ISSN 0084-6198

The Last Algol Bulletin no. 52

AUGUST 1988

CONTENTS PAGE

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 Helmut Partsch, University of Nijmegen).

Facilities for the reproduction of the Bulletin have been provided by courtesy of the John Rylands Library, University of Manchester. Word-processing facilities have been provided by the Barclay's Microprocessor Unit, University of Manchester, using their Vuwriter system.

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 $4.50 (or £2.40) each. However, it is regretted that only AB32, AB34, AB35, AB36, AB38-43 and AB45 onwards 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.

AB52.0 EDITOR'S NOTES.

As you will have noticed, it is almost four years since the last issue of the ALGOL Bulletin. The reason is, as usual, lack of material. Clearly, some consdideration of the future is called for. Let us first consider why the AB exists in the first place.

The ALGOL Bulletin is the official journal of IFIP Working Group 2.1, which is the body responsible for both ALGOL 60 and ALGOL 68. However, for many years past the primary focus of attention of the Working Group has moved away from algorithmic languages towards specification languages and program transformation paradigms, but this has not been reflected in the contents of the AB, which as I have always seen it, is primarily a vehicle for mutual support amongst the fans of ALGOL (both of them), and for the promulgation of official pronouncments from the WG (new language extensions, interpretations of the Report, and the like). In all of this, I believe that the Bulletin has played a valuable role. Now, however, ALGOL 68 as a language is very stable. It is used and loved by those who understand its benefits, and ignored (or misquoted) by the rest, and this is a steady state that will doubtless continue for some long time.

The Working Group discussed all this at some length, and finally came to the conclusion that the ALGOL Bulletin had now fulfilled its primary purpose, and that the time had therefore come for it to be laid to rest.

This current issue will therefore be the last. Since the majority of subscriptions also expire with this issue, that is also an administrative convenience. The expiry number of your present subscription will be found on your mailing label and, if it is greater than 52, then you will find a refund for the unexpired portion enclosed with this issue (or, if you use a subscription agency, the refund will have been sent there). I shall maintain the mailing list, in case we ever need to circulate the ALGOL community again, and I am still able to supply most back numbers for the usual fee.

However, it is possible that a Phoenix will indeed rise from the ashes. In this day and age it is more appropriate to keep in touch with a community of like-minded individuals by electronic means, and there has therefore been a proposal to set up an ALGOL Bulletin Board as either a moderated newsgroup or as a mailing list on USENET. Robert Dewar (cmcl2!acf2!dewar) and Robert Uzgalis (ucla-cs!buz) have been appointed as joint editors to try to set this up. If and when all of this has come to pass, we shall post an announcement in comp.lang.misc. Also, (in the absence of a more specific ALGOL 68 newsgroup), anything which the ALGOL community should know about will also be posted there.

It is now 16 years since I was appointed as Editor of the ALGOL Bulletin, and in that time I have produced 18 issues. I wish it had been possible to produce more, but it was not to be. And although it may be the end of the era for the ALGOL Bulletin, I am quite sure that it is not yet the end of the era for ALGOL.

AB52.1 Announcements.

AB52.1.1 Death of Prof. A van Wijngaarden

PROF. DR. IR. A. VAN WIJNGAARDEN has died on the 7th of February 1987. He was 70 years old.

Aad van Wijngaarden was one of the giants of the early Informatics in his country. At a time when there was no Informatics, when computers were often still called electronic brains, he decided that Holland could play a role in that area. He supervised the construction of most of the early computers in the Netherlands. He taught the first courses of algorithmics and introduced a generation of mathematicians to Informatics.

Outside of Holland he is mainly known for his contribution to the ALGOL effort. He was strongly involved in the development of ALGOL 60, right from the start. But his main venture was the design of ALGOL 68, to which he gave all his heart.

ALGOL 68 is now already history. Twenty years is very long in a branch of science where every five years a revolution is preached, where the majority of researchers have less than five years experience and therefore are convinced that the world started when they joined it. In actual fact, the developments in Informatics are not so fast and there is a clear continuity from the early sixties to the present time.

ALGOL 68 prepared the ground for functional languages by its clarification of concepts and its orthogonal expression structure. It presented compilermakers with a formidable challenge and thereby raised the state of the art enormously. The main criticism raised against it, that its definition is unreadable due to its formality, sounds somewhat ironic in the light of the ever-increasing level of formality in modern Informatics.

Although there is little present use of ALGOL 68, the language has had a lasting influence on the thinking about programming languages. Terms like coercion and references are common even to a language like C, although few people remember who pioneered those concepts.

Aad always insisted on elegance and precision in expression, on using formality where it counts. This attitude shaped both the language ALGOL 68 and its description. I am proud to have played, at his request and under his guidance, a minor role in the intellectual adventure of ALGOL 68.

Aad has influenced many people decisively in their scientific career. As a professor and as director of the Mathematical Centre for twenty years he formed practically all of the people who are presently responsible for the science of Informatics in the Netherlands.

Aad was a man about whom many anecdotes have been told; all who knew him will tell their own. At the MC we called him "baasje", the small boss. He could be wonderfully charming and totally exasperating. He was the ideal professor, teaching, inspiring and stimulating us all.

Those who, like me, have been his closest disciples, will remember also many struggles with him. Because he was such a father to us, it was very hard to go our own way when the time had come. In each case Aad has, I think, warmheartedly supported and encouraged us on our separate ways.

The last ten years have been very hard for Aad. Personal tragedy struck him, when he lost also his second wife. Professionally, his great work was completed and by no means warmly accepted. He must have had a feeling of being left by the wayside. He put himself aloof from the further development of Informatics and devoted himself to his children and his hobbies.

Now he has died. Nobody can hurt him anymore. Aad has led a full life. He has made his mark on the world. His memory is cherished by his children and by his pupils. Nobody can hope for more in his lifetime.

Nijmegen, February 1987
C.H.A. Koster.

AB52.1.2 FLACC

In order to increase the availability of ALGOL 68 for research and educational purposes, Chion Corporation is making the most recent version of FLACC, FLACC V1.7S, available in an inexpensive, unsupported version.

