ALGOL 60 Translation for Everybody

F.E.J. Kruseman Aretz

1964
ALGOL 60 Translation for Everybody

F. E. J. Kruseman Aretz 1), Amsterdam/Holland

Summary: It is the purpose of this article to sketch an ALGOL-Compiler structure that is expected to be comprehensible by everyone. So doing it has been abstracted as much as possible from any specific machine, and there are put forward only the essential features of translation. An example program for the translation of arithmetic expressions is given for illustration.


1. Introduction

Ever since the appearance of the first official publication on ALGOL 60 [1], there has been a worldwide effort directed toward the implementation of this problem-oriented programming language for several specific machines. Some of these investigations have been reported to a limited extent [2,3,4], or even in minute detail, while in some other cases only a few insiders have had the privilege of being initiated into the often masterly techniques used therein. Although there has been communication between the several designers about their work, and, consequently, common methods have been devised, nevertheless the various ALGOL 60 compilers diverge strongly, due, amongst other things, to:
1. differences in the machines;
2. differences with respect to the requirements and wishes posed for an ALGOL system by the various groups. To indicate the differences between machines, we mention here only such things as hardware stacking orders, indirect addressing, the size of the main store, and possible backing stores, whereas, on the other hand, the notions of efficient object program, of the detection of errors, and the size of translator and running system play an important role.

Hence, it is difficult for the uninitiated to gain an impression of how ALGOL translators actually do generate their object programs, and it is the purpose of this paper to sketch a translator structure that is, hopefully, comprehensible to everyone. So doing, we will abstract ourselves as much as possible from any specific machine, and will put forward only the essential features of translation. Therefore, no concession has been made to efficiency; everything has been sacrificed in the interests of readability. As an example, a program for the translation of arithmetic expressions has been worked out and is given here.

2. Macros

The simplest way to design a largely machine-independent ALGOL compiler seems to be to entrust the two logically distinguishable compiler tasks, namely:
1. analysis of the given program, and
2. construction of an equivalent machine code program, to two separate parts of the translator, namely:
1. the ALGOL-processor, and
2. the Macro-processor, respectively.

The first of these, then, will have to recognise the constructions used in the ALGOL source text, and represent
them in terms of a series of macro orders, possibly provided with one or more "metaparameters". These macros are independent, sharply described, small tasks, defined apart from any specific machine and capable of being suitably interpreted on any given machine.

The Macro-processor must generate a piece of object program for each such macro. This can be one hardware order, a series of orders, a subroutine call for a piece of running system, etc. Moreover, it is quite possible that a Macro-processor will replace a set of consecutive macros by another, more efficient one, and in this way will do quite a lot of optimizing. In any case, the Macro-processor will be strongly machine-dependent, but in principle can be a simple, straightforward program.

3. The language for our ALGOL compiler

Naturally, our machine-independent ALGOL translator will itself be formulated in a machine-independent programming language. As a side remark, we notice here that such a description would have some big advantages for any translator: 1. apart from the motivation, the translator would be its own description, and no documentation problems would arise afterwards; as soon as the translator has been finished, there exits a report, the translator text itself, which is accessible to anyone; 2. due to the greater readability, maintenance of the translator would be relatively simple, and, moreover, it would be fairly easy for customers to adapt the translator to specific requirements of their own.

If we then have to make a choice from the great variety of problem-oriented programming languages, we shall have to choose one that is really suitable for the problem. The most important property of ALGOL 60 that matters in this respect, seems to the author to be the essential recursivity in the definitions of the official report. Therefore, the translator must have a recursive structure, and our programming language will have to lend itself to recursive processes. Good examples of such languages are LISP [5] and ALGOL 60 itself. Although LISP, being a recursive list-processing language, seems to be highly suitable for the formulation of compilers, we have preferred ALGOL 60 for the purposes of this paper.

4. Some remarks on the storage-allocation problem

A consequence of the recursive structure of ALGOL 60 is that a block may sometimes be activated while it is already active. Then the values of the local variables of so-far incomplete activations must be preserved; thus, several values may correspond to each variable at any given time. As a rule this excludes a static form of addressing variables (something that could be done during translation). The commonly used way out of this so-called storage-allocation problem is the use of the memory organization now known as a stack [6], for a dynamic allocation of the variables. The stack then consists of a set of block cells, each belonging to just one activation of a block.

