will - except in the case of the integer division - completely ignore the difference between integers and reals. Among floating point representations of numbers the Grau representation has the extremely desirable property that the floating point version of an integer is identical to its fixed point notation. Hence, we need not be concerned about transfer functions (see 3.2.5 of [1]).

Example:

$$-a + b \times (c - d \uparrow e \;/\; f) \qquad (1)$$

All identifiers denote simple non-formal variables.

Like most translating systems, our treatment will be based on transposition of such formulae into Reversed Polish Form [8].

Formula (1) then reads like this:

$$a - b\ c\ d\ e\ f\ /\ \uparrow\ -\ \times\ + \qquad (2)$$

If a machine scans this from left to right, it will never need to store or postpone any operation.

A simple implementation rule would be the following:

If the machine finds a variable, it takes the value of the variable into the floating register F, if necessary storing the previous contents of F on the stack. If it finds a monadic operator (e.g. the first minus sign of (2)), it transforms F accordingly. If it finds a dyadic operator the corresponding operation is performed, using the top of the stack as first operand and F as second operand. In this case the stack pointer is decreased. The result is again left in F. Thus, all intermediate results are initially formed in F and, if an expression has been calculated as the first operand of a dyadic operator, then its value has to be

saved in the stack, in order to free F for the evaluation of the second operand.

An obvious advantage of (2) is that the order of the primaries has not been changed. This seems irrelevant in ALGOL 60, because in [1] there are no specific rules concerning the order in which the primaries have to be fetched. So - if read is an input function, made equal to the next number on a punched paper tape, or any other function causing a side effect on its following activation - the expression

read ↑ read

is undefined. But we think it highly desirable that such expressions involving side effects have a well-defined meaning and so we postulate for our implementation that the results be equivalent to those obtained by fetching the primaries in textual order.

To derive (2) from (1) the following general rules might be applied (these are given here as a definition of the Reversed Polish Form, not as a compiler algorithm):

a. specify the order of operations in the expression by complete bracketing;

b. transform recursively all expressions of the form

(< adding operator > < expression >)

involving a monadic operator, into

< expression > < adding operator >

and all expressions of the form

(< expression1 > < dyadic operator > < expression2 >)

into

< expression1 > < expression2 > <dyadic operator >.

We now give a possible translation of (2) in terms of macros and metaparameters. All macros have built-in stack pointer correction.

q---

P r o g r a m	c o m m e n t
TAV(a)	take arithmetic variable; F := a
NEG	F := -F
STACK	M[B] := head(F); M [B+1]:= tail(F); B :=B+2
	In future we will avoid the use of the head-
	and tail-functions and contract the first two
	statements of STACK into: M[B]:= F;
TAV(b)	
STACK	
TAV(c)	
STACK	
TAV(d)	
STACK	
TAV(e)	
TTP	to the power; forms d ↑ e in F
STACK	
TAV(f)	
DIV	F := d ↑ e / f
SUB	F := c - d ↑ e / f
MUL	F := b × (c - d ↑ e / f)
ADD	F := - a + b × (c - d ↑ e / f)

The upper part of the stack passes through the following stages:

< empty >
- a
- a, b
- a, b, c
- a, b, c, d,
- a, b, c, d ↑ e
- a, b, c
- a, b
- a

< empty >

We will now leave this example and discuss the translation of arithmetic macros into X8 machine code passing over the addressing problem.

For TAV (< operand >) one has

F :=< operand >; (3)

if the operand is a simple variable or a constant. Is it a constant < 32768 in absolute value, then the so-called absolute versions of the take-order may be used, i.e. the address part of the order contains not the address of the constant but its absolute value.

If a simple arithmetic expression is of the form

+ < term >

then the "+" may be ignored by the compiler. If the expression is of the form - < term >, then the operator will be translated by NEG, the machine code version of which is

NEG: F := -F; (4)

Dyadic macros for + and - are ADD and SUB. ADD is the easier case because of the commutativity of the addition.

ADD: F := F + M[B - 2]; B := B - 2; (5)

Subtraction being non-commutative, matters become slightly more difficult. An order F := M[B - 2] - F; B := B - 2 is missing. A completely general solution would be the translation of a - b by

F := a;
M[B] := F ; B :=B + 2; comment STACK;
F := b; comment or any set of orders leaving the second operand in F;
SUB: NEG;
ADD;

i.e. as if the source program had been a + (-b). It becomes clear here that optimization might do a good job in suppressing NEG operations, exploiting the negative fetch orders in the X8 code. For multiplication we have:

MUL: F := F × M[B - 2]; B := B - 2; (7)

Division causes complications, again because an inverse division order is missing from the code. But we can always transpose operands. So we define

UNSTACK: F := M[B - 2]; B := B - 2; (8)

DIV: UNSTACK;stock := F; (9)

F :=F / stock;

F :=F / stock;

For TTP and IDI (integer division) no suitable orders are available in the code. Therefore these macros are translated as jumps to subroutines in the complex.

Until now the treatment has been general; that is, the schemes will work for any operands, however complicated. E.g. an operand may be conditional expression or a function designator or a subscript expression. Such expressions are evaluated by a piece of object program that must fulfil two requirements:

1. It must deliver the value of the expression in F;

2. The stack pointer must have the same value before and after the operation.

Similar rules will hold for expressions of other types. The reader should keep this in mind whenever simple take orders are used in examples.

5.2. Optimization of arithmetic expressions.

Since the macros described thus far will make possible an adequate, if not optimal, translation of the class of arithmetic expressions under discussion, we might leave it as a matter of compiler strategy how far to exploit the many obvious shortcuts presented by the order code, If any effort is spent on optimization, it will be wise to have it concentrated on this subject of arithmetic expressions.

It is important to note in this respect, that optimization may be found at different levels in the compiling process. It may often very easily be introduced into the macro processor. Two logically independent macros, which have been transformed into machine code may, in some cases be welded together by combining their respective last and first orders into one machine order. The macro processor might be aware of a set of these and similar possibilities.

The analytical part of a compiler may also be concerned with the optimization problem. Then it may prove useful to introduce new macros solely for this purpose, e.g.