This in now the only version of FLACC which Chion is marketing (FLACC V1.4U has been discontinued). The supported version of FLACC V1.7S has proved very reliable, requiring no maintenance for over two years. We decided it was unfair to continue charging rental rates, so this version is now being offerred for an inexpensive, one-time charge of C$2000, with a 25% discount for educational institutions.

FLACC V1.7S runs either as a load-and-go system, or as a production compiler which produces object modules. It is also considerably faster than FLACC V1.4U.

FLACC runs on IBM machines and their clones, under the operating systems MVS, MVT, CMS and MTS.

Further information from
Chion Corporation
P.O. Box 4942
South Edmonton
Alberta
Canada T6E 5G8

AB52.1.3 Some Anniversaries

As we went to press, I just realised that two important anniversaries have crept up unnoticed.

The first is the 30th anniversary of the meeting, in May 1968, at which for the first time scientists from the USA and Europe agreed on a universal programming language. This was initially known as IAL, the International Algorithmic Language. It was later christened ALGOL 58, and its main implementation, JOVIAL, is still alive and well.

The second is, of course, the 20th anniversary of ALGOL 68, which was finally voted into being by IFIP WG2.1 on 20th December 1968 (see an account culled from early versions of the AB elsewhere in this issue). ALGOL 68 also is still alive, and even well in those places where its true worth is still appreciated. The plain fact is that, in 20 years, nobody has really come up with anything better as a complete language package, although there are of course several features fashionable in modern languages which ALGOL 68 does not possess.

AB52.1.4 Russian Standard for ALGOL 68

In spite of the failure to obtain an ISO Standard for ALGOL 68, the Soviet Union, where use of the language is apparently quite significant, is pressing ahead with a Russian Standard. This will come in two parts. The first will be essentially the Russian translation of the Revised Report, as already published by MIR Publishers, but with modified hardware representations to cope with the Cyrillic alphabet, and including also the IFIP Standard Hardware Representation. The second will contain optional language extensions, specifically the Modules and Separate Compilation proposal from AB43.3.2 together with a home-growm Exception Handling mechanism (described elsewhere in this issue).

AB52.1.5 Data Protection Act

Although this is the last issue of the ALGOL Bulletin, it is my intention to keep the mailing list intact, in case we ever need to reach the ALGOL community again.

You should therefore take note that I keep the mailing list on a computer file, that each of your names and addresses is on that file, and that in accordance with the U.K. Data Protection Act you are entitled to be aware of that fact and to register any objection.

AB 52.3.1 Survey of Viable ALGOL 68 Implementations

This Survey has been restricted to implementations which you can actually obtain and use. Each of them has an identifiable person or organisation responsible for its maintenance, and most have been used on at least one site other than that where it was developed.

Most of the column headings are self-explanatory. "Deviation" means that it is possible to write some program, valid and with defined meaning both in the given implementation and according to the Revised Report, which will provide results different from those defined by the Revised Report. Under "Money", "nominal" usually means under $200, "yes" means a realistic commercial rate. "MC Test" means that it has been tested using the MC Test Set (see AB 44.1.2) and that the implementor claims it ran correctly. In all cases, the people listed in the last column should be able to provide further information.

Name of System	Hardware	Operating System	Principal Sublanguage features	Principal Superlanguage features	Deviations?	Money?	MC Test?	Other features	Where to obtain it
FLACC	IBM 370 Amdahl Siemens	OS/VS/MVS /MFT/MVT CP/CMS MTS		exception handling Fortran interface	No	Yes	Yes	load and go version available very complete checking (hence not-so-fast running)	Chion Corporation Box 4942, EDMONTON Alberta Canada T6E 5G8
ALGOL 68C Release 1	IBM 360 IBM 370	OS/MVT OS/VS2 OS/MVS OS/MFT OS/VS1	no *sema* no *flex* no *format* restricted transput	automatic *op*:= for any *op* *upto*, *downto* and *until* in loop-clauses displacement operator (:=:=) *andf*, *orf* and *thef* separate compilation scopes not checked	Yes	Nominal to Universities	No	fast running no garbage collector	ALGOL 68C Distribution Service Computer Laboratory Corn Exchange Street CAMBRIDGE CB2 3QC United Kingdom
		VM/CMS							Robert Hill Computing Service University of Leeds LEEDS LS2 9JT United Kingdom
	DEC-10 DEC-20	TOPS-10 TOPS-20							Dr R. G. Blake Computing Service University of Essex Wivenhoe Park COLCHESTER CO4 3SQ United Kingdom
	DEC VAX	BSD 4.2 VMS							ALGOL 68C Distribution Service (see above)
	Prime								Enquiries to ALGOL 68C Distribution Service (see above)
	Tele-funken TR440 TR445	BS3	improved transput available						Klaus Hackenberg Rechenzentrum der Ruhr-Universitaet Postfach 102148 D-4630 BOCHUM Federal German Republic
	CYBER 205	VSOS 2.3	additional operations for vectors & matrices						Klaus Hackenberg (see above)

