ISSN 0084-6198

Algol Bulletin no.45

JANUARY 1980

CONTENTS

The ALGOL BULLETIN is produced under the auspices of the Working Group on ALGOL of the International Federation for Information Processing (IFIP WG2.1, Chairman Robert B. K. Dewar, Courant Institute).

Facilities for the reproduction of the Bulletin have been provided by courtesy of the John Rylands Library, University of Manchester.

The ALGOL BULLETIN is published approximately three times per year, at a subscription of $10 per three issues, payable in advance. Orders and remittances (made payable to IFIP) should be sent to the Editor. Payment may be made in any currency (a list of acceptable approximations in the major currencies will be sent on request), but it is the responsibility of each sender to ensure that cheques etc. are endorsed, where necessary, to conform to the currency requirements of his own country. Subscribers in countries from which the export of currency is absolutely forbidden are asked to contact the Editor, since it is not the policy of IFIP that any person should be debarred from receiving the ALGOL BULLETIN for such a reason.

The Editor of the ALGOL BULLETIN is:
Dr. C. H. Lindsey,
Department of Computer Science,
University of Manchester,
Manchester, M13 9PL,
United Kingdom.

Back numbers, when available, will be sent at $4 each. However, it is regretted that only AB32, AB34, AB35, AB38, AB39, AB41, AB42 and AB43 are currently available. The Editor would be willing to arrange for a Xerox copy of any individual paper to be made for anyone who undertook to pay for the cost of Xeroxing.

AB45.0 EDITOR'S NOTES

As you may deduce from the size of this issue (in all 3 dimensions), inflation has finally caught up with us, and those of you who receive a subscription reminder on this occasion will see that we have had to put the price up to $10 per 3 consecutive issues. Hopefully, this will enable us to rebuild our financial reserve.

My difficulty is that the price I fix now has to last for the next 18 months, and I have to make my guess as to what printing and postage costs may be by then. Be assured that IFIP makes no profit out of this operation, and if I should find that excessive funds are being accumulated they will all be passed back to you in the form of thicker issues, or extra issues, or delay of the next price increase.

AB45.1 Announcements

AB45.1.1 Book Review: TORRIX

Ref.: S.G.van der Meulen, M.Veldhorst,

TORRIX - A programming system for operations on vectors and matrices over arbitrary fields and of variable size.

MC Tracts No.86 Volume 1, Mathematisch Centrum, Amsterdam.

TORRIX is a collection of ALGOL 68 declarations, suitable for use as a library prelude. Its use is in manipulation of linear spaces (vectors and matrices), over any field. "Any field" means just that: TORRIX can be defined for arrays of integers, integers modulo some base, infinite precision numbers represented as strings, or any ALGOL 68 mode with operators that obey the algebraic rules defining a field.

TORRIX is a substantial break with past practices in several ways.

- It implements a "parametrized type": the package itself does not change when the underlying field, the type of the objects which are elements of the vectors and matrices, changes.

- An attempt has been made to follow the syntax of algebra, even to the extent of using infix operators to describe the addition and multiplication of arrays; but at the same time an attempt has been made to retain the efficiency allowed by other notations which map more clearly onto von Neumann architectures.

- Bound restrictions on vectors and arrays are removed, chiefly by (implicitly) extending them with zeroes in all directions. At the same time, programs which do not use this generality are not (substantially) adversely affected in performance.

Because of this, the TORRIX book and the system it describes are more than a manual and a system: they are an experiment in programming language design. Here are some of the important questions which are addressed:

- How suitable is ALGOL 68 for supporting a "parametrized type" What features are lacking and what features are needed?

- How suitable is the parametrized type notion to package design? Is it realistic to expect the same array manipulation primitives to be useful for arrays of different types?

- To what extent can the syntax or structure of algebra be imposed on architectures and languages that follow the von Neumann model of computation? What compromises are necessary or desirable, and why?

- To what extent can the semantics of algebra be imposed on finite computers? What compromises must be made in the use of algebraic notation to describe manipulations of the field of reals, when they are represented by the non-field of finite precision floating point numbers?

There is not space here for a detailed critique of the ways that the TORRIX system addresses these questions. Suffice it to say that the system is a major contribution to our understanding of the issue.

Incidentally, the tutorial methods used in the book are of some interest. In particular, the diagrammatic method used to explain the notion of reference in ALGOL68 and its interaction with the notion of arrays looks useful; this is an area that is fraught with difficulties for beginning ALGOL68 users, though it is apparently easy enough to use once learned.

Bruce Leverett

AB45.1.2 Book Review: Introductory ALGOL 68 Programming

Ref.: D.F.Brailsford and A.N.Walker.
Ellis Horwood Ltd. (alias Wiley), paperback edition £5.95.

Yet another textbook on ALGOL 68! What is this one particularly suited for? Certainly not as a teach-yourself beginner's book, although as an accompaniment to a biginners' lecture course it might fare better.

ALGOL 68 textbooks may be classified according to how they treat the ref problem. This one starts out with identity-declarations (real x = 24.5), followed by heap (sic) generators (ref real jim = heap real), possibly initialized (ref char letter = heap char := "a"). Then it talks of scope (hence ref real x = loc real := 17.2), and finally it gives the "abbreviated" form (real x := 17.2) and talks about variables as "ref" values thereafter. All this is pedentially true, and any teacher who chooses to teach in this way will need this book. As an adjunct to a course taught in any different way however, it would be a disaster.

(Let me declare my own position here. I consider that if you teach variable-declarations in the form loc real x := 17.2, most of the confusions disappear. You can explain (and it is almost true) that "loc" and "ref" mean the same thing, except that loc actually causes space to the generated, whereas ref always refers to a space already generated somewhere else. Unfortunately, the possibility of actually writing "loc" in a variable-declaration is only mentioned in passing in this book.)

The first two chapters build up modes, declarations and units in a complete and systematic manner (but complete programs do not appear as such until Chapter 3, which is why only a masochist could use the book in a self-teaching mode). After that, the presentation gets much better. Everything is taught with the aid of copious examples, and all details are explored with great thoroughness - as a reference book it should be quite

good. However, the so-called "Advanced features" of the language - jumps, long, short, bits, unions, casts, operation-declarations, flex, nil, is/isnt, and all programs of a "list processing" nature - receive only brief treatment in the final chapter.

The book was originally conceived in terms of ALGOL 68R and it still makes great play of the differences between ALGOL 68R and full ALGOL 68. Mostly, the differences are discussed in footnotes but, every now and then, they break out into the full text. For example, if you follow the first reference for "string" given in the index, you will find yourself in the middle of a long paragraph describing how "fish" is actually a bytes-declaration in ALGOL 68R, and the strange consequencies arising therefrom.

There are substantial appendices concerned with representations, standard-prelude, programming errors, syntax charts and implementations.

C.H.Lindsey.

AB45.1.3

Book review: Einführung in ALGOL 68

Ref.: Harry Feldmann: Einführung in ALGOL 68, Vieweg 1978, DM 29.8o

Dieses Lehrbuch wendet sich an Leser mit Programmierkenntnissen, insbesondere an ALGOL68-Kenner. Bereitschaft, mit Sprachbeschreibungsmechanismen umzugehen, wird vorausgesetzt.

Die Beschreibung der Sprache erfolgt hauptsächlich mit Hilfe von zweistufigen Syntaxdiagrammen, die am Anfang des Buches kurz erläutert werden. Die Darstellung ist sehr kompakt und für einen ALGOL68-Novizen sicher hartes Brot; hinzu kommt, daß die Diagramme mit einem Schnelldrucker hergestellt und dadurch nicht gerade übersichtlich geworden sind.

In jedem Abschnitt gibt es illustrative Beispiele und in jedem Kapitel eine Reihe von Testfragen mit Verweisen auf die zugehörige Textstelle und verdeckbaren Antworten, die die wichtigsten Aussagen des Kapitels abdecken. Das Buch hat übrigens einen ausgezeichneten Index.

Die Sprache wird - zumindest syntaktisch - weitgehend vollständig beschrieben. Leider folgt die Darstellung nicht immer dem orthogonalen Entwurf der Sprache. Beim Kopieren wird z.B. eine Unterscheidung zwischen Referenzen und anderen Werten gemacht, die sich gerade für ALGOL68 erübrigt. Ärgerlich ist, daß das Buch auch einige Fehler enthält, die sich sogar noch in den Testfragen wiederfinden. So sind z.B. Sprünge in tiefer geschachtelte Ranges auch in Sonderfällen nicht erlaubt.

Die höchste Dichte erreicht das Buch bei der Beschreibung des standard prelude. Hier muß sich der Leser durch eine lange Liste von Definitionen und seitenweise Auszüge aus dem Revised Report arbeiten. Den Sinn z.B. des straightening herauszufinden, bleibt ihm selbst überlassen.

Ergänzend enthält das Buch eine große Sammlung von Übungsaufgaben verschiedener Schwierigkeitsgrade, die sich nicht nur für einen Kurs über ALGOL68 eignen. Allerdings sind einige Aufgaben nur mit Stichworten angedeutet.

Im Anhang ist nach einer knappen Einführung in Zweistufengrammatiken die gesamte Grammatik des Revised Report in Englisch und Deutsch abgedruckt. Alle Begriffe sind wie auch im Hauptteil des Buches konsequent, manchmal etwas eigenwillig, eingedeutscht. Durch die Gegenüberstellung der beiden Fassungen ist die Korrespondenz jedoch leicht zu erkennen.