TNAV(a): F := - a; (10)

ADD1(a): F := F + a; (11)

MUL1(a): F := F × a; (12)

SUB1(a): F := F − a; (13)

DIV1(a): F := F / a; (14)

All but the first are one-parameter-variants of the non-parametric macros ADD, MUL , etc. Using these extra macros we will be able to improve the translation schemes presented in the following ways.

NEG's may, if the operands are simple non-formal variables or constants, be suppressed by using the negative fetch order

TNAV(a): F := − a

This may easily be extended to more complicated operands such as products and quotients, provided the first operand of the latter constructions is again a simple, non-formal variable, or a constant.

In the case of dyadic operators, for a sequence like

STACK;
TAV(b);
ADD;

one machine code order

ADD1(b);

may be substituted.

This substitution applies when the second operands is a simple non-formal variable or a constant. But if only the first operand satisfies these requirements, and if the operator is commutative, the compiler might venture a reordering of the formula, provided the second operand cannot produce side effects changing the value of the first operand. (For these and other purposes we define a s t r a i g h t f o r w a r d expression to be any expression not involving procedure identifiers or formals. The set of straightforward expressions is a subset - large enough, in this context - of the set of expressions having no side effects at all) .

As a result the initial translation, based on Reversed Polish Form, of the formula a + b /c, that is

TAV(a);
STACK;
TAV(b);
STACK;
TAV(c);
DIV;
ADD;

may, at one stage, contract into

TAV(a);

STACK;
TAV(b)
DIV1(c);
ADD;

and, if a, b and c are straightforward, even into

TAV(b);
DIV1(c);
ADD1(a);

Regarding subtraction, though it be a non-commutative operation yet the same method will apply here, since it may be decomposed into NEG and ADD.

So we can translate a – b × c, provided all variables be straightforward, as

TNAV(b);
MUL1(c);
ADD1(a);

In the case of division, a new non-parametric macro will prove useful:

DIV2: F := F/M[B–2]; B := B–2; (15)

This macro will enable us, if reordering of operands is admissible, to avoid the transposition orders involved in (9), translating e.g. (a+b)/(c+d) as

TAV(c);
ADD1(d);
STACK;
TAV(a);
ADD1(b);
DIV2;

Evidently, it is not necessary here, that the first operand be simple.

5.3. Boolean expressions

Here again we will deal only with simple Boolean expressions involving no if clauses.

The binary register C – the condition register – is chosen as the most appropriate place for storing intermediate results in evaluating Boolean expressions. To represent the value of a simple Boolean variable or to store an intermediate Boolean result on the stack, the sign-bit of an X8 machine word (in our notation, the sign of an integer), will be

used. Subscripted variables, however, will be stored bitwise. All these cases are listed here.

logical value	true	false
sign of integer	+	-
sign-bit of X8 word or bit in Boolean array word	0	1
contents of C-register	0	1

If the sign-bit of an integer represents a Boolean value, then the remaining 26 bits of the X8 word are irrelevant and, in fact, undefined.

Important machine features in this context are:

1) the sign-bit of M[62] is always equal to the C-register; so all condition-setting orders may change this sign-bit;

2) as a consequence of the ones-complement number representation of the X8 [7], there are two representations for zero: the integer +0 is a sequence of 27 zeroes; the integer -0 is a sequence of 27 ones. Similarly,the mantissa of the real number 0.0 will be a sequence of either zeroes or ones (see section 18).

3) most orders in the code have condition-setting variants; i.e. the orders may be extended by condition-setting elements. Two of these elements are defined as macros.

POS: C := F $\geq$ + 0;

ZERO: C := F = 0;

A negative pendant of POS is missing. However, one may circumvent this deficiency since there is a one-bit-register LS (last sign) that is equal to the sign-bit of the last result used in setting C and most orders have a condition-setting variant testing the equivalence of LS and the sign-bit of the order, result (see e.g. LST).

Our treatment of Boolean expressions will again be based - in principle - on the Reversed Polish Form. Thus, in our first general

approach to the calculation of the value of relations, we will assume the first arithmetic operand to be on top of the stack and the second operand in F. Similarly, logical operators will expect their Boolean operands to be on top of the stack and in C, respectively.

For transport of operands, whether named or nameless, we have the following macros:

```
TBV(a): S := a; C := S ≥ + 0; comment take Boolean
                value;                                          (16)
STAB:           S := M[62];                                     (17)
                M[B] := S; B := B + 1; comment stack Boolean
                value;
UNSTAB:         S := M[B-1]; B := B-1; C := S ≥ + 0;             (18)
```

Macros for logical operators:

```
NON "¬"         S := -M[62]; C:= S ≥ + 0;                       (19)
AND "∧"         if ¬ C then B := B - 1;                         (20)
                if C then UNSTAB;
OR  "∨"         f C then B := B - 1;                            (21)
                if ¬C then UNSTAB;
IMP "⊃"         if C then B := B - 1;                           (22)
                if ¬C then begin S:=-M[B-1]; B:=B-1; C:=S≥ + 0 end;
QVL "≡"         S := M[62]; LS := S ≥ + 0;                      (23)
                S := M[B-1]; B:= B-1; C := LS≡ S ≥ + 0;
```

To simplify our description of relational operators, we draw attention to the fact that all corresponding macros will begin with a subtraction. So we define

```
MINUS:          F := F - M[B-2]; B := B - 2;                    (24)
```

Macros for relational operators:

```
EQU "="         MINUS; ZERO;                                    (25)
UQU "≠"         EQU;                                            (26)
                NON;
LES "<"         MINUS; POS;                                     (27)
                if C then begin F := F - 0; POS end;
```

```
MST "≤"     MINUS; POS;                                          (28)
            if ¬C then ZERO;
MOR ">"     MST;                                                 (29)
            NON;
LST "≥"     EQU; LS := F ≥ + 0;                                  (30)
            if ¬C then begin S := -1; C := LS = S ≥ + 0 end;
```

The statement F := F - 0 in LES is a consequence of the ambiguity of the zero representation. In the case -0<+0 the result of the preceding MINUS would be positive.