Name of System	Hardware	Operating System	Principal Sublanguage features	Principal Superlanguage features	Deviations?	Money?	MC Test?	Other features	Where to obtain it
	TESLA 200 (similar to IBM 360)		no *flex* (except *string*) no *union* no *sema* no *heap*	bounds in formal-declarers	No	No	No	TRACE facility independent compilation of routines fast running	J. Nadrchal Institute of Physics Czechoslovak Academy of Sciences 180 40 PRAHA 8 Na Slovance 2 Czechoslovakia
CONTROL DATA ALGOL 68	CDC 6000 -7000 170 series	NOS 2 NOS/BE SCOPE 2	one *long* flexibility is an attribute of a multiple value	no transient name restriction ICF macros allow definition of operators in machine instructions	No	Yes	Yes	separate compilation	Control Data Services P.B. 111 RIJSWIJK (24) The Netherlands
A68S	CDC Cyber PERQ VAX SUN3	NOS 2 NOS/BE SCOPE 2.1 PNX 2 BSD 4.2,3	official sublanguage (SIGPLAN Notices 12 5 May 1977 or Informal Introduction Appendix 4) but *heap* is allowed		No	Nominal	No	very complete checking fast compilation slow running VAX and SUN versions use the Amsterdam Compiler Kit	Dr C. H. Lindsey Dept. of Computer Science University of Manchester MANCHESTER M13 9PL United Kingdom
A68RS	ICL 2900	VME/B	indicators to be declared before use no *sema* scopes not checked	*mode vector* indexable structures *forall* elements of array no transient name restriction modular compilation	Yes	Yes	Yes	source-level symbolic diagnostics	ICL local sales office
	Honeywell Level 68/DPS	Multics			Yes	Nominal to Universities	Yes	re-entrant object code source-level symbolic debugger	Richard Wendland Praxis Systems Limited 20 Manvers Street BATH BA1 1PX United Kingdom
	DEC VAX	VMS			Yes	Yes	Yes	re-entrant compiler and object code source-level symbolic debugger	Products Group SPL International Research Centre The Charter ABINGDON OX14 3LZ United Kingdom
	UNIVAC 1100 series	EXEC VIII	no garbage collector scopes not checked	*bin* of any primitive mode complex mathematical functions *min* and *max* matrix and vector operators exception handling	Yes	No	Yes	French representations (inhibitable by pragmat), independant compilation of routines	Daniel Taupin Laboratoire de Physique des Solides Universite de Paris XI 91405 ORSAY France
	IBM PC Sirius Victor Apricot RM NIMBUS	MS-DOS	no *par*, *format*, *goto*, *bytes*, *long*, *short*, *heap*, *flex*, *string* no anonymous routine texts restricted scope of arrays, restricted *[][]* modes	*mode address* for access to memory-mapped addresses access to MS-DOS primitives, machine coded subroutines and 8087 chip features		Yes	No	incremental compilation with immediate execution many of the missing features will appear in later releases	Algol Applications Ltd 11 Wessex Way, Grove WANTAGE Oxon OX12 0BS United Kingdom

AB52.4.1

Implementation of ALGOL68 on the CYBER 205

Klaus Hackenberg
(Ruhr–Universität Bochum, Rechenzentrum)

1. General design goals

The programing language FORTRAN is widely used on today's vector and parallel computers, but it is lacking nearly all relevant concepts for these new machine architectures. The remedy of introducing machine specific language extensions and subroutine calls requires great effort in program design and prohibits the portability of programs.

Therefore ALGOL68 was implemented on the CYBER 205 in Bochum, a language which is more suitable for new machine architectures. On one hand it contains vector operations (like the assignation of rows) and on the other it allows the definition of new operators within the existing language. Together with the adaption of the ALGOL68C compiler (from the University of Cambridge) a prelude with vector operations has been designed which allows an efficient use of the special CYBER 205 hardware.

The problems arising on the CYBER 205 when programing in FORTRAN 200 may be divided into three classes :

a) Problems depending on the hardware design : The vector size is limited to 65535 elements on the CYBER 205 and the elements being processed by a vector instruction have to be consecutive in memory. All these restrictions have to be handled explicitely by the programer.

b) The notation for special subroutine calls (for instance using semicolons, empty parameters and hexadecimal specification of the desired subinstruction by so–called "G–bits") together with an unusual choice for names (for instance ABS, CABS, VABS, VCABS for essentially the same function namely "absolute value") which differs from the FORTRAN concept of "generic names" result in less readable and less understandable vectorised programs.

c) Programs written in FORTRAN 200 cannot be run without alterations on other computers, because of their special notation and special subroutine calls.

Therefore the following design goals for an ALGOL68 vector prelude where set up which allow portable user's programs together with an efficient use of the special hardware properties of the CYBER 205 :

a) From the user's point of view there should be no restriction for the size of vectors (which means that any hardware vector instruction is repeated as often as necessary automatically). There should be no limitation to consecutive vectors in memory (which means that an appropriate GATHER or SCATTER operation is done automatically whenever necessary).

b) The same name should be used for the same function on different arguments (for instance *abs* should be used to denote the absolute value for scalars and (elementwise) for vectors and matrices — so called "overloading" of operator symbols. All notations should be standard language constructs. New operators should behave like other language constructs; therefore for instance equal lower and upper bounds are required for operations on rows, empty rows are permitted and so on.

c) The user's programs should be portable. Therefore the definitions of new operators are gathered into a prelude so that any machine dependency is in that prelude — and the user's programs can be run without alteration on any other (scalar or vector) computer. One version of this prelude is given using only scalar ALGOL68 so that the new operators can be easily implemented on any other computer. A second version of the prelude is running on the CYBER 205 using special hardware instructions instead of this scalar code.

2. Actual state of the project

Up to now a compiler for ALGOL68C has been implemented. This programing language has been developed at the University of Cambridge and differs slightly from ALGOL68 (as defined in the *"Revised Report"*); but ALGOL68C contains a relevant subset of ALGOL68. Especially all necessary language concepts for the CYBER 205 are included. The compiler has been modified to generate vector instructions when copying rows. In addition a large prelude containing operations on vectors and matrices usefull in linear algebra has been developed. Besides others it includes the operators listed below.

- The basic arithmetic operations for data types ***int***, ***real*** and ***compl*** for vectors and matrices are provided :

  +, −, *, / (or % respectively)

  These operators take two vectors or two matrices or a vector and a scalar or a matrix and a scalar as their arguments and perform the given operation on each element. According to ALGOL68's principle of orthogonality any mixture of the three data types mentioned above is supported. For reason of efficiency the operators

  +<, −<, *<, /< (or %< respectively)

  have been added which combine an operation and an assignation in order to avoid useless copying in statements of the form a := a + b. Additionally the operators ***i*** and ***conj*** for computations with complex vectors and matrices have been implemented and an operator <> denoting the (mathematical) product of two matrices.

- In generalising the well known standard functions the following transforming operators have been added :

  exp, ***ln***, ***sqrt***, ***abs***, ***re***, ***im***, ***widen***, ***entier***, ***round***,
  sin, ***cos***, ***tan***, ...