Each cell contains, besides the local variables and anonymous intermediate results of the evaluation of expressions, a set of administrative data, by means of which it is possible to find this and other block cells in the stack. Each variable can now be characterized by a dynamic address, consisting of a combination of two data: one for block identification and one for positioning inside the block cell concerned. It is a task of a block introduction macro to touch up the cell administration, and of the block exit macro to delete one or more cells out of the stack.

The top of the stack can be used for the evaluation of expressions. A so-called stackpointer refers to the first free position (this pointer might be an index register, but a pseudoregister can be used instead).

For the rest, all allocation problems are problems for the design of a running system. They have hardly any influence upon the writing of an ALGOL compiler.

5. The macros for the translation of arithmetic expressions

Since it goes far beyond the framework of this paper to sketch a complete ALGOL processor, we will restrict ourselves to a specific example. The simplest case is the translation of arithmetic expressions, and therefore in section 7, a number of procedures are given which are capable of converting an arithmetic expression into a series of macros. In this section, we will discuss what these macros are supposed to do, whereas in section 6 the basic functioning of the procedures will be elucidated.

We will assume that the running system has at its disposal two registers, called F and C. These may be real hardware registers, or programmed pseudoregisters. F is the register for floating point arithmetic, and for the sake of simplicity we will assume that F is also able to work with integers, which are distinguished from floating numbers only by means of an exponential part equal to zero (this is e.g. the case for the Grau representation [7] of floating numbers). C is a two-valued register, with value true or false. There exists a macro COJU, for conditional jump, which will continue the program from a different point only if C has the value false.

Furthermore, the running system will operate on the stack, and in principle, all binary arithmetic operations will take place with the top of the stack as the first operand and F as the second, the result being delivered in F. Examples are:

```
ADD addition with result F := stack[stackpointer - 1] + F;
SUB subtraction F := stack[stackpointer - 1] - F;
MUL multiplication F := stack[stackpointer - 1] × F;
DIV division F := stack[stackpointer - 1] / F;
IDI integer division F := stack[stackpointer - 1] ÷ F;
```

all with the side effect that:

```
stackpointer := stackpointer - 1;
```

The content of F can be saved in the stack by means of the macro:

```
STACK stack[stackpointer] := F;
```

Inversion of F is done by:

```
NEG F := - F;
```

To obtain the value of a simple variable, we introduce the macro:

```
TAV(<dynamic variable address>),
```

which will transform the given dynamic address (block identification number and position in the block cell) into a

234
memory address, and will copy the value of the variable into \( F \). Dealing with subscripted variables, the value of each index is evaluated and put into the stack, after which:

\[
\text{TSAV}(<\text{dynamic array address}>)
\]
carries out the indexing and delivers the required value in \( F \).

The treatment of formal, called by name, is done by the dynamic insertion of a piece of object program belonging to the corresponding actual parameter. This insertion is carried out by means of the macro:

\[
\text{DO}(<\text{dynamic parameter address}>)
\]
which causes execution of an order contained in the stack.

Finally we mention the macros:

\[
\text{JU}(<\text{program address}>): \text{goto \ address};
\]
\[
\text{COU}(<\text{program address}>) \quad \text{if } 1 \text{ \ then \ goto \ address};
\]
that play a role in the translation of \textit{if-then-else-constructions}.

6. The basic functioning of the procedures \text{Arithexp}, et seq.

In the next section a number of procedures will be declared, the effect of which is to translate arithmetic expressions. The fundamental idea of almost all of them is that the first basic symbol of the syntactical unit to be processed by that procedure has been read already (its value being assigned to "lastsymbol"); the procedure considers itself to have finished its task after reading the first symbol that can no longer belong to that unit syntactically. In the mean time, the translation of that unit has been produced.

For example we take here the procedure Factor. According to the definition in the Revised Report:

\[
<\text{factor}>::= <\text{primary}\\ | <\text{factor}\\ \times <\text{primary}>
\]

Therefore, a factor certainly must start with a primary, and so Factor begins with a call for Primary. On returning into Factor a set of macros has been produced, which, on execution, will deliver the value of the primary in \( F \). If now the primary is followed by a symbol \( \times \), then we have to save \( F \) in the stack, deliver the following primary in \( F \) and carry out the exponentiation. The procedure to produce the macros doing this is Next Primary. On completion of these three tasks a new test for the occurrence of a \( \times \) is necessary due to the recursivity in the definition of \text{factor}. This is carried out by Next Primary also, which, in case of absence of a \( \times \), has no effect at all.