5.4. Optimization of Boolean expressions

The following remarks about possible methods of producing more efficient object programs are optional again. Compilers may or may not spend time or space to improve on the general approach set forth in the preceding section.

A) For systematic reflections on simplification of Boolean expressions along lines similar to those known from propositional calculus, we refer to [4].

B) In constructions like

BE1 ∧ BE2

BE1 ∨ BE2

BE1 ⌉ BE2

where BE stands for Boolean Expression, it will often be sufficient to calculate only one of the operands, provided the other is straightforward. If both operands are straightforward, the compiler may select the simplest one to calculate first. Hence, in some cases, the following substitutions may be applied before translation:

```
For BE1 ∧ BE2 read    if BE1 then BE2 else false
                or    if BE2 then BE1 else false
For BE1 ∨ BE2 read    if BE1 then true else BE2
                or    if BE2 then true else BE1
For BE1 ⌉ BE2 read    if BE1 then BE2 else true
                or    if BE2 then true else ¬ BE1
```

C) If the second operand of LES is a constant the second machine order of LES may be suppressed (provided the compiler substitutes -0 for +0).

D) If one of the operands of a relation is constant and =0, then a nonnegligible simplification of the corresponding object code may be achieved by applying the following rules during translation:

1) if the zero operand is the right hand one, bring it to the left, if necessary substituting the inverse relational operator;

2) suppress the TAV and STACK operations concerning the left operand;

3) suppress the MINUS element in the relational macro and, if possible, merge the condition-setting element of the first machine order of the macro into the last preceding order; i.e. into the order forming the right hand operand in F.

5.5. Conditional expressions

For translation of conditional expressions of any type, we introduce the macros JU and COJU (jump and conditional jump).

JU(A): go to A; (31)

COJU(A) if ¬C then go to A; (32)

A conditional expression of the form

if BE then E1 else E2

will be translated according to the following outline.

(code for BE);
COJU (L1);
(code for E1);
JU (L2);
L1: (code for E2);
L2:

In section 5.2 we mentioned the optimization technique of welding two consecutive macros by combining their last and first order into one machine order. A very simple example of this method presents itself here. In translating an if clause of the form

if ¬ BE then

as

(code for BE);
NON;
COJU(L1);

the macro processor may easily detect the possibility of combining the last 2 machine orders into one:

UNLESS(L1): if C then go to L1;

5.6. Designational expressions

Since there are no operators in ALGOL 60 working on designational expressions, the latter will be of one of the following kinds:

1) label identifiers;

2) unsigned integers;

3) switch designators;

4) complicated designational expressions, composed of others by means of parentheses, if clauses and delimiters else.

A full treatment of all problems arising here will be given in section 12. In this context the following remarks should suffice.

Not all designational expressions will deserve a translation as separate conceptual units. We expect e.g., that most go to statements will be translated as JU orders.

If a separate translation is necessary, the machine code representing the designational expression in the object program will have to perform a task analogous to that in the case of expressions of other types; that is, it will deliver the v a l u e of the designational expression (a notion to be explained in section 12) in the registers A and S.

In some cases, the translator may not have been able to decide whether an unsigned integer figures as an arithmetic or as a designational expression. In these situation, the expression is considered to be of mixed type and the corresponding code will deliver both values (arithmetical and designational) in the appropriate registers.

5.7. String expressions

As mentioned in section 3, the one extension to ALGOL 60 envisaged in our implementation will be the introduction of the delimiter string as a declarator and as a type. So we will have string variables and

assignments to them, string arrays, string procedures and conditional string expressions; in short, the full gamut of expressional facilities that are at our disposal with respect to the other types, except own strings.

At first sight, the absence of operators might seem a restriction, but this gap can and must be filled by procedures having a body expressed in non-ALGOL code. In our system, this will be X8 machine code. The choice of the particular operations to be implemented in this way will of course depend on the use one wants to make of strings and, therefore, on the semantics of those strings. So it seems convenient to expand somewhat on this subject.

From our point of view, a string is a quantity having no inherent meaning. Though there is a section in [1] about the semantics of strings (2.6.3.), this section is almost void of contents. Hence, the programmer, in collaboration with the designers of the implementation, has the freedom to define the meaning of the strings with which he is dealing. A string may, for example, mean a complex number, or a sentence of natural language, or a symbolic expression in the sense of LISP, or an algebraic formula, etc. Now the natural way to convey the meaning of a string in an ALGOL program is to choose the appropriate machine code procedure for creating or processing it.This choice of machine code procedure will also fix the internal representation of the string. For it is highly improbable, that of all possible meanings of strings we see fit to implement, one method of internal representation will be efficient or even feasible for all.

So, to handle complex numbers in a concise way, it would be attractive to add e.g. the following machine code procedures to the system, (either by inserting them into a program, or by adding them to the library, that is, more or less considering them as standard functions like those mentioned in 3.2.4. of [1]):

1) complexnumber (realpart, imaginary part); this creates a string, to be interpreted as a complex number, from two arithmetic constituents;

2) readcomplexnumber;

3) multcomplexnumber (a, b); here the two operands would be strings, interpreted as complex numbers;

4) ealpart (complexnumber); a function procedure of type real; etc. etc.

Other possible string procedures in machine code would be the famous LISP set, cons, car and cdr, together with Boolean procedures eq and atom, operating on strings having the same interpretation.

It goes without saying that the requirements for internal representation of the latter kind of string would be entirely different from those for complex number strings. These questions of representation will fully be dealt with in section 15. Here it will suffice to mention, that the treatment of string expressions in the running sytem will differ in one important respect from that of other expressions. Strings are unrestricted as to length and intricacy (except for the implicit restriction by finite storage size). So the value of a string expression cannot be left in a register. Instead, it will be left in the counter-stack. At the same time, the k e y of the string is formed in the A- and S-registers as a two-word unit. One of the functions of this key is to record the location of the string in the counter-stack.

In appendix 2 we give a complete list of the corrections and additions to [1], made necessary by our extension of ALGOL 60 with respect to strings.