  In order to obey the principle of orthogonality these operators have also been defined for scalar arguments.

- Operators on vectors and matrices having a scalar result and operators on matrices having a vector result which are often needed in mathematical formulas have been included :
  sum and ***product*** for the sum and the product of all elements of a vector, ***eq*** (equality of all components), <> denoting the scalar product of two vectors and the product of a vector and a matrix and additionally the operators ***min***, ***max***, ***minabs*** and ***maxabs***.

- To allow the use of special CYBER 205 hardware instructions a number of machine oriented operators have been defined :

  gathered, ***scattered***, ***intervalvector*** and ***value***.

  They may be used to gather or scatter components of a vector, to generate a vector of equidistant values ***start*** *+ (i - 1) ** ***step*** ($1 \le i \le n$) and to assign a scalar value to every component of a vector or a matrix.

- Some special triadic operations taken from linear algebra (vector ***plusab*** vector ***times*** scalar, vektor ***plusab*** scalar ***times*** vector and matrix ***plusab*** outer product of two vectors) have been added which use the LINK instruction of the CYBER 205.

- Three language extensions have been added for reason of convenience only :

  diag, ***codiag*** und ***transpose***.

  The result of these operators is a pointer to the main diagonal or the opposite diagonal of a square matrix or a pointer to the transposed matrix without actually copying the elements of the matrix.

3. Example

To illustrate the advantage of using operators from this prelude let us now look at a mathematical example, the calculation of a definite integral. Compute

$$\int_a^b f(x)dx$$

of a real valued function f(x) using Hermite's quadrature formula. Let

$$g(y) := \frac{b-a}{2} f\Big(\frac{b-a}{2}y + \frac{a+b}{2}\Big)\sqrt{1-y^2}\ .$$

Then according to Hermite's formula

$$\int_a^b f(x)dx = \int_{-1}^{1} g(y)\frac{1}{\sqrt{1-y^2}}\ dy \approx \frac{\pi}{n+1}\sum_{k=0}^{n} g\Big(\cos\Big(\frac{2k+1}{n+1}\frac{\pi}{2}\Big)\Big).$$

What does this algorithm look like in ALGOL68 notation? There are only two relevant formulas. The first is the transformation g :

$$g(y) := \frac{b-a}{2} f\Big(\frac{b-a}{2}y + \frac{a+b}{2}\Big)\sqrt{1-y^2}$$

This can be written (using the mode ***function*** explained later) as follows :

```
function g = ([| real y) [| real :
             (b - a)/2 * f ((b - a)/2 * y + (a + b)/2) * sqrt (1 - y ** 2)
```

The second formula gives the result of the computation :

$$\frac{\pi}{n+1}\sum_{k=0}^{n} g\Big(\cos\Big(\frac{2k+1}{n+1}\frac{\pi}{2}\Big)\Big) = c\sum_{k=0}^{n} g\Big(\cos\Big(\frac{c}{2}+kc\Big)\Big) \quad \text{where} \quad c = \frac{\pi}{n+1}$$

Using operators of the prelude this can be written as follows :

```
    real c = pi / (n + 1)
  ; c * sum g (cos (intervalvector (c/2 step c size n+1)))
```

And here is the complete program :

```
    mode function = proc ([] real) [] real

  ; proc hermite approximation = (function f, real a, b, int n) real :
    begin
        function g = ([] real y) [] real :
            (b - a)/2 * f ((b - a)/2 * y + (a + b)/2) * sqrt (1 - y ** 2)
      ; real c = pi / (n + 1)
      ; c * sum g (cos (intervalvector (c/2 step c size n+1)))
    end
```

It uses the newly declared mode ***function*** being a function with a real vector as its argument and returning a real vector as its result. No intermediate storage has to be declared by the user and no care has to be taken for long vectors — and the time used for computation is comparable to the time used by a similar FORTRAN program.

4. Availability

The implementation described above is running on the CYBER 205 at Bochum under VSOS 2.2.5 operating system.

References :

[1] K. Hackenberg :
ALGOL68 auf der CYBER 205
Bochumer Schriften zur Parallelen Datenverarbeitung 7 ,
Rechenzentrum der Ruhr-Universität Bochum 1985.

AB52.4.2 An Exception Handling proposal for ALGOL 68.

G. S. Tseytin.
Leningrad State University.

In my view, the need for a good mechanism for exception handling becomes imperative when we accept the concept of modules (packages). Every time I tell my students that a package must take care of testing data for validity I am embarrassed at giving examples: *if the test fails then what? fi*. I cannot suggest to them to print a message and then to stop, but just as well I don't want to suspect an error at every call to a package function.

I have developed my proposals for exception handling in the same form as Part II of the Modules proposal [AB43.3.2]. Informally, these consist of two parts, first, the concept of raising and handling exceptions, second, system-defined exceptions.

The first is based on the idea of "dynamic identification", i.e., obtaining data defined in an environ of a call within the routine called. There must be a "primary" defining occurrence of a "dynamic" identifier, and then it can be "redefined" in newer environs with a reference to the same "primary" declaration. An applied occurrence of such an identifier "statically" identifies the primary defining occurrence but yields the value from the newest redefinition. We have been long experimenting with this idea. Indeed, a version of it was about 10 years ago implemented in our instrumental compiler used to bootstrap the working implementation.

So I had to introduce three new constructs. An **exception-declaration** provides the primary defining occurrence of an exception and specifies the mode of the handling routine for this exception. A **handler-declaration** "redefines" the exception by providing the actual handling routine. An **exception-call** looks like an ordinary **call** except for a special rule for fetching the routine.

It may look like this.

```
module stacks =
    def ...
    ......
    exception (stack) void stack underflow;
    ......
    pub proc pop = (stack s) int:
        ( ..... raise stack underflow (s) ...);
    ......
    on stack underflow: (stack s) void:
        (print("alas!"); stop);
    ......
    fed;
......
access stacks (
    ......
    on stack underflow: (stack s) void:
        ( ..... go to underflow; ...);
    underflow: ....
    ......
    pop(...)
    ......
    )
```

In this example an underflow detected during the call of *pop* will not result in printing *alas!* but most likely in jumping to *underflow*.