Es ist erstaunlich, wieviel Dr. Feldmann auf gut 3oo Seiten untergebracht hat. Als Einführung ist das Buch jedoch zu knapp und gibt zuwenig Hilfestellungen. Didaktisch besser gelungene Einführungen befinden sich in Van der Meulen/ Kühling: "Porgrammieren in ALGOL68", die dafür allerdings zwei Bände brauchen, und in Pagan: "Praktische Einführung in ALGOL68". Dieses Buch dagegen könnte sich eher als Nachschlagewerk eignen.

Christoph Oeters

AB45.3.1 **FUNCTIONS FOR MANIPULATING FLOATING-POINT NUMBERS**

J.K. Reid

Abstract

Floating-point arithmetic is used for most scientific and engineering calculations and needs frequently arise for determining the precision of a given floating-point number, for detailed access to its component parts and for reconstructing it from its parts, much as for complex numbers. We give detailed definitions of three suitable functions in the hope that their wide use will enhance portability. They are first given without reference to any particular language and then made specific to Fortran.

This note has been discussed by IFIP WG 2.5 on Numerical Software and has been approved by the working group, but does not constitute an official IFIP document.

Computer Science and Systems Division,
Building 8.9, A.E.R.E. Harwell,
Didcot, Oxfordshire.

June 13, 1979

1. Introduction

Virtually all scientific and engineering calculations nowadays are performed using floating-point arithmetic, that is with real numbers represented in the form

$$f \times b^m \qquad (1)$$

where b (the base or radix) is a small positive integer (usually 2, 10 or 16), m (the exponent) is a positive or negative integer and f (the fractional part or mantissa) is a number of the form

$$f = \pm \sum_{i=1}^{p} f_i \, b^{-i} , \qquad (2)$$

each f_i being an integer in the range $0 \le f_i < b$ (i.e. a base b digit). Such numbers are called "reals" in most languages, though in fact they consist of a restricted set of rationals.

IFIP Working Group 2.5 on Numerical Software proposed to the X3J3 committee that three intrinsic functions be added to FORTRAN 77 to permit the determination of the precision of a given number, the breaking up of a number into its parts and the use of parts to synthesise a number. This proposal was not accepted by X3J3, but the working group remains convinced that they are needed so that portable software may contain, for example, a precise criterion for stopping an iterative process, roundoff-free scaling, roundoff-free argument reduction for function evaluation, extended range arithmetic and very fast approximate calculation of logarithms. Explicit examples are given in sections 2 and 3.

The purpose of this document is to emphasize the need for these functions and provide their definitions in a form suitable for any language in the hope that they will become widely available and hence aid portability. In our submission to X3J3 we pointed out that certain machine parameters such as the base and the relative precision could be obtained by calls to these functions with specially chosen values of the arguments. This would have made these parameters available in a portable manner with virtually no impact on the language itself. However we feel that it is far more desirable for parameters (preferably those proposed by WG 2.5 [1]) to be directly available without "clever" function calls. Furthermore these side effects have diverted attention from the value of the functions in their own right. In fact we have made a minor change to one of the functions, set exponent (or SETXP) so that the largest and smallest possible exponents are not provided when the result would otherwise be out of range, because this speeds its execution in the ordinary case.

2. Precision function, ε

For the precision of a given processor number x we propose the function

$$\varepsilon(x) = \max\{x-x', x''-x, \sigma\} \qquad (3)$$

where x' and x" are the predecessor and successor of x in the set of processor numbers that can be stored in the main memory and σ is the underflow threshold.*

---

*The underflow threshold is the smallest positive real number σ such that no floating-point operation can cause underflow if its result is outside $(-\sigma,\sigma)$. If the predecessor or successor does not exist then x itself is used.

This function can be used directly to test convergence. As a very simple example consider the power series expansion

$$\sinh(x) = \sum_{i=0}^{\infty} \frac{x^{2i+1}}{(2i+1)!} \quad ,$$

which converges very rapidly for $|x| \leq 1$. A suitable program segment might take the form

```
sinhx:=0;  i2:=0;  term:=x;
while |term| > ε(sinhx) do
     begin sinhx=sinhx+term;
          i2:=i2+2;
          term:=term × x^2/(i2×(i2+1))
     end
```

This use of ε ensures that just as much accuracy as possible is obtained for sinhx, with proper account being taken of the fact that the precision is a piecewise constant function.

3. Functions for manipulating floating-point numbers

We propose the following functions for extracting the floating-point exponent of a given processor number x and for setting the exponent to a required value.

(a) integer exponent (x) returns the integer n that satisfies the inequalities

$$b^{n-1} \leq |x| < b^n$$

where b is the radix* of the processor representation of x. It is undefined if x is zero or n is outside the range of integer processor numbers.

*The radix b is the smallest integer greater than unity such that if r is a processor number that can be stored in the main memory and $|r| > 1$ then r/b is such a processor number and if r is such a processor number and $|r| < 1$ then r*b is such a processor number.

(b) set exponent (x,m) returns the value

$$f \times b^m$$

where f (the mantissa of x) is defined by the relations

$$x = f \times b^n, \quad b^{-1} \leq |f| < 1$$

and b is the radix of the processor representation of x, unless

a) x is zero, in which case the result is always zero;

or b) x is non-zero and m is so small or large that the result would lie under the underflow threshold or over the overflow threshold, in which case the result is undefined.

Note that the mantissa of x is available as set exponent (x,0), so there is no need for a special function for this purpose. Brown and Feldman [2] have proposed an explicit mantissa function and an explicit scale function which returns

$$\text{scale}(x,k) = x \times b^k.$$

The effect of scale may be obtained as

set exponent (x,integer exponent(x) + k)

when it is certain that the result will not be undefined because of the underflow or overflow limits. Where there is uncertainty, the size of integer exponent (x)+k should in either case be checked, though scale is defined to return a small number in the underflow case. We prefer to exclude these extra functions for the sake of simplicity since the effect may be obtained at only slight inconvenience.

As a simple example of the use of these functions, consider the calculation of the determinant of a matrix by multiplying together the diagonal elements of its triangular factorization. Without care it is likely that underflow or overflow will result but the following code determines the product in the form $x \times b^i$

```
x:=1;  i:=0;
for k:=1 step 1 until n do
begin i:=i+integer exponent(a_kk)+integer exponent(x);
      x:=set exponent(x,0)×set exponent(a_kk,0)
end
```

For modest values of n the body of the for loop may be simplified to

```
begin i:=i+integer exponent (a_kk);
      x:=x×set exponent (a_kk,0)
end
```

but for large values of n there is a danger that this will cause underflow in x.

4. Implementability

To implement the function $\varepsilon(x)$ it is necessary to interpret formula (3) in the light of hardware details. Special handling may or may not be required for the case x=0.

Each of the other functions typically requires only a few instructions to fetch or modify the exponent field of a floating-point number. It may also be necessary to shift the exponent into position, or to add or subtract a bias constant. Special care may be needed for negative x on machines where negation affects the exponent field. Also, on a twos-complement computer with the implicit b-point at the left, the machine mantissa of a negative power of 2 is -1 rather than $-\frac{1}{2}$, and therefore the machine exponent of such a number must be increased by 1. Finally, set exponent (x,n) may require special care when x=0.

The three functions have been implemented in IBM assembler language for the Harwell subroutine library with names FD02, ID04 and FD03 including versions for single and double precision. The single-precision code is given in the appendix.

5. Fortran representation

Where these functions are included in an existing Fortran library, there is no alternative to the use of the naming convention of that library, as was the case at Harwell (see section 4). Otherwise we hope that the names EPSLN,INTXP and SETXP which we used in our submission to X3J3 will be adopted. Ideally they should be the names of generic functions so that the actual function called will be the one appropriate for the type of the argument, and where this is possible we see no need for explicit names. This does however demand an extension to Fortran 77. To keep within this standard (and within Fortran 66) we propose the same names for explicit REAL versions and the names DEPSLN,INTDXP,DSETXP for explicit DOUBLE PRECISION versions.

References

[1] Parameterization of the environment for transportable numerical software. Ed. B. Ford (for WG 2.5). ACM Trans. on Math. Software, 4 (1978), 100-103; SIGNUM Newsletter, 13 (June 1978), 20-23; SIGPLAN Notices, 17 (July 1978), 27-30; IMA Bull. 14 (July 1978), 179-182; Algol Bull. 42 (May 1978), 7-10; Comp. Phys. Comm. 15 (1978), 1-3.

[2] Environment parameters and basic functions for floating-point computation. W.S. Brown and S.I. Feldman. Bell Labs. Computing Science Technical Report #72, Oct. 1978; in "Conference on the programming environment for development of numerical software", Ed. C.L. Lawson, J.P.L. report 78-92; SIGNUM newsletter, 14 (March 1979), 42-45.