6. The stack

Since a go to statement cannot lead into a block from outside (4. 3.4. of [1]), control will always enter a block at the first statment of that block, the block e n t r a n c e. We say that the block is then a c t i v a t e d. Until control has finished the execution of statements of the block, the block remains a c t i v e. A block will cease to be active:

1) by a n o r m a l b l o c k e x i t; that is, because the last statement of the text of the block (supposing that it is not a go to statement) has been executed.

2) by a l e a p i n g b l o c k e x i t; that is, by a go to statement leading towards a label outside the block.

In the text of an ALGOL 60 program, blocks of different levels may be distinguished. The level of a block is defined as being equal to one more than the level of the narrowest embracing block. The outermost block – the program itself – has level 0. It will sometimes be called the zero block.

In many cases activations and exits of the same block will alternate. But sometimes a block may be activated while it is still active, e.g. in the case of the recursive procedure call alluded to in 5.4.4. of [1] . In such situations, we say that several i n c a r n a t i o n s of the block are active, or that a number of n e s t e d activations of the block exists at one moment, whereas at most one activation may be u n d e r execution.

Since the other activations are yet unfinished and will possibly be completed later on, for all these activations the values of local variables, intermediate results of expressions and administrative data necessary for the proper completion of the activation, must be in store simultaneously. This rules out any statical addressing device as a general method.

On the other hand, if an incarnation has been finished or broken off by a leaping exit, the corresponding storage space must be left free

for other purposes (except for own variables, which are not discussed here. See section 14.).

Experience seems to indicate that the type of memory organization now commonly called a stack is the most suitable instrument for dealing with this dynamic storage allocation problem.

In our system a b l o c k c e l l is a set of consecutive memory words, reserved for the storage requirements of one block activation. At any moment during run time, the stack consists of an ordered set of adjoining block cells. The spatial order of the block cells in the memory (from the zero block upwards) reflects the temporal order of the unfinished block activations. The upper block cell belongs to the block under execution. Its length may vary during the execution of the various statements of the block, but it becomes fixed whenever a new block is activated, before it is finished. Then a new block cell is laid on the stack.

If a block exit occurs, one or more block cells are left free and the stack pointer (the B-register), indicating the address of the first free memory word "above" the stack, is correspondingly decreased. Thus, no shifting or sorting is necessary within the stack.

Roughly speaking, a block cell may contain:

1) link data (see section 9);

2) variables or sets of variables (arrays);

3) pseudo-variables, to be explained later, introduced by the compiling system;

4) displays, used for dynamic addressing (see below);

5) anonymous intermediate results of expressions being evaluated.

For every block cell, there is one principal point of reference, which for historical reasons is called pp (pronounced parameter pointer by Dijkstra [5] and procedure pointer by Randell and Russell [6]). This quantity pp coincides with the address of one of the lowest machine words of the block cell. Evidently, a pp is a dynamic quantity, evaluated anew for every activation of a block. In principle, therefore, all variables in the block cells have dynamic addresses; i.e. machine orders referring

to them are not equipped by the compiler with absolute machine addresses (p h y s i c a l a d d r e s s e s) but with an ordered pair of integers, r and q, denoting respectively the block level and the relative address in the block cell of the variable.

To explain this, we must recall that at any moment during run time only a subset of the variables, mentioned in the text of the source program, will be a c c e s s i b l e. This accessible subset comprises all variables declared either in the block under execution or in the blocks textually embracing it. However, not all values of these variables need be accessible; e.g. if a recursive procedure is active, the program has access only to the values belonging to the latest incarnation of the procedure.

So, if the block under execution has level n, then a d i s p l a y of n+1 pp's will suffice to define the absolute addresses of all relevant variables. As mentioned in sections 1. and 2. the order code of the X8 allows us to select any consecutive set of up to 58 memory locations as active display.

The reduction of the dynamic address of a variable v to its physical address, defined by

```
integer procedure pha(r,q); pha := M[D+r] + q;
```

or, more exactly (see section 2.)

```
integer procedure pha(r,q); pha := red (M[red(D) + r]) + q;
```

is then done automatically by hardware.

Thus, if in our previous descriptions of take orders, or any orders referring to variables, we wrote e.g.

```
F := a;
```

in a more precise rendering, this would run

```
F := M[pha(r(a), q(a))];
```

Now block activations and block exits will cause frequent changes in the set of accessible variables; therefore, if we had only one display, frequent updating would be necessary. To avoid this, several displays will, generally speaking, be kept in store in the stack. The union of all existing displays will cater for the potential accessibility of all active blocks.

Only one display is active at any one time, however, because D (= M[63]) stores the address of its first location. Adjusting D will be one of the tasks of the block activation and block exit routines in the complex.

7. Assignment statements

In this section we deal with all ALGOL 60 assignments, with the exception of assignments to simple formal variables not occurring in a value list.

Initially, our treatment of assignments will again be based on the Reversed Polish Form. To define this form for statements, we must here supply some priority rules for the assignment, considered as a dyadic operator:

1) the operator ":=" has a lower priority than any other operator occurring in an expression;

2) in multiple left part lists, any preceding ":=" has a lower priority than its successor.

In pseudo-ALGOL, bracketing would express the priorities as follows:

```
L1 := (L2 := (L3 := (E)));                    (33)
```

In Reversed Polish, this would run

```
L1  L2  L3  E  :=  :=  :=                     (34)
```

in accordance with 4.2.3. of [1].

At run time, every left part Li leaves on the stack an address description of a variable. Every assignment macro stores the value of the relevant register in the address described by the top o the stack and effects a suitable decrease of the stack pointer.

Evidently, in a case like (34), which involves a multiple left part list, all but the last assignment macro must leave the relevant register intact. For convenience, we extend this requirement to all assignment macros, with the exception of those affecting integer variables, which may round off the contents of F.

Note that in (34), as compared with (38), the order of the assignment operators has been reversed. This is important because macros used in translating these operators, though necessarily of the same "type" (4. 2.4. of [1]), need by no means be identical. They will reflect by themselves or by their metaparameters the properties (formal or not,

subscripted or not, etc.) of the Li preceding them in (33). In some cases, it may even appear effective to repeat part of the information, laid down in the object code of an Li, in the metaparameters of the assignment macro.