7. The procedures

\[
\begin{align*}
\text{procedure Arithexp;} \\
\text{begin} & \quad \text{integer future1, future2;} \\
& \quad \text{future1=} \quad \text{future2=} \quad 0; \\
& \quad \text{if } \text{last symbol} = \text{if then begin} \quad \text{next symbol; Boolean;} \\
& \quad \quad \text{if last symbol } \times \text{ then then ERRORMESSAGE (AE1)}; \\
& \quad \quad \text{MACRO2 (COU, future1);} \\
& \quad \quad \text{next symbol; Simple Arithexp;} \\
& \quad \quad \text{if last symbol } \times \text{ else then ERRORMESSAGE (AE2)} \\
& \quad \quad \text{else begin} \quad \text{MACRO2 (JU, future2);} \\
& \quad \quad \quad \text{SUBSTITUTE (future2);} \\
& \quad \quad \quad \text{next symbol; Arithexp;} \\
& \quad \quad \quad \text{SUBSTITUTE (future2)} \\
& \quad \text{end} \\
& \quad \text{end Arithexp;} \\
\end{align*}
\]

\[
\text{else Simple Arithexp}
\]

If an arithmetic expression begins with an \textit{if}-clause, first a procedure \textit{Boolean} is called for, to produce a piece of program to assign the value of the Boolean expression to \( C \). Next, the macro \textit{COU} is produced. However, since it is still unknown to which point of the program the jump must go, a provisional address part of zero is given to it. It is a task of the Macro- processor, to assign in this case the value of the order counter to future1 so as to make possible a later address substitution. This last is done by the procedure \textit{SUBSTITUTE}, which must substitute the current value of the order counter into the address part of the macro \textit{COU}. An analogous construction occurs somewhat further on, just after encountering the \textit{else} (future2 then contains the location of the macro \textit{JU} to be completed later on).

The function procedure "next symbol" assigns both to itself and to last symbol, the value of the next basic symbol of the source program. The function procedure "Identifier" reads an identifier, looks it up in the name list, and assigns to itself a code word containing information about the identifier, e.g. its type, its address, etc., or it assigns to itself an address in the name list, from which such information can be obtained. In a manner analogous to this, the procedure "Unsigned number" has to read a number, and look it up in the constant list. Moreover, it has to produce a macro \textit{TCV}, take constant value, with the appropriate address. The procedures \textit{subcvra}, \textit{form}, \textit{function}, and \textit{arithmetic} must answer questions about the identifier concerned, on the basis of knowledge supplied by declaration or occurrence.

There are two points in which the definition of an arithmetic expression goes beyond its own borders: the first is the \textit{if}-clause, in which a Boolean expression occurs; the other is the function \textit{designator}, in which actual parameters of all kinds may occur. We shall not go into this, and the procedures \textit{Boolean} and \textit{Function Designator} will be present in the program only as black boxes.

Besides being translated, the ALGOL expression presented is also checked for syntactical correctness. In general, we have tried for a construction such that after detecting an error and producing a message about it, the translation process can be continued as a syntactical test with some chance of success (albeit that the translation then produced is worthless). A good example of this strategy is the reaction to a missing \textit{else}-part after an \textit{if-then-construction} (see Arithexp), and also to the absence of a closing parenthesis (see Primary).
procedure Simple Arithexp;
  begin if last symbol = minus then begin
      begin MACRO (NEG);
      next symbol; Term;
      end end
  else begin if last symbol = plus then next symbol;
      begin MACRO (ADD); Next Term
      end end
  Next Term
end Simple Arithexp;

procedure Next Term;
  begin if last symbol = plus then begin
      MACRO (STACK);
      next symbol; Term;
      MACRO (ADD); Next Term
      end else if last symbol = minus then begin
      MACRO (STACK);
      next symbol; Term;
      MACRO (SUB); Next Term
      end end
end Next Term;

procedure Term; begin Factor; Next Factor end Term;

procedure Next Factor;
  begin if last symbol = mul then begin
      MACRO (STACK);
      next symbol; Factor;
      MACRO (MUL); Next Factor
      end else if last symbol = div then begin
      MACRO (STACK);
      next symbol; Factor;
      MACRO (DIV); Next Factor
      end else if last symbol = idi then begin
      MACRO (STACK);
      next symbol; Factor;
      MACRO (IDI); Next Factor
      end end
end Next Factor;