APPENDIX IBM Assembly Code in Harwell Subroutine Library

```
FD02AS   CSECT
*
*    FD02AS - A SUBROUTINE TO CALCULATE THE PRECISION
*
*      OF A GIVEN SINGLE PRECISION FLOATING POINT NUMBER.  IT
*      IS THE FUNCTION EPSILON(X) PROPOSED BY IFIP WORKING GROUP
*      2.5 AND IS DEFINED TO BE
*
*           EPSLN(X) = MAX(X-X',X"-X,SIGMA)
*
*      WHERE X' AND X" ARE THE PREDECESSOR AND SUCCESSOR OF X IN
*      THE SET OF PROCESSOR NUMBERS THAT CAN BE STORED IN THE MAIN
*      MEMORY AND SIGMA IS THE UNDERFLOW THRESHOLD.
*
*    USE:  THE SUBROUTINE IS A REAL FUNCTION SUBROUTINE.
*
*          EPSLN=FD02AS(X)
*
*    X     IS A REAL FLOATING POINT NUMBER WHICH IS ASSUMED TO
*          BE CORRECTLY NORMALIZED.
*    EPSLN (FUNCTION RESULT) SET BY THE SUBROUTINE TO THE
*          PRECISION OF X AS DEFINED ABOVE.  THE FUNCTION RESULT IS
*          FLOATING POINT AND RETURNED IN REGISTER ZERO.
*
*    REGISTER USAGE:  NO REGISTERS ARE SAVED AND 0 AND 1 ARE ALTERED.
*
R0       EQU   0     WORK REGISTER (ALTERED)
R1       EQU   1     ARGUMENT LIST ON ENTRY AND WORK (ALTERED)
R14      EQU  14     RETURN ADDRESS
R15      EQU  15     ENTRY POINT AND USED AS BASE
*
*    FLOATING POINT REGISTER 0 IS USED TO RETURN FUNCTION RESULT.
*
*
*        USING FD02AS,R15    ESTABLISH BASE
*
         L     R1,0(R1)      PICK UP X INTO ...
         L     R0,0(R1)      ... A GENERAL REGISTER
         N     R0,MASK       EXTRACT EXPONENT FIELD AND ...
         S     R0,RPLEVEL    ... REDUCE TO PRECISION LEVEL
         BNL   NL            WILL EXPONENT UNDERFLOW ?
*
         SR    R0,R0         YES: SET TO UNDERFLOW LEVEL
*
NL       O     R0,F1         NO: PUT IN MANTISSA VALUE ONE
*
         ST    R0,EPSLN      TRANSFER RESULT TO ...
         LE    0,EPSLN       ... FLOATING POINT REGISTER
*
         BR    R14           RETURN TO CALLER
*
EPSLN    DC    F'0'          TO HOLD RESULT
MASK     DC    X'7F000000'   TO EXTRACT EXPONENT FIELD
RPLEVEL  DC    X'05000000'   TO REDUCE EXPONENT TO PRECISION
F1       DC    X'00100000'   TO SET MANTISSA OF RESULT
         END
```



```
ID04AS   CSECT
*
*    ID04AS - A SUBROUTINE TO EXTRACT THE EXPONENT PART OF A
*      GIVEN SINGLE OR DOUBLE PRECISION FLOATING POINT NUMBER.
*      IT IS THE FUNCTION INTEGER EXPONENT(X) PROPOSED BY IFIP
*      WORKING GROUP 2.5.
*
*    USE:  THE SUBROUTINE IS A INTEGER FUNCTION SUBROUTINE.
*
*          INTXP=ID04AS(X)
*
*    X     IS A FLOATING POINT NUMBER (SINGLE OR DOUBLE PRECISION)
*          WHICH IS ASSUMED TO BE CORRECTLY NORMALIZED.
*    INTXP (FUNCTION RESULT) SET BY THE SUBROUTINE TO THE VALUE OF
*          THE EXPONENT OF X.  THE FUNCTION RESULT IS INTEGER AND
*          RETURNED IN GENERAL REGISTER ZERO.
*
*
*    REGISTER USAGE:  NO REGISTERS ARE SAVED AND 0 AND 1 ARE ALTERED.
*
R0       EQU   0     WORK REGISTER (ALTERED)
R1       EQU   1     ARGUMENT LIST ON ENTRY AND WORK (ALTERED)
R14      EQU  14     RETURN ADDRESS
R15      EQU  15     ENTRY POINT AND USED AS BASE
*
*    FLOATING-POINT REGISTERS ARE NOT USED
*
*        USING ID04AS,R15    ESTABLISH BASE
*
         L     R1,0(R1)      EXTRACT ...
         IC    R0,0(R1)      ... EXPONENT FIELD ...
         N     R0,MASK       ... OF X
         S     R0,C64        CONVERT TO SIGNED INTEGER
         BR    R14           RETURN TO CALLER
*
C64      DC    F'64'         IBM EXPONENTS KEPT IN EXCESS 64 FORM
MASK     DC    X'0000007F'   TO EXTRACT EXPONENT FIELD
         END
```

```
FDØ3AS   CSECT
*
*    FDØ3AS - A SUBROUTINE TO BUILD A SINGLE PRECISION FLOATING
*      POINT NUMBER GIVEN A SIGNED INTEGER EXPONENT VALUE AND A
*      SINGLE PRECISION FLOATING POINT NUMBER HAVING THE REQUIRED
*      MANTISSA. IT IS THE FUNCTION SET EXPONENT(X,M) PROPOSED
*      BY IFIP WORKING GROUP 2.5.
*
*    USE:  THE SUBROUTINE IS A REAL FUNCTION SUBROUTINE
*
*        SETXP=FDØ3AS(X,M)
*
*    X   IS A REAL FLOATING POINT NUMBER CONTAINING THE
*        MANTISSA.  IT IS ASSUMED TO BE CORRECTLY NORMALIZED AND
*        IS NOT ALTERED.
*    M   IS THE INTEGER EXPONENT VALUE AND IS NOT ALTERED.
*  SETXP (FUNCTION RESULT) IS THE FLOATING POINT NUMBER HAVING THE
*        EXPONENT M AND MANTISSA OF X.
*
*    REGISTER USAGE: REGISTERS Ø AND 1 ARE ALTERED AND NOT SAVED.
*
RØ
R1       EQU   1     ARGUMENT LIST AND WORK (ALTERED)
R14      EQU   14    RETURN ADDRESS
R15      EQU   15    ENTRY ADDRESS AND BASE
*
*    FLOATING POINT REGISTER Ø IS USED TO RETURN RESULT.
*
*
         USING FDØ3AS,R15    ESTABLISH BASE
*
         L     RØ,4(R1)      ARG ADDRESS M
         L     R1,Ø(R1)      ARG ADDRESS X
         LE    Ø,Ø(R1)       COPY OVER ...
         STE   Ø,RESULT      ... ARG X
*
         LR    R1,RØ         PICK UP ...
         L     RØ,Ø(R1)      ... EXPONENT VALUE AND ...
         A     RØ,C64        ... CONVERT TO IBM EXCESS 64 FORM
*
         IC    R1,RESULT     PICK UP OLD EXPONENT FIELD
         N     R1,MASK       KEEP ONLY SIGN BIT
         OR    RØ,R1         COMBINE WITH NEW EXPONENT
         STC   RØ,RESULT     PUT IN RESULT
*
         SER   Ø,Ø           SET RESULT, ALLOWING FOR
         AE    Ø,RESULT      ...THE ZERO CASE, AND
         BR    R14           ... RETURN TO CALLER
*
RESULT   DC    F'Ø'          TO CONTAIN RESULT
C64      DC    F'64'         IBM EXPONENTS HELD IN EXCESS 64 FORM
MASK     DC    X'00000080'   TO CLEAR EXPONENT FIELD
         END
```


AB45.4.1

ALGOL 68 Implementations - FLACC

The following information regarding the FLACC implementation should be of interest to your readers:

- FLACC was developed by the Chion Corporation in cooperation with Professor B.J.Mailloux of the University of Alberta. The system runs on, and produces code for computers which support the IBM/370 problem-state instruction set.

- FLACC implements the language defined by "The Revised Report on the Algorithmic Language ALGOL 68".

- Precise implementation of the standard language was the primary goal of the project.

- The Mathematisch Centrum (Amersterdam) has developed a comprehensive acceptance test for Algol 68 implementation. FLACC meets and exceeds the acceptance requirements of the MC Testset.

- All interactions with the host operating system are handled by a tightly-defined, separate module. Operating system interfaces currently exist for the OS/VS/MVS family, CP/CMS, and MTS.

- The FLACC system is designed to perform well in virtual-memory environments. The entire system is re-entrant, and can therefore be installed in the LPA. The load-and-go version of FLACC occupies about 350K (static csects) and normally requires an additional 250K of workspace. FLACC does not require utility files for compilation.

- A batching monitor is supplied with FLACC. The monitor supports a simple job control language, and is intended for use by students. The batching monitor also eliminates the need for linkage editing and loading the object code.

- FLACC can be used as a production compiler and produces standard OS object modules. The object code performance is comparable to that of the IBM PL/I Level F compiler.

Version 1.2 of FLACC is leased for C$400.00 per month, including the production library. The VM resident compiler system, which is primarily intended for student batch use, is leased for C$287.00 per month. The lease charges apply on an installation basis, and include support which is similar to IBM class B maintenance.

FLACC may be obtained without charge for a trial period of thirty days.

Further information and licensing details may be obtained by writing to:

Chion Corporation
Box 4942
Edmonton, Alberta
CANADA T6E 5G8

AB45.4.2 ALGOL 68 Implementations - ALGOL 68C, Release 1.

The ALGOL68C language is based on the Algol 68 Revised Report and is a subset with extensions - a list of differences is given below. The compiler was developed at the University of Cambridge and achieves portability through the use of an intermediate language, ZCODE, which is tailored for the target computer.

The ALGOL68C system is designed for general use in that it is not tailored specifically for student batch work nor is it a system employing global optimisation techniques. However, attention has been paid to run-time efficiency since the compiler is itself written in ALGOL68C and so compile times depend on the generation of efficient code by the compiler.

Release 1 does not include a garbage collector, although the heap is available, and the restrictions of the previous "Prerelease" versions have been removed. The garbage collector, provision of a library mechanism, improved run-time diagnostics and the removal of some of the incompatibilities with Algol 68, as indicated below, are scheduled for Release 2 on which work has already started.