An **exception-call** is allowed to yield a value, supposed that the programmer knows what to do with it. Thus raising an exception does not necessarily imply a jump to an outer environ. So this mechanism can be used, e.g., by a *stacks* module to find out what size of stack the user wants. It seems to be very inefficient and certainly un-algol-like to use this sort of mechanism systematically but I think it is all right to use it as an exception.

It would seem to be more orthogonal to allow to connect with exceptions all kinds of values, not only routines. Then the exception mechanism described above would serve only to fetch the routine, which could then be called by the regular calling mechanism. However I did not generalize it that far; the call caused by an exception must be different from an ordinary call in order to make it possible to define propagation of exceptions. There is no explicit *reraise*; its part is played by an ordinary *raise*. Raising the same exception while elaborating an **exception-call** has the effect of starting the search for a handler routine from the environ statically containing the **handler-declaration** used for the first time the exception was raised.

Handler-declarations are syntactically **declarations** though they have no applied occurrences, They could instead have been **units** that could be elaborated several times within the same environ. But I thought the first solution safer.

The intended implementation of this sort of dynamic identification is that a table is built at compilation time to connect each **handler-declaration** with the portion of object code where this **declaration** can be used, i.e., from the semicolon following the handler-declaration to the end of the range. Then any **exception-call** can be resolved either at compile time or at run time by examining the return address for the current routine. (In our instrumental compiler we used a different method.)

The second part of my proposal, the list of system-defined exceptions, is somewhat less secure because it has to make allowance for peculiarities of local operating systems. So my specifications for such exceptions have been made as unobliging as possible. Completion of a system-produced **exception-call** results in terminating the program or, more precisely, raising the *terminate* exception. Usually an operating system allowing to set a trap for an exception in some 'environ' destroys all inner 'environs' when the exception is actually trapped, so it is impossible to return to the place where the exception occurred. Under my definitions the only way to leave a handler routine without completing it is to jump out of it, either directly or by raising another exception. So it is necessary to set traps for all system exceptions in every environ containing any **handler-declaration**.

However I have allowed for some system-defined (or, better, implementation-defined) exceptions not to follow the above scheme. The following exceptions allow the program to continue after successful completion of the **exception-call**:

a) all exceptions caused by transput events (they are all parallel to the existing transput routines);

b) the exception caused by a wrong value of a bound in a **slice**; in this case the handler routine may provide a new value for the bound.

Handlers for some exceptions asscoiated with multiple values can obtain additional information by raising special exceptions (e.g., a handler can find out whether the offending name is a subname of some given name).

Here is the formal definition in the form of amendments to RR. The English language version was developed parallel to the Russian language version (with Russian paranotions, etc.); only the Russian version will become part of our standard extensions to ALGOL 68, alongside a Russian translation of the Modules and Separate Compilation proposals from AB43.3.2. Some possible alternatives to the decisions presented here (including a fully different and much weaker system) have been discussed as well. At the last meeting our WG insisted on extending the list of standard exceptions, even at the price of making the conventions less obliging.

Formal Definition of Exception-handling extension.

1.1.4.4. Recovery actions

a) For some cases where elaboration is said to be undefined {1.1.4.3.a,b} recovery actions are specified. This means that such recovery action is to be taken unless the implementer has provided a more satisfactory solution for this situation. However the implementer must preserve for the programmer a way to require that the action taken be exactly the recovery action specified here.

{The recovery action usually includes raising an appropriate exception.}

b) A recovery action consists in a calling of some routine, possibly with {parameter} values. The routine is specified by means of a representation of an **applied-identifier** yielding this routine in the environ of the **particular-prelude**.

1.2.3.

E) DEC :: ... ; **PROCEDURE EXCEPTION TAG.**
V) **EXCEPTION** :: **exception** ; **handler.**

2.1.2.

h) "To handle" is a relationship between a value {a routine} and a scene {an exception definition} which may hold "inside" a specified locale. This relationship is made to hold upon the elaboration of a **handler-definition**.

i) An environ can be "connected" to another environ {with older scope} "by means of" a scene {an exception definition}. This relationship may hold for an environ created at the time of the elaboration of an **exception-call**.

2.1.4.3.

h) .... {For some events recovery actions are defined, see 1.1.4.4.}

i) An action may be interrupted if the computer discovers that time (memory space) is nearly exhausted. The recovery action for such a case is a calling of the routine *time exhaustion recovery* {10.2.5.p} (*space exhaustion recovery* {10.2.5.q}). {It is expected that the remaining amount of time (space) will be sufficient for the recovery action to secure an orderly termination or to obtain additional resources.}

3.2.2.

a) ...., the recovery action being a calling of the routine *scope error recovery* {10.2.5.m}.

b) {{Replace the last lines of "Case A"}}
For each constituent **exception-definition** X, if any, of C,
• the scene composed of
(i) X, and
(ii) the environ E,
is ascribed in E to the **exception-identifier** of X;
If each **'PROP'** enveloped by **'PROPSETY'** is some **'DYADIC TAD'**, or **'label TAG'** or **'PROCEDURE exception TAG'**,
then E is said to be "nonlocal" {see 5.2.3.2.b};

3.3.2.

b) {{Append to "Case C"}}
If not all of the descriptors of V_1, ... , V_m are identical, the recovery action is as follows:
• let U be some multiple value of the mode specified by *[] rows* {see 10.2.3.1.a} with the descriptor ((1,m)) and such that, for i = 1, ... , m, the element selected by the index (i) is some multiple value with a descriptor identical to that of V_i;
• the routine *display error recovery* {10.2.5.k} is called with the {parameter} values U and n, where n is the number of pairs in the descriptor of {say} V_1.

4.1.1.

A) **COMMON :: ... ; EXCEPTION.**

4.8.1.