procedure Factor; begin Primary; Next Primary end Factor;

procedure Next Primary;
  begin if last symbol = ttp then begin
      MACRO (STACK);
      next symbol; Primary;
      MACRO (TTP); Next Primary
      end end
end Next Primary;

procedure Primary;
  begin integer n;
    if last symbol = open then begin
      next symbol; Arithexp;
      if last symbol = close then next symbol
      else ERRORMESSAGE (P1)
    end else if digit last symbol then begin
      Unsigned number
      begin n:= identifier;
      if 1 arithmetic (n) then ERRORMESSAGE (P2);
      Arithname (n)
      end end
else ERRORMESSAGE (P3)
end Primary;
procedure Arithname (n); integer n;
begin
  if subcvar (n) \or\ form (n) \and\ last symbol = sub then Subcvar (n)
  else if function (n) \or\ form (n) \and\ last symbol = open then Function Designator (n)
  else if form (n) then MACRO2 (DO, n) else MACRO2 (TAV, n)
end Arithname;

procedure Subcvar (n); integer n;
begin
  if last symbol = sub then begin
    next symbol; Subscript list;
    if last symbol = bus then next symbol
  else ERROMESSAGE (SV1);
  if form (n) then MACRO2 (DO, n)
  else MACRO2 (TSAV, n)
end Subcvar;

procedure Subscript list;
begin
  Arithexp; MACRO (STACK);
  if last symbol = comma then begin
next symbol; Subscript list end
end Subscript list;

8. Final considerations

He who would analyse the procedures presented in the last section will find that arithmetic expressions are transformed into a macro program which corresponds to the so-called Reversed Polish Form [8]. This is carried out, however, without any need to give to the arithmetic operators and to the other delimiters any priority number, or to program a stack for operators, such as is described e.g. by Dijkstra [9]. The priority rules are taken into account automatically by the sub- and side-ordering of the appropriate procedures. Likewise, there is no need for a separate “future list” for the provision of the “anonymous jumps”, like those occurring in if-then-else-constructions. The only list that is assumed to exist, apparently, is a name list.

Furthermore, it will be clear how close the procedures are to the syntactical definitions, given in 3.3. of the Revised Report. As a matter of fact, they are just a straightforward transformation thereof, and this fact almost excludes errors during the writing of such procedures. No far-reaching conclusions have been drawn from the report; no attempt has been made to short circuit certain chains of thought; and in general such attempts may be the cause of errors.

Perhaps it is useful to say something about some difficulties which one encounters when trying to extend the process, followed in Arithexp, to other syntactical structures in the language. It is a property of arithmetic expressions that at any moment we know what type of structure has to be processed next during a sequential scan. This is no longer true for Boolean expressions, where e.g., after an opening parenthesis, either a Boolean expression or an arithmetic one (namely as part of a relation) can follow. Here we have to create a procedure that can read both types of expressions and that itself investigates what kind is presented to it. Moreover, such a procedure must notify the results of that investigation to its surroundings; even the answer; “yet undecided”, may not be excluded. Still more general procedures will be necessary for the translation of actual parameters - which can be anything -.

A case analogous to the foregoing arises in the assignment statement, where, after ‘<left part>=’, we don’t know whether an ‘<expression>‘ or again a ‘<left part>‘ follows. Just as before, we need procedures that can handle both situations. It is important here to have at our disposal a number of macros, suitable for both cases, such that the translation process can continue even before the discrimination between the two possibilities has been made.

It has been the purpose of this paper to give some insight into a plausible structure for an ALGOL 60 translator. In fact, the real problems arise during the design of the running system that has to see to it that the macros be defined work properly. The design of the translator is a relatively easy job.

This paper would not have been possible without continuous contacts with the collaborators of the Mathematical Centre, especially within the group preparing the ALGOL 60 implementation for our future Electrológica X 8 computer.

The author would like to mention here separately the cooperation with J. Nederkoorn in the definition of a macro system for that machine. Furthermore, the author thanks B. Mailhoux, who was helpful in the final formulation of the text.

In appendix A a program is given, with which the procedures given in section 7 have been tested using the MC-I ALGOL implementation on the X 1. In appendix B some examples of input and output of this program are reproduced.