IBM 360/370 version:

OS/MVT, OS/VS2 (SVS) or OS/MVS is assumed but only minor changes are needed for OS/MFT and OS/VS1. A minimum of 180Kbytes are required by the compiler and 200K to 220Kbytes is a more realistic figure for reasonably sized programs. The Universal Instruction Set is required on 360 series computers and no 370-only instructions are generated. The system is distributed on 9-track magnetic tape written at 800 or 1600 b.p.i. in object module form and source code for the ALGOL68C and assembler parts of the run-time system are also included. The assembler part of the run-time system has provision for using certain 370-only and extended precision instructions - these are activated by assembly switches but the distributed object modules have been assembled so as not to use 370-only or extended precision instructions.

For an order form, the terms and conditions of distribution, or for further information, please write to

ALGOL68C Distribution Service,
Computer Laboratory,
Corn Exchange Street,
CAMBRIDGE, CB2 3QG,
U.K.

DEC system-10 and system-20 version:

TOPS-10 (running on a KA10 with floating-point hardware, KI10 or KL10 processor) or TOPS-20 (on the DEC-20) operating system is assumed. A minimum of about 70K of memory is required for the compiler on a KA10 or other non-VM system. The compiler requires the KA10 instruction set, and by default generates code to suit the processor on which it runs. The run-time library requires the KA10 or KI10 instruction set, as appropriate. The DEC-20 version of the compiler and run-time does NOT use the compatibility package. The compiler runs under the control of a command scanner which accepts command strings in the TOPS 10/20 manner.

The distribution tape is 9 track, 800 b.p.i., and is written in DEC-10 (BACKUP) INTERCHANGE format (readable by DUMPER on the 20). It contains two save-sets for each system (KA, KI, KL and DEC20). These save-sets hold the compiler and run-time library in object form, together with other necessary system files.

For an order form, the terms and conditions of distribution, or for further information, please write to

Dr. R.G. Blake,
Computing Service,
University of Essex,
Wivenhoe Park,
COLCHESTER, CO4 3SQ,
U.K.

Telefunken TR440/TR445 version:

Small programs are compiled in 40K KSB whereas, for large programs (around 30 pages of written code), up to 50K may be required. The code generated is comparable, in terms of speed and size, with that from the Fortran and Algol 60 compilers but the ALGOL68C compile-time is substantially longer.

Two versions of the standard prelude are available and are selected by an option in the compile command - one version is that defined in the ALGOL68C Reference Manual and the other is the full Revised Algol 68 standard prelude (with transput for all types of TR440 books) but with the exceptions of long and short modes, parallel clauses, semaphores and formatted transput.

The ALGOL68C separate compilation mechanism maps conveniently onto the TR440 library concept.

Work is in hand on the support of the NAG library under ALGOL68C and both Fortran subroutines and Algol 60 procedures may be called from ALGOL68C programs.

For an order form, the terms and conditions of distribution, or for further information, please write to

Dipl.-Math. H. Wupper,
Rechenzentrum der Ruhr-Universitat Bochum,
Postfach 102148,
D-4630 BOCHUM,
Germany.

Other machines:

Work is in progress on implementations for the PRIME 300, NORD 10 and DEC PDP11 computers; no time estimates can be given at present for these implementations.

The compiler is itself written in ALGOL68C and assumes a binary (either one's or two's complement) computer with an integer of at least 16 bits width. Although these are requirements for the computer on which the compiler runs, the compiled code may be used on other machines.

For an order form, the terms and conditions of distribution, or for further information, please write to

ALGOL68C Distribution Service,
Computer Laboratory,
Corn Exchange Street,
CAMBRIDGE, CB2 3QG,
U.K.

ALGOL68C Reference Manual

The language, but not its use, is described in formal terms by this manual but it is more readable than the Algol 68 Revised Report. It may be obtained from

Computing Service Bookshop,
Computer Laboratory,
Corn Exchange Street,
Cambridge, CB2 3QG,
U.K.

Differences between Algol 68 and ALGOL68C

- No parallel clauses (-).
- No flexible names, no vacuums (-).
- No formatted transput, no get, no put, no binary transput, no random access transput, no displays in read and print but read and print may take multiple parameters (+-*).
- An indicant may not be used as both an operator and a mode-indication (-).
- Round brackets are not available in variable-declarations of ref-row-of objects if HEAP or LOC are omitted, but ROW()AMODE may be used (+-).
- Colon-symbol must not be present in virtual rowers (-).
- := and =: are not available for operators; however, for any operator, op, the form op:= may be used as an 'assign-formula'.
- Widening of BITS and BYTES is not provided (-).
- GO TO is not available as an alternative for GOTO (-).
- Blanks may not occur in denotations (-*).
- Newlines are not permitted in tags (-*).
- The scope of an environ is local if it contains more than a single phrase even if it contains no declarations (-).
- The right operands for UPB and LWB operators are not values of a mode united from a sufficient set of modes each of which begins with 'row'; instead, a sufficient set of UPB and LWB operators are provided each taking a right operand, the mode of which begins with 'row'.
- Labels are permitted in enquiry-clauses (+).
- UPTO and DOWNTO may be used to specify the count direction in a loop-clause for cases where the sign of the increment is not determinable at compile-time (+).
- UNTIL may be used in loop clauses instead of WHILE NOT (+).
- Dyadic operator priorities from 1 to 15 may be used (+).
- Monadic-formula is a secondary (+).
- Displacements (which are similar to assignations but yield the value previously referred to by the destination) may be used (+).

- Assign-formulas (becomes-formulas, op:=, and displace-formulas, op:=:=, for an operator, op) are automatically available for all operators (+).

- Predicates are available as alternatives to boolean AND and OR to give a defined order of elaboration - thus a ANDF b is equivalent to IF a THEN b ELSE FALSE FI and a ORF b is equivalent to IF a THEN TRUE ELSE b FI (+).

- Separate compilation for program segments is provided (+).

- Escaped-characters are available in string-denotations (+).

- Square brackets may be used instead of round brackets for calls (+).

- Thef-symbol is allowed in conditional-clauses, e.g. IF a THEF b THEN c ELSE d FI (+).

- :~=: may be used as a representation of the is-not-symbol (+).

(*) It is intended that a Release 2 compiler should support get and put, binary and random-access transput, allow blanks within denotations and newlines within tags, as well as providing a garbage collector.

(-) Algol 68 sublanguage feature.

(+) Algol 68 superlanguage feature.

ALGOL68C Release 1 Charges

(June 1979)

Academic:

University, Polytechnic and selected other academic research institutions

Initial charge	£25, postage and packing extra
Maintenance charges	see below

Except that the TR440/TR445 version is free to academic users.

Other noncommercial institutions:

Initial charge	£100, postage and packing extra
Maintenance charges	see below

Commercial in-house use:

Initial charge	£400, postage and packing extra
maintenance charges	see below

Other:

Commercial bureaux, distribution to third parties and all other uses not appropriate to the above categories - subject to negotiation.

Maintenance charges:

New versions of ALGOL68C Release 1 or other maintenance material will be made available from time to time and will be supplied to all types of installation at a nominal handling fee plus postage and packing costs.

ALGOL68C Reference Manual: £1-95, postage and packing extra

Notes

1) Postage and packing is charged extra in all cases.
2) V.A.T. (value added tax) is chargeable to commercial customers.
3) Do not send money with the order as an invoice for the total charge, including postage and packing, will be sent on dispatch of the material.
4) Payments should be made in sterling.

AB45.4.3

A Proposal for Conversational Transput.

G. Baszenski and H.Wupper
(University of Bochum)

It has often been asked how an interactive terminal should be interfaced to the ALGOL 68 Transput procedures. The simplest solution to this problem is the "line by line" method, in which the terminal is regarded as two separate sequential access books opened on two separate files, one for input and one for output. The Revised Report does not specify in what manner the text typed by the user becomes part of the book. Under no circumstances should the system pause and wait for the user to type a line of characters, even an empty line, if the program has no intention of using the characters. Hence this updating of the input book should be delayed as long as possible, for instance until 'get good line' is called. Likewise, physical output of the text is not initiated until the program calls 'newline'.

This interpretation is in complete accordance with the Report, with the understanding that the input book consists of the lines typed by the user, and the output book of the lines printed by the machine, the two appearing merged on the printed page. Additional features to smooth the interaction, such as editing to remove the user's mis-typings of the input line, and automatic prompting, may be provided by the implementor, or, hopefully, by the operating system, but they should be transparent to the user program.