F) **QUALITY :: ... ; PROCEDURE EXCEPTION.**

4.10. Exceptions and handlers

{An exception is a condition, discovered by implementation or by the user program, that requires some action depending on the current environ. The action has the form of calling some routine. An **exception-definition** introduces a new kind of exception and specifies the mode of the routine to be called for this exception. A **handler-definition** specifies the particular routine to be used for this exception during the lifetime of the current environ, except for derived environs that may specify their own routines for the same exception. The **handler-defining-identifier** of a **handler-definition** is treated as an **exception-applied-identifier** that must identify the **exception-defining-identifier** of the respective **exception-definition**. There are no **handler-applied-identifiers**.}

4.10.1. Syntax

a) **NEST EXCEPTION declaration of DECS**{41a} :
EXCEPTION{94f} **token,**
NEST EXCEPTION joined definition of DECS{41b,c}.

b) **NEST exception definition of PROCEDURE exception TAG**{41c} :
formal PROCEDURE NEST plan{46p},
PROCEDURE exception NEST defining identifier with TAG{48a}.

c) **NEST handler definition of PROCEDURE handler TAG**{41c} :
where PROCEDURE exception TAG identified in NEST{72a},
PROCEDURE handler NEST defining identifier with TAG{48a},
colon{94f} **token, PROCEDURE NEST source**{521c}.

{Examples:
a) *exception (real) void invalid argument* •
on invalid argument: (real x) void : finish
b) *(real) void invalid argument*
c) *invalid argument: (real x) void : finish* }

4.10.2. Semantics

a) The elaboration of an **exception-declaration** {involves no action, yields no value and} is completed.

b) A **handler-declaration** D in an environ E is elaborated as follows:

• the constituent sources of D are elaborated in E collaterally;
For each constituent **handler-definition** D1 of D
• let V be the yield of the source of D1;
• let X {a scene composed from an **exception-definition**} be the yield of an **exception-applied-identifier** akin to the **handler-defining-identifier** of D1, in E;
• V is made to handle X inside the locale of E.

5.1.

D) **PRIMARY :: ... ; exception call**{545a} **coercee.**

5.2.1.2.b

b) {{insert before "Case A"}}
If N is nil the recovery action is a calling of *nil error recovery* {10.2.5.l};
If W is newer is scope that N the recovery action is a calling of *scope error recovery* {10.2.5.m};

{{append to "Case B"}}
If the descriptors of W and V are not identical the recovery action is as follows:
• let n be the number of pairs in the descriptor of W;
• let i be some integer such that $1 \leq i \leq n$ and the i-th pairs in the descriptors of W and V are not identical;
• the routine *assignment error recovery* {10.2.5.h} is called with {parameter} values N, W, n, i;

5.3.1.2.

{{replace $\bullet_2$ by}}
• it is required that V {if it is a name} be not nil, the recovery action being a calling of *nil error recovery* {10.2.5.l};

5.3.2.2.

a) {{replace by}}
a) The yield W of a slice S is determined in the following steps:

Step 1:
- let V and (I_1, ... , I_n) be the {collateral} yields of the **PRIMARY** of S and of the **indexer** {b} of S;
- it is required that V {if it is a name} be not nil, the recovery action being a calling of *nil error recovery* {10.2.5.1};
- let ((r_1,s_1), ... , (r_n,s_n)) be the descriptor of V or of the value referred to by V;

Step 2: For i = 1, ... , n
- Case A: I_i is an integer:
  - it is required that $r_i \le I_i \le s_i$;
- Case B: I_i is some triplet (l,u,l'):
  - let L be r_i if l is absent, and be l otherwise;
  - let U be s_i if u is absent, and be u otherwise;
  - it is required that $r_i \le L$ and $U \le s_i$;
  - I_i is replaced by (L,U,l');
- The recovery action for this step is as follows:
  - let i and C be some numbers such that $1 \le i \le n$ and either
    Case A1: I_i is an integer, $I_i < r_i$ or $I_i > s_i$, and C = I_i; or
    Case B1: I_i is some triplet (l,u,l'), as possibly modified by previous steps, l is not absent, l < r_i and C = l; or
    Case B2: I_i is some triplet (l,u,l'), as possibly modified by previous steps, u is not absent, u > s_i and C = u;
  - let R be the routine *name bound error recovery* {10.2.5.j}, if V is a name, and the routine *bound error recovery* {10.2.5.i} otherwise;
  - let C' be the yield of calling R with {parameter} values V, n, i and C;
  - for the case A1: I_i is replaced by C';
  - for the case B1: l of I_i is replaced by C';
  - for the case B2: u of I_i is replaced by C';
  - Step 2 is taken again;

Step 3: For i = 1, ... , n,
- If I_i is some triplet (l,u,l'),
  - let D be 0 if l' is absent, and be l-l' otherwise;
  {D is the amount to be substracted from l in order to get the revised lower bound;}
  - l' is replaced by D;

Spep 4:
- W is the value selected in {2.1.3.4.a,g,i} or the name generated from {2.1.3.4.j} V by (I_1, ... , I_n).

5.4.5. Exception calls

{An **exception-call** serves to raise an exception and thus to call a routine assigned to handle this exception in the current environ. An **exception-call** may supply parameters for this routine. The handling routine is searched for, starting from the current environ throughout the environs with older scopes except for the case when during the elaboration of an **exception-call** the same exception is raised again. In the latter case the inner **exception-call** does not use the handler found for the outer **exception-call**, and the search for the handler continues from the environ with the next older scope than that of the environ with the locale containing the first handler. In some programming languages a similar process is termed exception propagation.}

5.4.5.1. Syntax

a) **MOID NEST exception call**{5D} :
    raise token{94f} **option**,
        procedure PARAMETY yielding MOID exception applied identifier with TAG{48b},
        NEST parametrization PARAMETY{b,c}.

b) **NEST parametrization with PARAMETERS**{a} :
    actual NEST PARAMETERS{543b,c} **brief pack**.

c) **NEST parametrization**{a} : **EMPTY**.