9. References

These procedure identifiers that do occur in the program without having been declared are library routines, which, in the MC-1 ALGOL implementation, are considered as standard functions, just like enter, abs, sqrt, etc. They are:

RE7BIT:
a function procedure assigning to its identifier the value of the next heptad on the input paper tape;

PUTBIT (n):
a procedure, punching the value of n (0 ≤ n ≤ 127) as a heptad on the output paper tape;

PUNLCR:
a procedure, punching a new line carriage return symbol on the output paper tape;

PUSPACE (n):
a procedure, punching n space symbols on the output paper tape;

PUTEXT (string):
a procedure, punching the actual string on the output paper tape ( and are the MC hardware representations of string quotes);

ABSFIXP (n, 0, x):
a procedure, punching the absolute value of x rounded to an integer, in n figures, replacing leading zeroes by space symbols;

stop:
a procedure that stops the execution of the program until the operator pushes the button "continue" on the machine console.

The program translates a set of expressions on the input tape, separated by semicolons, and closed by a semicolon followed by a stopcode symbol (this last symbol corresponds to a punching but has no visible mark on the typewriter sheet). In Backus Normal Form we could define the job as:

A4: <job> := <arithmetic expression> ; stopcode <arithmetic expression> ; <job>

Any set of basic symbols comment (any sequence not containing ; ) is skipped on the input tape. The program starts with the reading of a small tape, containing some of the basic symbols in hardware representation, to define the internal representation thereof. This is just a preparation of the input procedures. The small tape consists of the symbols:

begin
  comment
  Test program for Arithexp,
  programmed by F. E. J. Kruseman Aretz,
  identification: R969 / Arithexp;

integer last symbol, pointer, macrocounter, stock, empty, symbol,
blank, erase, case, lower case, upper case,
tab, space, new line, stopcode,
colon, equal, bar, underlining, comment, semicolon, comma,
open, close, sub, bus, if, then, else,
plus, minus, mul, div, idi, ttp,
AE1, AE2, P1, P2, P3, SV1, SV2, FD1, MP1, STOP.
NEG, ADD, SUB, MUL, DIV, IDI, TTP, STACK, ENTER, RET,
JU, COJU, TAV, TCV, TBV, TSAVE, DO, FDES;

integer array list[0:200];
procedure Initialization;
begin
AE1:= NEG:= JU:= 1; AE2:= ADD:= COJU:= 2;
P1:= SUB:= TAV:= 3; P2:= MUL:= TCV:= 4;
P3:= DIV:= TBV:= 5; SV1:= IDI:= TSAV:= 6;
SV2:= TP:= DO:= 7; FD1:= STACK:= FDES:= 8;
MP1:= ENTER:= 9; STOP:= RET:= 10;
blank:= 0; erase:= 127; case:= lower case:= 122;
upper case:= 124; tab:= 62; space:= 16; new line:= 26;
stop code:= 11; stock:= empty:= 1;
colon:= next tape symbol; equal:= next tape symbol;
bar:= next tape symbol; underlining:= next tape symbol;
comment:= next string symbol; semicolon:= next string symbol;
comma:= next symbol; open:= next symbol; close:= next symbol;
sub:= next symbol; bus:= next symbol; if:= next symbol;
then:= next symbol; else:= next symbol; plus:= next symbol;
minus:= next symbol; mul:= next symbol; div:= next symbol;
id1:= next symbol; ttp:= next symbol;
end Initialization;

comment input procedures;

integer procedure next tape symbol;
begin
integer n;
  n:= RETBIT;
if n = blank V n = erase then next tape symbol:= next tape symbol
else if n = lower case V n = upper case then
  begin case:= n; next tape symbol:= next tape symbol end
else next tape symbol:= if n = tab V n = space V n = new line V
  n = stop code V case = lower case then n else n + 128
end next tape symbol;

integer procedure next string symbol;
begin
integer n, m;
if stock < 0 then n:= stock:= - stock else n:= next tape symbol;
if n = colon then begin m:= next tape symbol;
  if m = equal then n:= 256 else stock:= - m
end :=
else if n = bar then n:= next tape symbol + 256
else if n = underlining then begin n:= next tape symbol + 512;
  m:= next tape symbol;
  if m = underlining then
    begin n:= 512 X n + next tape symbol;
      AA:= stock:= - next tape symbol;
      if stock = - underlining then
        begin next tape symbol:= goto AA end
    end word delimiter
  else stock:= - m
end underlining;
next string symbol:= n
end next string symbol;

integer procedure next symbol;
begin
integer n;
  n:= next string symbol;
if n = tab V n = space V n = new line then next symbol:= next symbol
else if n = comment then begin for n:= next string symbol while n # semicolon do ;
  next symbol:= next symbol
end comment
else next symbol:= last symbol:= n
end next symbol;
Boolean procedure digit last symbol;
digit last symbol := (0 < last symbol \&\& last symbol < 9) \lor (18 < last symbol \&\& last symbol < 26) \lor last symbol = 32;