However, the purpose of this paper is to propose a more elaborate system, whose possibilities are illustrated by the following example of a conversation. (In this example, u indicates that the computer has unlocked the keyboard and possibly issued a prompt; s indicates that the user has finished typing and requires his message to be sent to the computer.)

```
LEVEL 1 COMMANDS:u?s
POSSIBLE VALUES:
MESSAGE
MONITOR
STOP
COMPILE
EXECUTE

CORR.:uCOMPILEs
(SOURCEFILE=uBETAs
,LANGUAGE=u?s
ALGOL68
ALGOL60
FORTRAN
COBOL
BASIC
PL/1
PASCAL
CORR.:uALGOL 69s
"ALGOL 69" NOT ALLOWED. CORR.:uALGOL68s
,PROGRAM NAME=uP1s
BETA COMPILED; PROGRAM NAME=P1 - COMPILE SUCCESSFUL

LEVEL 1 COMMANDS:uCOMPILE(GAMMA,ALGOL68,P2);
MESSAGE(P2 COMPILED SUCCESSFULLY,TERMINAL);EXECUTE(P1,DATA)s
*****GAMMA CONTAINS SYNTAX ERRORS.  - COMPILE FAILED

LEVEL 2 COMMANDS:uCOMPILE(GAMMA,PL/1s
,PROGRAM NAME=uP2s
GAMMA COMPILED; PROGRAM NAME=P2 - COMPILE SUCCESSFUL
```



```
LEVEL 2 COMMANDS:uMONITOR(OFF)s
LEVEL 2 COMMANDS:us
P2 COMPILED SUCCESSFULLY

STOP P1 2.649 SEC.
LEVEL 1 COMMANDS:uEXECUTE(P1,FILE13);EXECUTE(P2,AFILE);MONITOR(ON);
EXECUTE(P3,A3)s
+++++FILE FILE13 IS EMPTY.
ABORT P1  .035 SEC.
LEVEL 2 COMMANDS:uEXECUTE(P3,AFILE3)s
*****P3 NOT FOUND.
LEVEL 2 COMMANDS:uEXECUTE(P1,AFILE3)s
STOP P1 2.884 SEC.
LEVEL 2 COMMANDS:us
STOP P2 9.821 SEC.  - MONITOR SUCCESSFUL

*****P3 NOT FOUND.  - EXECUTE FAILED

LEVEL 1 COMMANDS:uMESSAGE(PLEASE MOUNT TAPE 123456,OPERATOR);
EXECUTE(P1,123456)s
OPER WILL NOT RECEIVE MESSAGES - MESSAGE FAILED

LEVEL 2 COMMANDS:uSTOPs
```

In this method, it is clearly defined at each instant of time whether it is the user's or the machine's turn to use the terminal. We therefor assume a mechanism whereby the keyboard may be "locked" whenever the user is not permitted to type. Ideally, this will be a physical locking but, where this is not possible, it may simply be that anything typed is not reflected (but some systems or some terminals might store up such "typed ahead" characters until a subsequent unlocking). When the keyboard is officially unlocked (which may be indicated by a bell, a red light or by a system prompt transparent to the program), the user may type his message, after which he will press some SEND key.

In an ideal system, changeover between input and output, and vice versa, may take place on any position in a line. There are, however, systems where SEND is related to the NEWLINE function in some way: sometimes SEND may imply NEWLINE (thus making it impossible for the machine to continue on the same line as the user); sometimes, sadly though, NEWLINE implies SEND (so in one conversational cycle not more than one line can be typed in). As shown in the last paragraph, even if in an operating system NEWLINE implies SEND, multi-line input may easily be simulated by the implementor inside the ALGOL 68 transput system.

Unfortunately, when the program unlocks the keyboard, there is no way to prevent the user from typing less or (what may be worse) more information than the programmer had envisaged. If too much is typed (perhaps many lines of it), then any replies printed by the program will be printed far removed from the input lines they refer to. Moreover, the program may now have to ask the user to type corrections to the earlier lines before it can return to process the remainder of the user's original input. Since the corrections typed by the user may excite further remarks from the program, with requests for even further clarifications, the whole process is clearly recursive. At one point of time there may exist several portions of input text, each separated by lines of program output, which the program may be actively engaged in reading. It therefore is important that the program may both have means to ask for new input before the previous portions of input are read up, and to detect the end of each separate portion of input. In the current proposal, these portions of input are regarded as separately opened books.

The Proposal.

The program first opens the terminal for output in the usual manner, presumably via 'stand out channel'. An input file may be opened on the same terminal, using perhaps a special 'converse channel' or perhaps some special 'idf'. During the elaboration of 'open' the keyboard then is unlocked and the program waits for the user to type his message (which may be just a few characters without NEWLINE, or which may be several lines long). When the user presses SEND, the whole of the text typed since the unlocking is deemed to be a complete book with its logical file end at the point where the SEND was pressed. The current position of the output file must now be updated to reflect the position of the typehead or cursor (note that this must be done even if the input file will be closed before its characters are ever read by the program). Beyond adjusting the position numbers this should involve no action on the part of the implementor (unless he wants to have for his output book such properties as 'get possible' and 'set possible', in which case he will actually have to copy the text to the output book). (Note that 'set', 'reset' etc. are in principle possible, perhaps even useful, on the input file.) After all this has happened, the elaboration of 'open' is completed and the program continues.

Subsequently, if more input is required, another input book must be opened (on the same or another file). Thus, several input books may be open at any time, each with its own current position, all derived from separate texts typed earlier by the user. Typically, the later ones are parts of conversations with the program clarifying matters encountered in the earlier ones. If the program has finished with one particular input text it will close the corresponding file, thus indicating that the storage used to keep the text may be returned to the system.

The following example program, that was used to produce the conversation above, shows some typical programming techniques: for each logical state a different 'on logical file end' routine has to be provided that performs the appropriate actions in case the user typed in too little. Sometimes, upon logical file end the file is closed and the program returns to a lower level of conversation (for example, if an error has been corrected, like at the end of procedure 'get item'), sometimes, after closing, a new book is opened on the same file (like in the procedure 'expect' when the input is found to be too short).

```
# Command Language Interpreter #
BEGIN
   MODE COMMAND   = STRUCT(STRING            command name,
                           [1:0]PARAMETER    par syntax,
                           PROC([]VALUE)BOOL execute      ),

        PARAMETER = STRUCT(STRING            par name,
                           TABLE             par value    ),

        VALUE     = UNION (INT,STRING),

        TABLE     = [1:0] STRING;

   [] STRING any = (),
   [] PARAMETER none = ();


   [] COMMAND command language =
      (
         ("MESSAGE", (("TEXT", any),
                      ("DESTINATION", ("TERMINAL","OPERATOR","PRINTOUT"))),
```



```
                     NEST MESSAGE
                        #  A procedure to display messages.
                           For the "NEST" notation see AB43.3.2  #  ),
         ("MONITOR", PARAMETER("STATE", ("ON","OFF")),
                     ([]VALUE p)BOOL:
                        (monitor := (p[1] | (INT i): i=1); TRUE)),
         ("STOP"   , none, ([]VALUE p)BOOL: stop),
         ("COMPILE", (("SOURCEFILE", any),
                      ("LANGUAGE",
                         ("ALGOL68","ALGOL60","FORTRAN","COBOL",
                          "BASIC","PL/1","PASCAL")),
                      ("PROGRAM NAME", any)),
                     NEST COMPILE
                        #  A procedure to invoke the appropriate
                           compiler  #  ),
         ("EXECUTE", (("PROGRAM NAME", any),
                      ("INPUT FILE", any)),
                     NEST EXECUTE
                       #  A procedure to initiate execution of
                          the named program  #  ),
         (""       , none, ([]VALUE p)BOOL: TRUE)
      );

   PROC commands = (INT prio)VOID:
         #  Ask for sequences of commands with level = prio;
            Read and execute one sequence after the other until a
            sequence is found with last command = ""  #
      WHILE # sequence of commands, level = prio #
         put(out, (newline, "LEVEL ", whole(prio,0), " COMMANDS:"));
         LOC FILE in; open for input(in);
         #  FILE in now contains one sequence of commands
            which is executed in the following loop  #
         LOC STRING cmd; #  = command name;
                            Used to terminate the current loop if cmd = ""
                            for the last command in the sequence  #
         WHILE # single command #
            INT nmb = get item(in,  command name OF command language);
            cmd := command name OF command language [nmb];
            IF (execute OF command language [nmb])
                  (parameters(in, par syntax OF command language [nmb]))
            THEN
               ( monitor AND cmd/=""
               | put(out, (" - ", cmd, " SUCCESSFUL", newline))
               );
               skip(in, ";")
            ELSE (monitor | put(out, (" - ", cmd, " FAILED", newline)));
               IF skip(in, ";")
               THEN #  Input not ended;
                       Perhaps user wants to mend error  #
                  commands(prio+1); TRUE
               ELSE FALSE
               FI
            FI
         DO SKIP OD;
         close(in);
         cmd/=""
      DO SKIP OD;

   PROC get item = (REF FILE from where, [] STRING allowed)INT:
      IF LOC INT pos;
         STRING value = get string(from where);
         string in row of string(value, pos, allowed)
```

```
    THEN pos
    ELSE
       IF value="?"
       THEN put(out, (newline, "POSSIBLE VALUES: ", newline));
          FOR i TO UPB allowed DO put(out, (allowed[i], newline)) OD
       ELSE put(out, ("""", value, """ NOT ALLOWED. "))
       FI;
       put(out, "CORR.:");
       LOC FILE corr; open for input(corr);
       pos := get item(corr, allowed);
       close(corr);
       pos
    FI;

 PROC skip = (REF FILE where, CHAR expected)BOOL:
    BEGIN LOC FILE g := where;
       on logical file end(g, (REF FILE f)BOOL: GOTO false);
       LOC CHAR ch; get(g, ch);
       IF ch/=expected
       THEN put(out, (newline, "WARNING: """, expected,
                      """ EXPECTED; """, ch, """ FOUND.", newline))
       FI;
       TRUE # End of input not yet reached #
    EXIT
    false:
       FALSE # Input was ended #
    END;

 PROC parameters = (REF FILE par,
                    [] STRUCT(STRING parname, [] STRING parvalue) syntax
                   ) [] VALUE:
    BEGIN LOC [UPB syntax] UNION(INT, STRING) actual par;
       IF UPB syntax = 0
       THEN SKIP # No parameters necessary #
       ELSE
          FOR p TO UPB syntax
          DO expect(par, (p=1 | "(" | ",",
                    parname OF syntax[p] + "=");
             actual par [p] :=
                IF [] STRING allowed = parvalue OF syntax [p];
                   UPB allowed = 0
                THEN get string(par) # Anything allowed #
                ELSE get item(par, allowed)
                FI
          OD;
          skip(par, ")")
       FI;
       actual par
    END;

 PROC get string = (REF FILE from where)STRING:
    BEGIN LOC FILE stringfile := from where;
       make term(stringfile, "(,);");
       on line end(stringfile, (REF FILE f)BOOL: (newline(f); TRUE));
       LOC STRING value;
       get(stringfile, value);
       value
    END;

 PROC expect = (REF FILE from where, CHAR what, STRING prompt)VOID:
    IF skip(from where, what)
    THEN SKIP
```



```
    ELSE close(from where);
       put(out, (what, prompt));
       #re#open for input(from where)
    FI;

 PROC open for input = (REF FILE f)VOID:
    open(f, "", converse channel);

 PROC string in row of string = (STRING value, REF INT pos,
                                 [] STRING allowed)BOOL:
    (  LOC BOOL found := FALSE;
       FOR i FROM LWB allowed TO UPB allowed
       WHILE NOT found
       DO (found := (value=allowed[i] | pos := i)
       OD;
       found
    );

    LOC FILE out; open(out, terminal, stand out channel);
    LOC BOOL monitor := TRUE;
    DO commands(1) OD
END
```