{Examples:
  a) *<u>raise</u> invalid argument (x)* }

5.4.5.2. Semantics

a) The yield W of an **exception-call** Y, in an environ E, is determined as follows:
- let X {a scene composed from an **exception-definition**} be the yield of the **exception-applied-identifier** of Y, in E;
- let H and F be, respectively, the handler and the handling environ {b} for X in E;
- let E1 be the new environ established {locally, see 3.2.2.b} around E; E1 is said to be connected to F by means of X;
- let V_1, ... , V_n be the {collateral} yields of the constituent **actual-parameters** of Y, if any, in E1;
- W is the yield of the calling {5.4.3.2.b} of H in E1, possibly with V_1, ... , V_n;
- it is required that W be not newer in scope than E, the recovery action being a calling of *scope error recovery* {10.2.5.m}.

b) The handler H and the handling environ F for a scene X in an environ E are determined as follows:
- it is required that E be not older in scope that the environ of X {for, otherwise, no handler can be found}, the recovery action being a calling of *general exception recovery* {10.2.5.o};

If there is a value R which handles X inside the locale of E
then H is R and F is E;
otherwise,
- let E1 be the reference environ {c} for X in E;
- let E2 be the environ upon which E1 is established {3.2.2.b};
- H and F are the handler and the handling environ for X in E2.

c) The reference environ F for a scene X in an environ E is determined as follows:
If E is connected by means of X to another environ E1
then F is E1;
otherwise, F is E.

6.1.1.

F) **MORF :: ... ; NEST exception call.**

6.2.2.

{{replace $\cdot_2$ by}}
- it is required that N be not nil, the recovery action being a calling of *nil error recovery* {10.2.5.1};

7.1.1.

```
c) WHETHER QUALITY1 TAX1 independent QUALITY2 TAX2{a,48a,c,72a} :
        ... ;
        where (TAX1) is (TAX2) and (TAX1) is (TAG),
             where (QUALITY1) is (PROCEDURE1 EXCEPTION1) and
                  (QUALITY2) is (PROCEDURE2 EXCEPTION2),
             WHETHER (EXCEPTION1 EXCEPTION2) is (exception handler) or
                  (EXCEPTION1 EXCEPTION2) is (handler exception).
```

7.2.1.

```
c) WHETHER QUALITY1 TAX resides in QUALITY2 TAX{a,b,48d} :
        ... ;
        where (QUALITY) is (PROCEDURE1 EXCEPTION) and (QUALITY2) is
        (PROCEDURE2 EXCEPTION),
             WHETHER PROCEDURE1 equivalent PROCEDURE2{73a}.
```

9.4.1.

f) {{append}}

exception symbol	*exception*
handler symbol	*on*
raise symbol	*raise*

10.2.1.

v) proc *L int overflow enabled* = bool : c true, *if at a condition for which the recovery action is specified as a call of the routine 'L int overflow recovery'* {10.2.3.13}, *the implementation actually takes this action;* false *otherwise* c ;

{{Similarly for *L real overflow enabled*, *L real underflow enabled*, *L int argument error enabled*, *L real argument error enabled.*}}

w) {{Similarly for *assignment error enabled* {5.2.1.2.b}, *bound error enabled* {5.3.2.2.a}, *row display error enabled* {3.3.2.b}, *nil error enabled* {5.2.1.2.b, 5.3.1.2, 5.3.2.2.a, 6.2.2}, *scope error enabled* {3.2.2.a, 5.2.1.2.b, 5.4.5.2.a}, *deadlock enabled* {10.2.4.d}, *time exhaustion enabled* {2.1.4.3.i}, *space exhaustion enabled* {2.1.4.3.i}. }}

10.2.3.13. Recovery actions for standard operators and functions

For cases when operators and functions of the section 10.2.3 do not give a meaningful result recovery actions are defined as callings of routines from 10.2.5.g.

The routine *L int overflow recovery* (*L real overflow recovery*) is called on a failure when it is expected that a similar computation could be successful in another implementation with a greater value of *L max int* (*L max real*).

The routine *L underflow recovery* is called on a failure caused by the yield (of the mode *L* real) of an operator or function being too small to be represented (within the accuracy implied by the value of *L small real*) by the underlying hardware.

The routine *L int argument error recovery* (*L real argument error recovery*) is called on a failure with a {parameter} value X of the mode L int (*L* real) when this value has been used as the value of an **actual-parameter** or **operand** amd the failure is due to the fact that the result is not mathematically defined for this value.

10.2.4.

d) {{extend the pseudo-comment}}

; if all processes descended from the **particular-program** *have been so halted and none of then is resumed the further elaboration is undefined, with the recovery action defined as a calling of 'deadlock recovery'* {10.2.5.n} *with the yield of 'edsger'* {as a parameter value}

10.2.5. Standard exceptions and recovery routines

```
a) exception void L int overflow,
        void L real overflow,
        void L underflow,
        (L int) void L int argument error,
        (L real) void L real argument error ;

b) exception (int) void assignment error,
        (int, ref int) void bound error,
        void row display error,
        void nil error,
        void scope error,
        (sema) void deadlock,
        void general exception ;
```

{*general exception* is raised when no handler is found for some raised exception.}

```
c) exception void time exhaustion,
        void space exhaustion,
        void termination,
        void ? immediate termination ;
                  {see 10.5.1.k}
```

{The *termination* exception is raised in cases of irrecoverable errors in order to give the programmer an opportunity to secure necessary closedown actions by defining a handler for this exception in some environs. But it is expected that the elaboration of the handling routine will end with raising the same exception again in order to secure closedown actions in older environs.}

d) mode *?* refrows = c *an actual-declarer specifying a mode united from* {2.1.3.6.a} *a sufficient set of modes each of which begins with 'reference to row' or 'reference to flexible row'* c ;

```
e) exception rows row specimen,
        rows destination specimen,
        [] rows row specimen list,
        (refrows) bool is slice of,
        int dimension, bool isname ;
```