The necessity of evaluating the left part first and in textual order (otherwise the identity of left part variables would possibly be changed by side effects of expressions to the right, contrary to 4.2.3. of [1]), falls away if a left part variable is a simple non-formal variable. In these cases, the compiler will suppress the Li in the object program; their role is taken over by the assignment macro. Thus, in these simple cases, no address description is laid on the stack.

For subscripted variables, the address descriptions referred to are not physical addresses. Indeed it would be wrong to let the macro IND (indexer) derive the machine address in the array storage from the values of a set of subscript expressions, before the evaluation of E, because part of the arrays (including at least the own arrays) will be located in the counter stack, a set of machine words subject to shifting and shuffling. Some of these deplacements may be brought about during the calculation of E. Thus, at least for own array elements, the effects of IND will have to be postponed until the assignment macro gets its turn. Since this arrangement causes no loss of efficiency, it is adopted for all kinds of arrays.

Among factors differentiating assignment macros, type is preeminent, since it determines the storage requirements of a variable.

A list of these requirements is given here.

t y p e o f v a r i a b l e	n u m b e r o f X 8 w o r d s
real	2
integer	1
Boolean, non-subscripted	1
Boolean, subscripted	(1 bit, see 7.2.)
label (designational pseudo-variable)	2
key of a string	2

7.1. Assignments to simple, non-formal variables

7.1.1. Assignments to simple, non-formal reals

For the real case, we have a simple solution, involving only one machine order.

STR(v): v := F; (35)

This is an abbreviated description of the order. A more complete one would be:

M[pha(r(v),q(v))] := head(F); M[pha(r(v),q(v))+1] := tail(F);

7.1.2. Assignments to simple, non-formal integers

For integers, matters become more complicated, because there is no hardware rounding order. By our decision to represent integer values in the F-register during the evaluation of arithmetic expressions, we thus far successfully avoided transfer functions from real to integer representation and vice versa. But now, to get a two-word floating point number properly stored in a one-word integer location, the system must perform explicit normalizing and rounding operations. Consequently, we must eventually dwell at some length on the X8 floating point representation.

If i and exp denote integers and x a real, some triplets x, i and exp will satisfy the equations

$$x = i \times 2 \uparrow exp$$
$$|i| < 2 \uparrow 40$$
$$|exp| < 2 \uparrow 11$$
exp minimal
if exp = 0, then exp = + 0

Only numbers x belonging to such triplets can be represented exactly. If e is another integer and si and se are sign-bits of i and e respectively, e being defined by

e:= if si=0 then exp else - exp

then the corresponding hardware representation of x is shown in the following diagram:

	1 bit	11 bits	1 bit	14 bits
address(x) :	se	e	si	head of i
	1 bit		26 bits	

address(x)+1 : si tail of i

From a user's point of view, this diagram may be taken for a description of both the F-register and an arbitrary two-word unit storing a real number in the memory. Now, for all integers stored normally and exactly,

$$e = \pm 0$$

and

$$si = se$$

from which it follows, for all integers k, such that

$$|k| < 2 \uparrow 26,$$

that the contents of address(k) are either +0 or −0 and all significant bits of k can be found in the second word of the representation.

After these preparations, the following macro should be comprehensible.

```
STI(v):     S:= head(F); C := S=0 ;                          (36)
            if ¬ C then procedure (RND);
            v := tail(F);
```

In the procedure RND a special integer $3 \times 2 \uparrow 38$ (Scholten's constant) is employed. By adding and subtracting this constant to and from an arbitrary floating number x, both the normalizing and the rounding of x will be correctly performed. The order "procedure(RND) " is a subroutine jump laying its link on the stack.

```
RND:        F := F + scholtens constant;
            F := F − scholtens constant;
            S := head(F); C:= S=0;
            if C then return by top of stack;
            go to monitor;
```

In multiple assignments to integer variables, since the first assignment (i.e. the last, textually speaking) will have rounded the number in F, for all subsequent assignments a more simplified macro will suffice

```
SSTI(v):    v := tail(F);                                    (37)
```

The same macro may even be employed in translating a single (i.e. non-multiple) assignment or the first assignment of a multiple list, provided the right hand expression of the statement is a constant integer within the capacity of one X8 word, or a variable or a function-designator, declared to be integer.

7.1.3. Assignments to simple, non-formal Booleans

STB(v): S := M[62]; (38)
v := S;

For comments, if necessary, see 5.3.

7.1.4. Assignments to simple non-formal pseudovariables, corresponding to labels and to strings

A designational pseudo-variable being a two-word unit, we again distinguish a head and a tail of such a variable.

STL(v): head(v):= A; (39)
tail(v):= S;

To assign a string value to a string variable is a more complicated task, performed by a subroutine in the complex. So we translate

STS(v): tail(F):= pha(r(v),q(v));
procedure (STS);

For a full description, see section 16.

7.2. Assignments to subscripted variables

In this subsection, we will pay no attention to formal elements in subscripted variables, but we will consistently assume that the formal operations to be described in section 10 will have external effects analogous to those of the corresponding non-formal operations.

All macros storing values in locations of subscripted variables begin by having an indexing routine IND (or INDB for Booleans) compress the address descriptions on the stack (more fully defined in section 13) into a physical address, delivered in A, or, for Booleans, in A and S. Note that all macros set forth in this subsection are non-parametric.

7.2.1. Assignments to subscripted reals

Here again, the real case is the simplest one. In the object program we find:

STSR: procedure(IND); (41)
M[A]:= F;

7.2.2. Assignments to subscripted integers

In the complex we have the following routine.

```
STSI:           procedure(IND);                                    (42)
                S:= head(F); C := S=0;
                if C then begin M[A]:= tail(F);
                                  return by top of stack end;
                F := F + scholtens constant;
                F := F - scholtens constant;
                S := head(F); C := S=0;
                if C then begin M[A]:= tail(F);
                                  return by top of stack end;
                go to monitor;
```

In the simplified version, however (see 7.1.2), two orders in the object program will suffice.

```
SSTSI:          procedure(IND);                                    (43)
                M[A]:= tail(F);
```

7.2.3. Assignments to subscripted Booleans

The Boolean indexing routine INDB fulfils the following requirements:

1) it leaves the condition-register C unchanged; this may be realised in an easy way by the hardware feature of the "restoring jump" (see 2.7.2. of [2]).