In the example above, for sake of brevity no attempt towards "batch compatibility" was made; it was assumed that the calls of 'open' for terminal input never fail and that 'get item' will succeed in any case. However, modification to a program that will run as well in batch mode, accepting the same language, but making no attempts to ask for corrections, is straightforward: if the first 'open for input' fails, 'stand in' is used instead, while failure of subsequent calls of 'open' will have to terminate the interpretation of the actual "command".

On the other hand, some batch programs may easily be turned into conversational ones by adding suitable calls of

```
PROC prompt = (REF FILE f, STRING s)VOID:
   (close(f); put(out, s); open(f, converse idf, converse channel))
```

and by

```
on logical file end(infile,
            (REF FILE f)BOOL: (prompt(f, "INCOMPLETE; GOON"); TRUE))
```

Implementation in Unfriendly Operating Systems.

While most operating systems seem to provide (physical or logical) locking and unlocking of the keyboard, many of them interpret the NEWLINE function as NEWLINE AND SEND, so by no means may more than one line be input at a time. There are other systems where each character typed in is SENT immediately, which means that input cannot be structured at all. Even in such systems a decent multi-line conversational transput may be simulated by the ALGOL 68 implementor without difficulties; implementor and user only must agree to use some special character or sequence of characters to denote the new SEND function.

When the program calls

```
   open(infile, converse idf, converse channel)
```

first of all a scratch file is established on 'stand out channel'. The transput system then loops copying lines or characters from terminal input to that file, unlocking the keyboard and writing newlines whenever necessary, without giving control back to the program. Only when the code agreed to denote SEND is detected the system closes the scratch file and reopens the same book for 'infile', this time on 'stand in channel'. Therefore, the elaboration of 'open' is completed in the usual manner; it

should not be of any relevance to the program that 'infile' is now attached to quite another channel than asked for in the call of 'open'.

The authors wish to express their thanks to C.H.Lindsey for fruitful discussions of the problem and for helping to prepare this article, and to Hartmut Ehlich, who was so interested in the proposal that he spent a few of his spare evenings to provide an implementation.

AB45.4.4

Overprinting in ALGOL 68.

D.Grune (Mathematisch Centrum, Amsterdam),
C.H.Lindsey (University of Manchester).

The Transput specified in the Revised Report does not provide any means for overprinting; in all books, a character position may be occupied by at most one character.

The Problems.

The most likely applications of overprinting are:

1. Emphasizing parts of texts by underlining, double intensity, etc. - in effect to provide extra founts of characters.
2. Producing typographical marks (e.g. ↕) not available in the machine's character code. (However, it is the responsibility of the implementor to print all characters which are in his code, even if on some devices he is forced to overlay several marks.)
3. Simulating graphic output with little blocks of different grey values, e.g. ▓ ▒ ▚ O : , .

The Solutions.

The following techniques are available to fulfil these requirements:

A. Provide special CONVs for the extra founts, etc. (and probably also a special channel to administer it all).
B. Some character is chosen to have the effect of "backspace" when included in strings that are output. A procedure 'make backsp' enables the user to specify which character, if any, is to have this property.
C. A procedure 'sameline' is provided which enables the user to specify that the next line is to be printed on top of the previous line (cf. the carriage control characters of FORTRAN). In conjunction with this, the user is given explicit control over a set of multiple buffers, either by means of a subroutine package provided for the purpose, or by letting him program it out himself.

Discussion.

Methods A and B in general require the implementor to maintain two or more line buffers behind the scenes.

We suggest that A should be the preferred solution to Problem 1 (and, moreover, it enables the same techniques to be used for other fount change applications, such as inverse video, and even photo-typesetters). It is also our preferred solution to Problem 3 (the grey levels are simply REPRs of the integers '0..max abs char'). The disadvantage of A is that it requires a new CONV for each new application, and implementors may be reluctant to provide such generous facilities. Also, it is unlikely that a book written in this way could be read back again.

Method B is the preferred solution to Problem 2. It would be quite impossible to read back a book written in this way.

Method C is the easiest to implement, especially on existing implementations. It requires no extra buffers behind the scenes, but it places the greatest onus on the user. However, if implementors are unwilling to provide methods A and B, this may be the only solution to all three problems. Moreover, it is the method most likely to permit the book to be

re-input.

Effect on the Language.

Method A involves no extra language feature, although it requires some significant effort from the implementor. The other two methods are language entensions which we now proceed to define formally.

Make backsp.

A special procedure 'make backsp' is provided which associates a string with a file. Any character in that string will then act as a genuine backspace character upon output.

For this purpose there is a new environment enquiry 'make backsp possible' and a field 'STRING ? backsp' is added to FILE.

We have then

```
PROC make backsp = (REF FILE f, STRING b) VOID:
   (make backsp possible (f) | backsp OF f := b | undefined) ;
```

Empty STRINGs have to be added in the file assignations in 'open' and 'establish', and the line 'IF found THEN' in 'put char' (10.3.3.1.b) must be replaced by

```
      IF make backsp possible (f)
         AND char in string (char, LOC INT, backsp OF f)
      THEN IF (c -:= 1) < 1 THEN undefined FI;
         C the character at position (p, 1, c) of the book is marked C
      ELIF found
      THEN
         IF C the character at position (p, 1, c) of the book is marked C
         THEN k := C an unmarked character representing a combination of
            'k' and the marked character {not necessarily different from
            'k' or the marked character, since it is not intended that an
            infinite number of composite characters be available} C
         FI;
```

Should a composite character output in this manner subsequently be re-input, it would depend on the 'conv' field of the file then in use as to what character, if any, was obtained. {However, if the marked character is already the result of a previous combination, the resulting character may perhaps not differ from 'k' or the marked character.}

Sameline.

A special procedure 'sameline' is provided which may be called in place of 'newline' in between the lines to be overprinted. It is arranged that 'line number' (10.3.1.5.b) counts only lines visibly distinct on the printed page, so that the page end event occurs only when the printed page is full. This is brought about, in terms of the model of the Revised Report, by increasing the page size each time 'sameline' is called {of course, all that has to be done in an implementation model is to omit to increase the line count}. However, if the book should subsequently be 'reset', or 'close'd and re'open'ed for input, there is in general no way in which the two kinds of line can be distinguished and it should be expected that the gremlins (10.4.2) will have renumbered the lines accordingly {so that 'line number' will no longer yield the value that it did at the time the line in question was originally 'put'}. Note that each overprinted line is of the same length as the (possibly compressed) original line.

There is a new environment enquiry 'sameline possible' which returns TRUE only if the book linked to the file is capable of supporting the facility. It is not anticipated that this facility will be possible on random-access books. Clearly, 'put possible' must be TRUE. 'sameline' itself is then defined as follows

```
   PROC sameline = (REF FILE f) VOID:
      IF NOT sameline possible (f) OR set possible (f) THEN undefined
      ELSE set write mood (f); CO Commentary 31 (AB44.3.1) is assumed CO
         newline (f);
         REF INT p = p OF cpos OF f, l = l OF cpos OF f;
         FLEXTEXT text = (text OF f | (FLEXTEXT t2): t2);
         [LWB text[p] - 1 : UPB text[p]] FLEX [1 : 0] CHAR t;
         t[  : l-2] := text[p][  : l-1];
         t[l-1] := text[p][l-1];
         t[l :  ] := text[p][l :  ];
         text[p] := t; l OF lpos OF book OF f := l -:= 1;
            CO 'cpos OF f' and 'lpos OF book OF f' both point to the
               start of the line 'text[p][l]', which is of the same length
               as the (possibly compressed) line 'text[p][l - 1]' CO
         C the line 'text[p][l]' is marked to indicate that, when it is
            eventually copied by some system-teask to a printing device,
            it is to be printed over the top of the line 'text[p][l-1]' C;
      FI ;
```

For formatted transput, a new alignment 'm' should be provided, whose effect is to call 'sameline' the number of times given by its replicator {however, there would be little point in having this number other that 0 or 1}. In the procedure 'alignment' (10.3.5.i), the following line should therefore be inserted in the appropriate place

```
      ELIF a = "m" THEN TO r DO sameline (f) OD
```

Although the 'sameline' facility is not particularly easy to use as it stands, it is suggested that a convenient way to use it would be to declare an extra file 'associate'd with a LOC [1 : level][1 : 1][1 : line length] CHAR. Procedures could be written which allow the user conveniently to assemble a (conceptual) line of text, consisting of several actual lines. For convenience, these procedures should be of mode PROC (REF FILE) VOID so as to be useable within data lists. The various lines would subsequently be written to the proper file using 'sameline'. This process might well be initiated as a result of a page end event on the 'associate'd file.