Boolean procedure letter last symbol;
begin integer n;
n := if last symbol < 128 then last symbol else last symbol - 128;
letter last symbol := (34 < n \&\& n < 42) \lor (48 < n \&\& n < 57) \lor (66 < n \&\& n < 74) \lor (80 < n \&\& n < 89) \lor (96 < n \&\& n < 105) \lor (114 < n \&\& n < 122)
end letter last symbol;

comment output procedures;

procedure ERRORMESSAGE (n); integer n;
begin switch s := AE1, AE2, P1, P2, P3, SV1, SV2, FD1, MP1, STOP;
PUNCTCR; PUTEKT1 ("ERROR"); goto S[n];
AE1: PUTEKT1 ("AE1"); goto EX; AE2: PUTEKT1 ("AE2"); goto EX;
P1: PUTEKT1 ("P1"); goto EX; P2: PUTEKT1 ("P2"); goto EX;
P3: PUTEKT1 ("P3"); goto EX;
SV1: PUTEKT1 ("SV1"); goto EX; SV2: PUTEKT1 ("SV2"); goto EX;
FD1: PUTEKT1 ("FD1"); goto EX; MP1: PUTEKT1 ("MP1"); goto EX;
STOP: PUTEKT1 ("STOP");
EX:
end ERRORMESSAGE;

procedure MACRO (i); integer n;
begin switch s := NEG, ADD, SUB, MUL, DIV, IDI, TTP, STACK, ENTER, RET;
PUNCTCR; ABSFIXP (3, 0, macrocounter); PUSPACE (3); goto s[i];
NEG: PUTEKT1 ("NEG"); goto EX; ADD: PUTEKT1 ("ADD"); goto EX;
SUB: PUTEKT1 ("SUB"); goto EX; MUL: PUTEKT1 ("MUL"); goto EX;
DIV: PUTEKT1 ("DIV"); goto EX; IDI: PUTEKT1 ("IDI"); goto EX;
TTP: PUTEKT1 ("TTP"); goto EX; STACK: PUTEKT1 ("STACK"); goto EX;
ENTER: PUTEKT1 ("ENTER"); goto EX; RET: PUTEKT1 ("RET");
EX: macrocounter := macrocounter + 1
end MACRO;

procedure MACRO2 (i, n); integer i, n;
begin integer k, last case;
switch s := JU, COJU, TAV, TCV, TBV, TSAV, DO, FDES;
PUNCTCR; ABSFIXP (3, 0, macrocounter); PUSPACE (3); goto s[i];
COJU: PUTEKT1 ("COJU");
JU: PUTEKT1 ("JU"); ABSFIXP (3, 0, n); n := macrocounter; goto EX;
TAV: PUTEKT1 ("TAV"); goto NAME;
TCV: PUTEKT1 ("TCV"); goto NAME;
TBV: PUTEKT1 ("TBV"); goto NAME;
TSAV: PUTEKT1 ("TSAV"); goto NAME;
DO: PUTEKT1 ("DO"); goto NAME;
FDES: PUTEKT1 ("FDES");
NAME: last case := 0; for k := n step 1 until pointer - 1 do
begin if list[k] > 127 then
begin if last case = 2 then begin last case := 2; PUTBIT (upper case) end;
PUTBIT (list[k] - 128)
end
else begin if last case = 1 then begin last case := 1; PUTBIT (lower case) end;
PUTBIT (list[k])
end
end
pointer := n;
EX: PUTEKT1 ("EX"); macrocounter := macrocounter + 1
end MACRO2;

procedure SUBSTITUTE (n); integer n;
begin PUTEKT1 ("SUBSTITUTE"); ABSFIXP (3, 0, macrocounter);
PUTEX1 ("in address part of macro"); ABSFIXP (3, 0, n)
end SUBSTITUTE;
comment supplementary translating procedures;

procedure Boolean; begin integer n; n:= identifier; MACRO2 (TBV, n) end Boolean;