2) the address of the X8 word containing the target bit (i.e. the bit, that represents the variable in question), is left in A;

3) it leaves in S a mask word, consisting of 26 ones and a zero, the position of the latter corresponding to that of the target bit.

In the object program, the macro STSB is translated as a jump to the subroutine

```
STSB:           procedure(INDB);                                   (44)
                stock := S;
                logical multiplication(S, M[A]);
```

```
                    if ¬ C then  logical addition (S,-stock);
                    M[A]:= S;
                    return by top of stack;
```

For an explanation of the terms logical addition (or "carry-less" addition) and logical multiplication see 2.3 of [2].

Of course, the policy followed here of allocating only one bit of storage space to subscripted Booleans, will entail some loss of efficiency in fetching and storing these Booleans, for the sake of greater compactness. Here, we may recall our remark of section 3. to the effect that other running systems, fashioned after the needs of the user preferring "fast Booleans", may easily be defined and introduced into the macro processor as an option. Such systems would differ from that described here in a small number of macros only (part of the Boolean fetch and store orders, Boolean array declaration), and would, in all other respects, be fully compatible with our overall organization.

7.2.4. Assignments to subscripted strings

After some preparations, the subroutine STSS will transfer to its non-subscripted analogue STS.

```
STSS:               stock := A;                                    (45)
                    stock1 := S;
                    procedure(IND);
                    tail(F):= A;
                    A := stock;
                    S := stock1;
                    go to STS;
```

7.3. Assignments to procedure identifiers

To the set of variables declared in the heading of the procedure body, the compiler will - in the case of type procedures - add a pseudo-variable, representing the procedure identifier and reflecting its type. Thus, assignments to such identifiers offer no difficulties any longer, since the same macros are used as in the case of ordinary simple variables. A suitable take order must be inserted into the object program,

at the end of the piece of code corresponding to the procedure body, by the compiler, thus ensuring that the value of the function-designator, after completion of the procedure activation, be delivered into the correct register.

7.4. Assignments to formal variables called by value

Here too, the compiler will introduce extra pseudo-variables, local to the procedure block, and representing the "value formals". The type of these can be derived from the specifications, which are obligatory in this case (see 5.4.5. of [1]). For every such formal, a sequence of macros performing the assignment of the actual value to the corresponding pseudo-variable, must be inserted into the object program. This is the only occasion, where - in legitimate ALGOL 60 - assignments to designational variables may occur.

No "fictitious block", as prescribed in 4.7.3.1. of [1], will be created. Hence, in order to avoid a conflict with 5.2.4.2. of [1], these value assignments must be executed prior to all array declarations. Further, we must allow the locals introduced in this fashion (but only these) to occur in lower and upper bound expressions of array declarations of the procedure block.

8. For statements and conditional statements

To explain our implementation of the for statement, we will first set forth a general approach matching the most complicated constructions. Then we will show the effects of optimization on simpler cases.

8.1. The general for statement

Consider a statement of the form

for V := E1, E2 while BE, E3 step E4 until E5 do S; (46)

V may be any variable, simple or subscripted, formal or non-formal. E1, E2,... are arithmetic expressions and BE is a Boolean expression. All expressions, including possibly subscript expressions in V, may be unrestrictedly complicated, involving mutual side effects, etc. They may contain function designators activating other for statements and even the for statement under discussion. S may be any statement. It may contain an arbitrary number of for statements, possibly nested. This high degree of freedom in the source program will require great caution in our design of the compilation method, particularly because in our way of dealing with the three kinds of for list elements we will consistently follow the models given in 4.6.4. of [1].

To facilitate compilation, the object code produced by the compiler will, on the whole, show a close correspondence to the textual order of the source program. Thus, though possible subscript expressions in V must be evaluated four times during every step of a step-until element (see 4.6.4.2. of [1]), the code for these subscript expressions will occur only once in the object program, preceding the code for E1 etc.

To achieve this, the compiler scans the source text and produces (at least in principle, see 8.2.) for every expression Ei, BE and for the left part V an implicit subroutine.

This is a piece of code, satisfying some special requirements. In section 10 we will meet implicit subroutines of a more complicated sort, used in translating actual parameters of procedure statements and function designators, but in this section the requirements are simple. An implicit

subroutine will be activated by a subroutine jump which lays its link on the stack. So it will end by a jump "return by top of stack". Furthermore it will deliver the value of its expression (E1 or BE) into the appropriate register (F or C).

As to V, it gives rise to a more elaborate construction. Its implicit subroutine will have two entrances. If it is activated by a jump order to entrance I, then it is expected to deliver the v a l u e of the left part variable into F. If, however, control comes in by entrance II, then it will produce an address description of V and lay it on the stack, preparing for a subsequent assignment.

After this, the compiler produces code for the statement S. Then it proceeds to compile short sequences of macros, each sequence corresponding to a for list element and functionally equivalent to one of the 3 patterns of 4.6.4. of [1].

To record its state of progress among the elements of the for list, each for statement needs a pseudo-variable s t a t u s. The compiler introduces these pseudo-variables as locals to the smallest embracing block in which the for statement occurs. If two for statements are textually nested (one being part of the other's dependent statement S), then they need 2 distinct status-variables. But if for statements are textually separate, they may share the same variable, provided they belong to the same block. Thus, the compiler can easily determine the number of status-variables needed for a block.

Essentially, the system proceeds from one for list element to another in two different ways. If control is dealing with an element of the first kind, an arithmetic expression, then status (being a non-orthodox designational variable) is given the value of an address in the object program, where either the following for list element, or the successor of the complete for statement begins (4.6.3. of [1]). On the other hand, when the two other kinds of for list elements come to be executed, status is set to an address i n s i d e the for list element and the transfer of control to the following element (or to the successor statement) depends on a test.