Acknowledgements.

These proposals have benefitted greatly from discussions with members of the ALGOL 68 Support Subcommittee, and especially with Hanno Wupper.

AB45.4.5

The Translation of Algol 68 into Chinese.

Lu Ru-qian

Institute of Mathematics, Acedemia Sinica
Peking, China.

I obtained a copy of the report of ALGOL 68 at the end of 1972 and decided to translate it into Chinese. It was because the Chinese computer scientists should be aware of the latest progress in their field. By the end of 1973, the work was finished and the book was published at the beginning of this year. But, over a year ago, I was told that the revised report on ALGOL 68 had appeared. I soon found the issue of ACTA INFORMATICA that carried it and began to translate the new version. This work was completed in 1977.

During the translation, I had to make some inflections in the Chinese version against the English one so as to keep both the structure of the report and its mnemonic character. In addition, the translated version must be faithful both to the report and to the Chinese language. This article will report the main changes which have been made. At the beginning of every paragraph some words from section 1.1.5 of the revised report are quoted.

I. To what degree the translation must be done.

> "The originals contained in each production tree of T may be different protonotions obtained by some uniform translation of the corresponding production tree of D."

Taking into consideration that every protonotion, which is a notion, has its meaning in English, it was decided to translate all the notions into Chinese. I did even more - all the symbols were also translated. But the metanotions were left untranslated because most of them were not English words.

II. Introducing Chinese characters.

> "Different syntactic marks {1.1.3.1.a} may be used {with a correspondingly different metaproductions rule for 'ALPHA'}."

In order to express the notions and symbols in Chinese (according to I.), it was necessary to introduce Chinese characters. On the other hand, the English letters were saved because they were still needed in describing the syntax.

There are following changes:

1. To the right side of the metaproduction rule ALPHA a hypernotion 'bold-faced Chinese characters' was added.

2. 'bold-faced Chinese characters' was also included in "small syntactic marks" in 1.1.3.1.a).(i).

3. At the end of 1.1.3.1.g, the following subsection was added:

> "When the term 'bold-faced Chinese characters' appears in a metaproduction rule, it represents all the bold-faced Chinese characters, uniformly arranged according to some fixed order, and separated by semi-colons; when it appears in 1.1.3.1.a).(i), it means the same except that the Chinese characters are separated by commas; when it appears in a production rule, it means the same except that the Chinese characters are not separated at all."

III. How to save the mnemonic character.

> "The method of derivation of the production rules and their interpretation may be changed to suit the peculiarities of the particular natural language."

In order to follow the Chinese grammar, it is necessary to rearrange the relative positions of metanotions and other elements within the hypernotions.

For example, the following original text of 4.2.1.a of the revised report:

> a) NEST mode declaration of DECS{41a}:
> mode{94d} token, NEST mode joined definition of DECS{41b, c}.

was translated as follows (according to the order in Chinese):

> a) DECS of NEST mode declaration{41a}:
> mode{94d} token, DECS of NEST mode definition joined{41.b.c}.

IV. How to avoid ambiguities.

> "In a highly inflected natural language, it may be necessary to introduce some inflections into the hypernotions."

The problem of ambiguity relates mainly to the translation of modes. Because of the difference between Chinese and English, some previously unambiguous mode names become ambiguous during the translation.

1. The Chinese translation of

union of reference to integral real mode

and

reference to union of integral real mode

would be the same, i.e.

reference to integral real union mode.

2. The following would be alike for the Chinese translation:

reference to row of integer

and

row of reference to integer.

Both would be

reference to integer row.

3. In Chinese,

reference to procedure yielding integer

and

procedure yielding reference to integer

would be the same, i.e.

reference to integer procedure.

In order to avoid these ambiguities, there must be some forms of parentheses. In the Chinese translation, 'from' and 'union mode' were used as parentheses (like the English parentheses 'union of' and 'mode') for UNION; 'one' and 'mode' as parentheses for 'ROWS of mode'; 'without parameter' and 'procedure' as parentheses for 'procedure yielding MOID'.

Thus the six modes above would be translated as

a) from reference to integer real union mode.

b) reference to from integer real union mode.

c) reference to one integer row.

d) one reference to integer row.

e) reference to without parameter integer procedure.

f) without parameter reference to integer procedure.

V. On the equivalence of modes.

"A more elaborate definition of 'equivalence' between protonotions".

In order to test the equivalence of MODEs, the revised report splits every MODE into two parts, i.e. HEAD and TAILETY. But this is not sufficient for the Chinese translation. A MODE must be splitted into three parts, i.e. HEAD, TAILETY and APPENDIXETY. It was decided to make this inflection because there were three HEADs, i.e. 'PREF', 'FLEXETY ROWS of' and 'procedure with', which must be splitted into two parts in the translation, one placed before TAILETY, the other after TAILETY. This fact aroused a lot of changes in the corresponding section. Below a list of these changes is given.

1. The first statement of the second paragraph of Chapter 7 is now

"Modes are composed from the primitive modes, such as 'boolean', with the aid of 'HEAD's, such as 'structured with', and 'APPENDIXETY's, such as 'procedure', and they may be recursive."

2. 7.1.1.A was changed to

A) PREF :: without parameter; REF to.

3. In 7.3.1 following changes were made:

B) HEAD :: PLAIN; PREF{71A}; structured with;
FLEXETY one; with; from; void.

C) TAILETY :: MOID; FIELDS mode; PARAMETERS yielding MOID;
MOODS union mode; EMPTY.

D) APPENDIXETY :: ROWS mode; procedures; EMPTY.

b) .....
WHETHER (HEAD3) is (HEAD4) and (APPENDIXETY3) is (APPENDIXETY4)

where SAFE3 HEAD3 TAILETY3 APPENDIXETY3 develops from SAFE1 MOID1{c}
and SAFE4 HEAD4 TAILETY4 APPENDIXETY4 develops from SAFE2 MOID2{c}.

c) WHETHER SAFE2 HEAD TAILETY APPENDIXETY develops from SAFE1 MOID{b,c}:
where (MOID) is (HEAD TAILETY APPENDIXETY),
.....

WHETHER SAFE2 HEAD TAILETY APPENDIXETY develops
from MU has MODE SAFE1 MODE{c};
.....

WHETHER SAFE2 HEAD TAILETY APPENDIXETY develops from SAFE1 MODE{c}.

4. From line 7 in the pragmatic of this section (p.105), the text was changed as follows:

..... and split into its 'HEAD', its 'TAILETY', and its 'APPENDIXETY', e.g. 'without parameter MOID procedure' is splitted into 'without parameter', 'MOID' and 'procedure'.
If the 'HEAD's and 'APPENDIXETY's differ, then the matter is settled (rule b); otherwise the 'TAILETY's are analysed according to their structure (which must be the same if the 'HEAD's and 'APPENDIXETY's are identical). In each case, except where the 'HEAD's were 'from', ...

VI. On predicates.

"Descendents of those production trees need not be the same if their originals are predicates."

Since the bold-faced Chinese characters were also introduced as small syntactic symbols, there must be corresponding changes for the predicates. In fact, the production rule 1.3.1.j was changed to:

j) .....
WHETHER (ALPHA1) coincides with (ALPHA2) in
(abcdefghijklmnopqrstuvwxyz bold-faced Chinese characters){k,1,-}
.....

VII. On paranotions.

"Different inflections for paranotions". "Some pragmatic remarks {1.1.2} may be changed."

In the Chinese translation we need not be worried about the inflections of paranotions when they appear at the beginning of a statement or in the plural from. There is no difference between "capital" and "small" Chinese characters. Nor is there need to add "s" at the end of a name. Besides, hyphens are also not needed. Thus, all sub-sections dealing with this theme were deleted from the revised report (p.28, from line 14 till line 28).

VIII. On the terminal symbols.

"T defines the same reference language {9.4} and the same standard environment {10} as D."

By translating the terminal symbols into Chinese the mnemonic character of these symbols were taken into account. It was somewhat difficult to translate the 'bold to symbol', which is used in the revised report both in the to-part of a loop clause and in go-to-option of a strong-MOID-NEST-jump. There is no Chinese word having the meaning of both. Hence in the Chinese version there are two terminal symbols corresponding to 'bold to symbol', one of which is 'bold end value symbol' (used in loop-clauses), the other is 'bold to symbol' (used in go-to-option). They have the same representation.

IX. On metaproductions.

"Additional means for the creation of extra metaproduction rules".

A new metaproduction rule for APPENDIXETY is introduced, while some other metaproductions are modified. In fact, we know this already from the discussion above.

AB45.4.6 **ALGOL 68 and Algebraic Manipulation.**

D. C. Ince.
The Open University, Milton Keynes, U.K.

The decade has seen research into algebraic manipulation by computer progress to the point where the majority of computer users have access to at least one algebraic manipulation system. The growth of these systems has not however been accompanied by a corresponding growth of reports of their applications. A number of reasons have been put forward by users and potential users to explain this disparity, of which the two that occur the most are:

1) The user interface for a number of present systems is poor, manipulations on algebraic expressions being expressed in a non-natural way, usually by means of a series of subroutine calls, or in a language which, although well suited to the construction of such systems (eg LISP), tends to be alien to the programming experience of the average scientist or engineer that make up the potential user community. ABC ALGOL and its derivatives (1) and SAC (2) are examples of systems to which this criticism can be applied, while even (3), the only previous attempt to write an algebraic manipulation system in Algol 68 could have this criticism levelled at it.