procedure Unsigned number; begin integer n; n:= number; MACRO2 (TCV, n) end Unsigned number;

procedure Function Designator (n); integer n; begin integer future; future:= 0; MACRO2 (FDES, n); if last symbol = open then begin MACRO2 (JU, future); MACRO (ENTER); next symbol; Arithexp; MACRO (RET); SUBSTITUTE (future); if last symbol = close then next symbol else ERROREMSSAGE (FD1) end end Function Designator;

integer procedure number; begin number:= pointer; list[pointer]:= last symbol; pointer:= pointer + 1; next symbol; if digit last symbol then number end number;

integer procedure identifier; begin identifier:= pointer; list[pointer]:= last symbol; pointer:= pointer + 1; next symbol; if letter last symbol V digit last symbol then identifier end identifier;

comment informative procedures;

Boolean procedure arithmetic (n); integer n; arithmetic:= true;

Boolean procedure subscrvar (n); integer n; subscrvar:= last symbol = sub;

Boolean procedure function (n); integer n; function:= last symbol = open;

Boolean procedure formal (n); integer n; formal:= false;

procedure Arithexp; < body of Arithexp > ;

procedure Simple Arithexp; < body of Simple Arithexp > ;

procedure Next Term; < body of Next Term > ;

procedure Term; < body of Term > ;

procedure Next Factor; < body of Next Factor > ;

procedure Factor; < body of Factor > ;

procedure Next Primary; < body of Next Primary > ;

procedure Primary; < body of Primary > ;

procedure Arithname (n); integer n; < body of Arithname > ;

procedure Subscrvar (n); integer n; < body of Subscrvar > ;

procedure Subscript list; < body of Subscript list > ;
TAV (a)
STACK
FDES (sin)
JU ( 0 )
ENTER
TAV (omega)
STACK
TAV (t)
MUL
RET
MUL
TBV (first)
COJU ( 0 )
TCV (0)
JU ( 0 )
TBV (last)
COJU ( 0 )
TCV (0)
STACK
TSAV (A)
STACK
TSAV (A)
JU ( 0 )
TCV (0)
STACK
TSAV (A)
FDES (in)
JU ( 0 )
ENTER
FDES (abs)
JU ( 0 )
ENTER
TAV (n)
STACK
TAV (n)
STACK
TCV (1)
SUB
MUL
STACK
TCV (2)
DIV
RET
RET
STACK
TBV (pos)
COJU ( 0 )
FDES (sqrt)
JU ( 0 )
ENTER
TAV (x)
RET
JU ( 0 )
FDES (sqrt)
JU ( 0 )
Enter
TAV (x)
NEG
RET
ADD

substitute 10 in addresspart of macro 3
substitute 4 in addresspart of macro 1
substitute 12 in addresspart of macro 5
substitute 15 in addresspart of macro 11 substitute 15 in addresspart of macro 3
substitute 17 in addresspart of macro 4
substitute 18 in addresspart of macro 1
substitute 26 in addresspart of macro 22
substitute 27 in addresspart of macro 20
substitute 33 in addresspart of macro 28 substitute 33 in addresspart of macro 26
FORTRAN

Summary: The development of FORTRAN since its initial appearance is reviewed, especially its growth from an algorithmic to a programming language. Impending developments are indicated.


It is now 7 1/2 years since the first FORTRAN system was issued. The developments which have taken place in FORTRAN itself since that time are of course only a part of the development which has occurred in the field as a whole. Nevertheless they are of considerable interest, as I shall explain in a moment, and it is with them that this paper will be concerned. In these subsequent developments, incidentally, I have had no direct part, I write about them therefore as an interested, but disinterested, onlooker.

What gives these developments their particular interest is the relation of FORTRAN to practical computation. Every programming language and every language processor is inevitably a compromise among factors too numerous and in some cases too ill-defined to mention here. The considerations which should affect the compromise can never be fully known to us, and moreover they are continually subject to the changes that occur in applications, hardware, and computing practice generally. With FORTRAN, probably more than with the other programming languages because its use has been so great and because its users have been so effective in making their experience count in its evolution, the developments reflect in a most interesting way the actual and changing requirements of computation.

The effects of this relation to practice can be noticed in many areas. One could write, for example, a whole article on the development of the method for distributing, maintaining, and

1) IBM Research Laboratory, Yorktown Heights, New York, U.S.A.