One may easily verify that the translation schemes presented in the following paradigm guarantee an adequate execution of the for statement

(46), independent of the textual order in which the 3 kinds of elements occur.

For assignments, the paradigm gives a general scheme. An assignment

```
V := Ei
```

is, in translation, described as

```
address V;
F := Ei;
store  ;
```

Each line may represent a subroutine jump here. It should be understood, however, that the compiler will insert the appropriate assignment version, selecting one of the possible arrangements given in section 7 (or, if V is formal, in section 10).

For the moment we refrain from writing syntactically correct ALGOL. We also depart from our convention that each line in a translation scheme represents one machine order.

```
                go to L2;                                          (47)
                (implicit subroutine for V)
                (implicit subroutine for E1)
                (implicit subroutine for E2)
                        etc.
            L1: (code for S)
                go to status;
            L2: address V;
                F := E1;
                store   ;
                status := L3;
                go to L1;
            L3: status := L4;
            L4: address V;
                F := E2;
                store;
                C := BE;
                if C then go to L1;
            L5: status := L6;
                address V;
                F := E3;
                go to L7;
            L6: address V;
                F := V;
                STACK;
                F := E4;
                ADD;
            L7: store;
                F := V;
                STACK;
                F := E5;
                SUB;
                STACK;
                F := E4;
```

```
F := -F; C := F=0; LS := F ≥ + 0;
if C then B := B-2;
if ¬ C then begin F:=M[B-2];C:=LS=F≥+0;B:=B-2 end;
if ¬ C then C := F=0;
if C then go to L1;
```

8.2. Optimization of for statements

Foremost among means of optimization here is the possibility of suppressing implicit subroutines whenever the expression or the variable involved is a simple variable or a constant, or even a simple variable, preceded by "+" or "-".

Then, to the addition and subtraction implicit in the step-until element, some of the remarks made in 5.2. apply.

Furthermore, if only one element occurs in a for list, the assignments to the status-variable become superfluous. In fact, the compiler may skip this sort of for statement in its count of needed status-variables.

In some cases, however (conceptually speaking very special cases, but probably covering the vast majority of actual for statements from a statistical point of view), even a more drastic simplification becomes practicable. In order that this may apply, the following conditions are sufficient:

1) V simple, non-formal;
2) the for list consists of one element, a step-until element;
3) E3, E4, and E5 simple, non-formal variables or constants.

In such situations a for statement may be translated as follows.

```
        F := E3;                                    (48)
        go to L1;
L0:     S;
        F := V;
```

```
        F := F+E4;
L1:     v := F;
        F := F - E5; C := F=0; LS := F ≥ + 0;
        if ¬C then begin F:=F-E4; C:= LS=F ≥+0 end;
        if ¬C then begin F:=E4    ; C:= F=0 end;
        if C then go to L0;
```

Here again, except for S, our convention that each line corresponds to one machine order holds.

Virtually the same set of orders may be used even if E3 or E5 are complicated or formal. If E4 happens to be a constant =/ 0, then even the penultimate order may be omitted.

8.3. Conditional statements

A conditional statement of the form

```
if BE then S1 else S2;
```

will be translated as

```
        (code for BE);
        COJU(L1);
        (code for S1);
        JU(L2);
L1:     (code for S2);
L2:
```

which, of course, closely resembles the outline given in 5.5 for the conditional expression. But here, since "else S2" may be missing, in some cases the code from the JU-order onward will be suppressed.

9. Non-procedure blocks

9.1. Entering a non-procedure block

At the beginning of the block activation, reservations have to be made in the block cell for the variables and pseudo-variables which are local to the block. These reservations are brought into effect by adjusting (and storing among link data of the block cell) a w o r k i n g s p a c e p o i n t e r, w p. Furthermore, to make these variables accessible to the statements in the block, the pp of the block must be recorded in the display.

The working spaces in the block cell, reserved for local variables, can be divided in two parts:

1) locations of simple variables and pseudo-variables; the total amount of this reservation can be established during compilation;

2) locations of arrays; in general, the storage space needed for local arrays must be calculated at run time, since it will depend on the values of the lower and upper bound expressions at the moment of block activation.

Hence, the adjustment of the wp is brought about in two steps.

First, under control of a metaparameter m, established by the compiler, the macro ENTER reserves the memory space needed for link data and local scalars.

At this moment, some test must be inserted to prevent the stack from growing into the counter-stack, or into buffer storage. For this purpose we introduce the macro TESTB, to be explained in section 14.

Later, if array declarations appear in the block head, the corresponding dynamic reservations are made by special type-bound declaratory macros, one for each set of lower and upper bound expressions (see section 13)

.