2) The majority of users do not wish to perform algebraic manipulations in isolation, the manipulations being just one stage in what may be a large program in which numerical or even other symbolic processes may take a dominating part. A number of systems, CAMAL (4) and REDUCE (6) for example, while having excellent facilities for algebraic

manipulation tend to fall short with respect to the numerical and data structuring facilities required and even fall short with respect to the control facilities necessary for efficient programming.

There does however exist in Algol 68 a number of features which not only make it emminently suitable for constructing algebraic manipulation systems but also lead to the construction of systems to which the above two objections do not apply. These features are:

1) Operator declarations

The availability of operator declaration in Algol 68 means that a user can express his manipulations in a manner near to his normal working notation. It should be possible in an Algol 68 based system to write

$$a := (1 + x)*(1 + y - x)$$

assuming declarations of +, -, * on suitable data structures, rather than a series of subroutine calls, or worse still as a LISP expression. This facility alone removes the first objection outlined previously.

2) List processing facility

A property of the problems that are capable of solution by an algebraic manipulation system is that the user is unable to specify in advance the amount of storage required for the algebraic expressions that are created. It is for this reason that almost all

present systems are based on list structures and languages that are particularly good in handling lists. Although Algol 68 falls short, in terms of list processing facilities, of those languages that have been used to construct algebraic manipulation systems it does however provide enough facilities for the construction of such systems.

3) Additional numeric modes

One of the alarming properties of the problems that have been solved by algebraic manipulation systems is the speed at which numeric coefficients expand and overflow the exact arithmetic capability of the host computer. An example of such a problem is the computation of the f and g series (5) of celestial mechanics which overflows a 48 bit integer after 15 iterations. One solution adopted by a number of systems is to convert to floating point when overflow has occured. This solution is not ideal by any means as often the user of an algebraic manipulation system is interested in the patterns generated by his programs, thus it would be a lot more difficult for a user to discern a pattern in

$$.400x + .429x\uparrow 2 + .444x\uparrow 3 + .455x\uparrow 4$$

rather than

$$2/5x + 3/7x\uparrow 2 + 4/9x\uparrow 3 + 5/11x\uparrow 4$$

An alternative solution to the problem of integer overflow is to arrange that numeric coefficients can be represented in a number of alternative integer forms of increasing maximum magnitude, thus a language such as Algol 68, which has the capability for additional numeric modes as well as the ability to switch between these modes via unions, would seem to be the answer.

An algebraic manipulation system, written in Algol 68R, has been developed to handle polynomials in arbitrarily many indeterminates with real or long int coefficients. It consists of a series of operators and procedures which operate on expressions defined by

```
mode expression = ref term;
mode term       = struct (ref [ ] element factor, coeff coeff,
                          int order, ref term next term);
mode coeff      = union (real, rational);
mode rational   = struct (long int num, denom);
mode element    = struct (int atom, power)
```

Where term describes a term in a polynomial and coeff describes the numeric coefficients in each term. The system was written with the aim of making the user interface compatible not only with the normal algebraic notation familiar to its potential users but with the numeric facilities available in Algol 68.

The system consists of a series of operators and procedures which can be separated into a number of categories:

1) The common algebraic operations defined on polynomials
2) Procedures for the input and display of polynomials
3) Procedures for the integration and differentiation of polynomials
4) Procedures for the numeric evaluation of polynomials and symbolic substitution in polynomials
5) Procedures for the selection of polynomial terms based on a variety of criteria.

In order to illustrate the system I shall take two problems and display the programs needed to solve them.

1) The generation of Legendre polynomials

using the relation

$$P_n(x) = \frac{(2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)}{n}$$

where $P_o(x) = 1$ and $P_1(x) = x$

and $P_n(x)$ is the nth Legendre Polynomial

the program is

```
'BEGIN'                                  'C' GENERATION OF THE FIRST   'C'
'EXPRESSION' X = GENEXP("X");            'C' 20 LEGENDRE POLYNOMIALS   'C'
[0:20] 'EXPRESSION' P;                   'C' BY RECURRENCE RELATION    'C'
P[0]:= 'EXPRN' 1;
P[1]:= +X;
PRINT ("P[0]="); EXPOUT (P[0]);
PRINT ("P[1]="); EXPOUT (P[1]);
'FOR' I 'FROM' 2 'TO' 20 'DO'
    'BEGIN'
    P[I]:=((2*I-1)*X*P[I-1]-(I-1)*P[I-2])/I;
    PRINT (("P[", I, "]="));
    EXPOUT (P[I])
    'END'
'END'
'FINISH'
```

2) The computation of the f and g series in celestial mechanics

The f and g series are given by

$$f_n = u\frac{\partial f}{\partial a}_{n-1} + v\frac{\partial f}{\partial b}_{n-1} + w\frac{\partial f}{\partial c}_{n-1} - a\,g_{n-1}$$

$$g_n = u\frac{\partial g}{\partial a}_{n-1} + v\frac{\partial g}{\partial b}_{n-1} + w\frac{\partial g}{\partial c}_{n-1} + f_{n-1}$$

where $u = -3ab$, $v = c - 2b^2$, $w = -ab - 2bc$ and $f_o = 1$

and $g_o = 0$

The program for this computation is

```
'BEGIN'                          'C' CALCULATION OF THE FIRST 'C'
[0:20]'EXPRESSION'F,G;           'C' 20 TERMS OF THE F AND G  'C'
'EXPRESSION' A= GENEXP("A"),     'C'       SERIES             'C'
             B= GENEXP("B"),
             C= GENEXP("C"),
             U= GENEXP("-3*A*B"),
             V= GENEXP("C-2*B↑2"),
             W= GENEXP("-B*(A+2*C)");
F[0]:= 'EXPRN' 1;
G[0]:= 'EXPRN' 0;
PRINT ("F[0]="); EXPOUT (F[0]);
PRINT ("G[0]="); EXPOUT (G[0]);
'FOR' I 'TO' 20 'DO'
   'BEGIN'
   F[I]:= DIFF(F[I-1],A)*U + DIFF(F[I-1],B)*V + DIFF(F[I-1],C)*W
          - G[I-1]*A;
   G[I]:= DIFF(G[I-1],A)*U + DIFF(G[I-1],B)*V + DIFF(G[I-1],C)*W
          + F[I-1];
   PRINT (("F[", I, "]=")); EXPOUT (F[I]);
   PRINT (("G[", I, "]=")); EXPOUT (G[I]);
   'END
'END'
'FINISH'
```


1. Riet R P Van de

'ABC Algol: A language for formula manipulation systems - Part 1 The language'

Mathematical Centre Tracts No 46, 1975.

2. Collins G E

'The SAC1 system for algebraic manipulation'

Proc. 2nd symposium on symbolic and algebraic manipulation 1971.

3. Danicic I, Long F W

'Algebraic manipulation of polynomials in several indeterminates'

Proc. conference on application of Algol 68, 1976.

4. Barton D, Bourne S R, Fitch J P

'An algebra system'

Computer Journal. Vol 13, No 1, 1970.

5. Barton D, Bourne S R, Fitch J P, Harton J R

'Some applications of the Cambridge Algebra System'

University of Cambridge Computing Lab Technical Manual March 1971.

6. Hearn A C

'REDUCE 2 users manual'

U of Utah, Salt Lake City, Utah. 2nd edtn 1973.

AB45.4.7

An Algorithm for the Execution of Limited Entry Decision Tables in ALGOL 68

The ALGOL 68 procedure shown in Fig. 1 is used for the execution of limited entry decision tables described by the modes also shown in Fig. 1. The procedure is the kernel of a mixed entry decision table system, written by the author and embedded in Algol 68C, which deals with more general decision tables, has extensive statistics gathering and trace facilities, and also contains procedures used in checking for completeness, ambiguity, and redundancy. The actual procedure used in executing limited entry decision tables and the modes involved may be of interest to programmers who are involved in the production of ALGOL 68 programs containing a large number of IF..ELSE..FI constructs.

The procedure has been modified for publication in two ways. Firstly the statistics and trace gathering procedures which are used to monitor the execution of the decision table have been removed. Secondly the non-standard way that ALGOL 68C handles access to individual characters in a string, via an ELEM operator, has been changed to normal string subscripting.

D.C. Ince,
Faculty of Mathematics,
The Open University,
Walton Hall,
Milton Keynes,
Bucks.


```
 # modes used in decision tables #

MODE CONDITION = PROC BOOL;
MODE ACTION = PROC VOID;
MODE ACTORCOND= UNION(ACTION,CONDITION);
MODE TABROW = STRUCT(ACTORCOND pr,STRING entry);
MODE DECISIONTABLE = [maxrow]TABROW;


# procedures used for executing decision tables #

PROC execute =(INT entrycolumn,DECISIONTABLE table)VOID:
#executes the decision table held in table, entrycolumn #
#is the rule number at which the execution starts       #
   BEGIN
   INT column:=entrycolumn,finalcolumn:=UPB(entry OF table[1]),
       finalrow:=UPB table,row;
   BOOL conditionflag,invokeflag:=FALSE;
   WHILE column<=finalcolumn AND NOT invokeflag DO
      conditionflag:= TRUE;row:=1;
      WHILE row<=finalrow AND conditionflag DO
         IF (entry OF table[row])[column] /="-" THEN
            CASE pr OF table[row] IN
              (CONDITION c):(BOOL f1:=(entry OF table[row])[column]="T";
                             conditionflag:=conditionflag AND (IF f1 THEN
                                                 c ELSE NOT c FI)),
                  (ACTION a):(a;invokeflag:=TRUE)
            ESAC
         FI;
         row+:=1
      OD;
      column+:=1
   OD
   END;
```