Each assignment to wp is combined with a corresponding change of

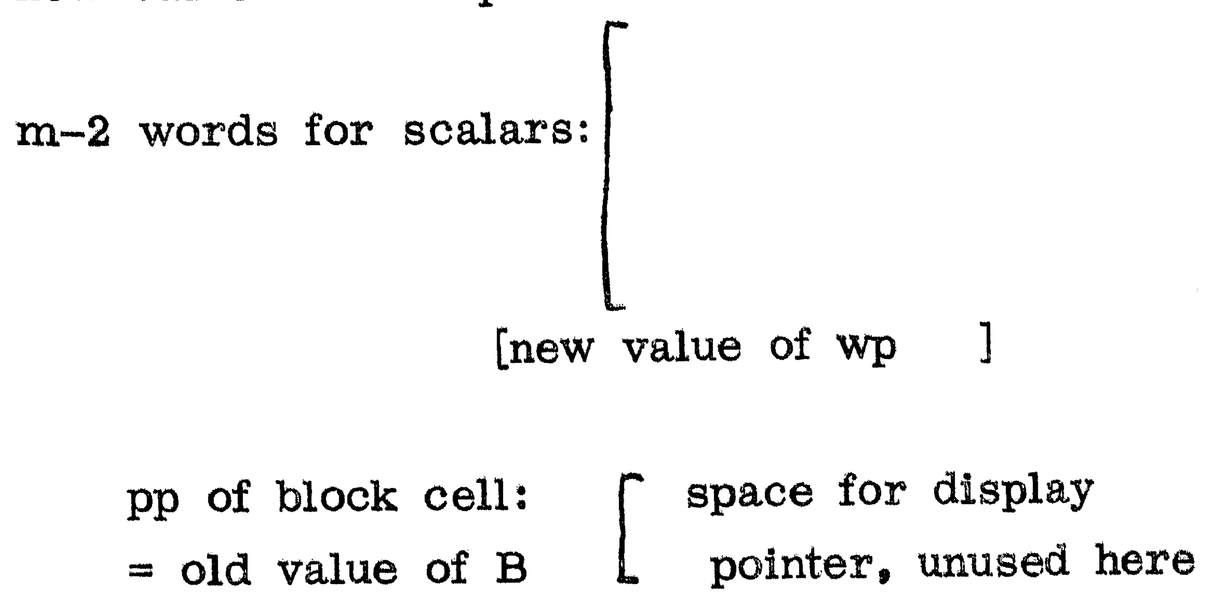
the stack pointer B. Thus, when declarations are completed and statements begin, wp is equal to B. This equality is, in general, lost during the execution of statements. After completion of each statement, however, B has automatically been reset to wp by implicit hardware stack pointer corrections. Therefore, keeping record of wp as a separate variable of the running system would be superfluous but for one difficulty: during the activity of the block under discussion (say, block A) , other blocks (say, blocks B, C, etc.) may be activated; in one of the latter blocks control may meet a leaping block exit to a label, local to A. In such cases, the block cells of B, C, etc. must be left free and the stack pointer must be reset to the value it had immediately after the declarations of A. A reconstruction of this value wp would be almost impossible, were it not that wp had been stored among the link data of A.

In the non-procedure blocks, considered in this section, the wp is the only link datum. The two other link data, the link proper, and the display pointer, only become relevant for the other kinds of blocks.

Thus, if m defines the storage requirements for link data and locals, regardless of arrays, and n is the block level, then, in the object program, the macro ENTER runs as follows:

```
ENTER(m, n):   M[D+n]:= B; comment  display [n]:= pp;          (49)
               B :=B+m   ; comment stack pointer adjusted;
               M[B-m+1]:=B; comment wp adjusted and stored;
               TESTB;
```

Part of the effects of this may be visualised in a diagram.



cell of enclosing block ⌈

In appendix 3 we give a complete list of all items taken into account in calculating m.

9.2. Exit of a non-procedure block;

We describe only normal exits here. For leaping block exits see section 12.

Let n again be the block level. Then, in the object program, the macro EXIT can be very simple.

EXIT (n): B := M[D + n]; (50)

Consequently, all reservations made by the corresponding

ENTER

macro are thereby cancelled.

For all programs not involving own arrays and variable strings (that is, all programs, the counter-stacks of which are empty), this will be sufficient. In the opposite case, it will be necessary to examine, whether part of the counter-stack may be left free. The technique of this inquiry and its possible effects are explained in section 14. In such programs, the compiler will conclude all non-procedure blocks by

EXITC(n): B := M[D+n]; (51)
procedure (FREE CHAIN);

19. References

1. Naur, P. (ed.).(1962). Revised Report on the Algorithmic Language ALGOL 60. Regnecentralen, Copenhagen.

2. Dijkstra, E.W. (1959). Communication with an automatic computer. Thesis, Amsterdam.

3. Grau, A.A. (1962). On a floating-point number representation for use with automatic languages. Communications of the ACM, 5, 3, pp. 160-161.

4. Wijngaarden, A. van, (1962). Switching and programming. Mathematical Centre, Amsterdam, report MR 50.

5. Dijkstra, E.W. (1960). Algol 60 Translation. Mathematical Centre, Amsterdam, report MR 35.

6. Randell, B. and L.J. Russell, (1964). ALGOL 60 Implementation. A.P.I.C. Studies in Data Processing No.5. Academic Press, London and New York.

7. Poel, W.L. van der, (1956). The logical principles of some simple computers, Thesis, Amsterdam.

8. Hamblin, C.L. (1962). Translation to and from Polish notation. Computer Journal, 5, 3, pp. 210-213.

Table of contents (preliminary)

Appendix 2

Amendments in [1]c, made necessary by the extension of ALGOL 60 with string as a declarator and type.

Section 2.3. DELIMITERS.

Replace the formulae for <declarator> and <specificator> by

<declarator>::= own |Boolean|integer|real|array|switch|
procedure|string

<specificator>::= label|value

Section 2.8. VALUES AND TYPES.

Replace the first paragraph by

"A value is an ordered set of numbers (special case: a single number),

an ordered set of logical values (special case: a single logical value),
an ordered set of strings (special case: a single string),or a label."
Replace the first sentence of the last paragraph by
"The various types (integer, real, Boolean, string) basically
denote properties of values."

Section 3. EXPRESSIONS.
Replace the formula for <expression> by
"<expression> ::=<arithmetic expression>|<Boolean expression>|
<designational expression>|<string expression>"

Section 3.2.3. Semantics.
Replace the first sentence by
"Function designators define single numerical or logical values
or strings, which result through the application of given sets of
rules defined by a procedure declaration (cf. section 5.4. PROCEDURE
DECLARATIONS) to fixed sets of actual paramaters."

After section 3.5.5. insert the new sections:
"3.6. STRING EXPRESSIONS.
3.6.1. Syntax.
<simple string expression>::=<string>|<variable>|<function designator>|
(<string expression>)
<string expression>::= <simple string expression>|
<if clause><simple string expression> else <string expression>"
Section 4.2. ASSIGNMENT STATEMENTS.
Replace the formula for <assignment statement> by
"<assignment statement> ::=<left part list> <arithmetic expression>|
<left part list><Boolean expression>|
<left part list><string expression>"
Section 4.2.4. Types.
Insert after the second sentence
"If this type is string, the expression must likewise be string."

Section 4.7.5.1.
Delete this section.

Section 4.7.5.4.

Delete the words "or a string" in the first sentence.

Section 5.1.1. Syntax.

Replace the formulae for <type> and <local or own type> by

"<type> ::= real|integer|Boolean|string

<local or own type> ::= <type>|own real |own integer|own Boolean"