{These exceptions are used in some recovery routines for other exceptions.}

```
f) proc terminate = void :
        (c some system action helping to identify the current environ c;
        raise termination;
        raise immediate termination) ;
```

```
g)  proc ? L int overflow recovery = void :
         (raise L int overflow; terminate),
    ? L real overflow recovery = void :
         (raise L real overflow; terminate),
    ? L underflow recovery = void :
         (raise L underflow; terminate),
    ? L int argument error recovery = (L int i) void :
         (raise L int argument error (i); terminate),
    ? L real argument error recovery = (L real r) void :
         (raise L real argument error (r); terminate) ;

h)  proc ? assignment error recovery =
    (refrows destination, rows source, int n, i) void :
         (on destination specimen: rows :
                   c some multiple value with a descriptor identical
                   to that of the value referred to by the name
                   yielded by 'destination' c
              row specimen: rows :
                   c some multiple value with a descriptor identical
                   to that of the value yielded by 'source' c,
              is slice of: (refrows rr) bool :
                   if c the name yielded by 'rr' has not been
                        generated (2.1.3.4.j,l) from another name c
                   then c true, if every subname of the name yielded
                        by 'destination' is a subname of the name
                        yielded by 'rr', or can be obtained from such
                        subname by one of more selections by 'TAG'
                        (2.1.3.3.e);
                        false otherwise c
                   else skip
                   fi,
              dimension: int : n;
         raise assignment error (i);
         terminate) ;

i)  proc ? bound error recovery =
    (rows value, int n, i, bound) int :
         (on row specimen: rows :
                   c some multiple value with a descriptor identical
                   to that of the yield of 'value' c,
              is slice of: (refrows rr) bool : skip,
              dimension: int : n,
              isname: bool : false;
         int b := bound;
         raise bound error (i, b);
         b) ;

j)  proc ? name bound error recovery =
    (refrows name, int n, i, bound) int :
         (on row specimen: rows :
                   c some multiple value with a descriptor identical
                   to that of the value referred to by the name
                   yielded by 'name' c,
              is slice of: (refrows rr) bool :
                   if c the name yielded by 'rr' has not been
                        generated (2.1.3.4.j,l) from another name c
                   then c true, if every subname of the name yielded
                        by 'name' is a subname of the name yielded by
                        'rr', or can be obtained from such subname by
                        one or more selections by 'TAG' (2.1.3.3.e);
                        false otherwise c
                   else skip
                   fi,
              dimension: int : n,
              isname: bool : true;
         int b := bound;
         raise bound error (i, b);
         b) ;

k)  proc ? row display error recovery =
         ([] rows specimen, int n) void :
              (on row specimen list: [] rows : specimen,
                   dimension: int : n;
              raise row display error;
              terminate) ;

l)  proc ? nil error recovery = void :
         (raise nil error; terminate) ;

m)  proc ? scope error recovery = void :
         (raise scope error; terminate) ;

n)  proc ? deadlock recovery = (sema s) void :
         (raise deadlock (s); terminate) ;

o)  exception bool ? general exception recursion ;
         {see 10.5.1.1}
    proc ? general exception recovery = void :
         if general exception recursion
         then raise immediate termination
         else on general exception recursion: bool : true;
              terminate
         fi ;

p)  proc ? time exhaustion recovery = void :
         (raise time exhaustion; terminate) ;

q)  proc ? space exhaustion recovery = void :
         (raise space exhaustion; terminate) ;
```

10.3.1.3.

{{A replacement for part of a sentence in 10.3.1.3.cc}}
If the event routine returns false another attempt is taken to recover by raising the corresponding exception. This results in calling another routine with a function similar to that of the event routine but attached to the current environ rather than to the current file. If this routine returns false too, then the system continues with its default action.

```
t)  exception (ref file) bool logical file end,
         (ref file) bool physical file end,
         (ref file) bool page end,
         (ref file) bool line end,
         (ref file) bool format end,
         (ref file) bool value error,
         (ref file, ref char) bool char error ;
```

```
u)  proc ? logical file end repaired = (ref file f) bool :
            if (logical file mended of f) (f)
            then true
            else raise logical file end (f)
            fi,
        ? physical file end repaired = (ref file f) bool :
            if (physical file mended of f) (f)
            then true
            else raise physical file end (f)
            fi,
        ? page end repaired = (ref file f) bool :
            if (page mended of f) (f)
            then true
            else raise page end (f)
            fi,
        ? line end repaired = (ref file f) bool :
            if (line mended of f) (f)
            then true
            else raise line end (f)
            fi,
        ? format end repaired = (ref file f) bool :
            if (format mended of f) (f)
            then true
            else raise format end (f)
            fi,
        ? value error repaired = (ref file f) bool :
            if (value error mended of f) (f)
            then true
            else raise value error (f)
            fi,
        ? char error repaired = (ref file f, ref char c) bool :
            if (char error mended of f) (f, c)
            then true
            else raise char error (f, c)
            fi ;
```

{{A replacement for the appropriate sentence in 10.3.1.6.dd}}
... The routine *logical file end repaired*, *physical file end repaired*, *page end repaired* or *line end repaired* is therefore called as appropriate. ...

{{A replacement for the appropriate phrases in 10.3.3 and 10.3.3.2.hh.(ii)}}
... the routine *line end repaired* (or, where appropriate, *page end repaired*, *physical file end repaired* or *logical file end repaired*) ...

{{Similarly in 10.3.3.2, for the routines *char error repaired* (cc, dd), *line end repaired*, *page end repaired*, *physical file end repaired*, *logical file end repaired* (all in hh); also in 10.3.4.1.1, for the routines *format end repaired* (gg), *value error repaired* (hh, ii), *char error repaired* (ll); also for the routine *value error repaired* in 10.3.4.8.1.aa,bb,dd,ee and 10.3.4.10.1.aa}}

{{Throughout all the forms contained in section 10.3 the following changes are to be made:

```
    (logical file mended of f) → logical file end repaired
    (physical file mended of f) → physical file end repaired
    (page mended of f) → page end repaired
    (line mended of f) → line end repaired
    (format mended of f) → format end repaired
    (value error mended of f) → value error repaired
    (char error mended of f) → char error repaired
```
}}

10.5.1.

```
j)  on termination: void : stop ;

k)  on immediate termination: void : stop ;

l)  on general exception recursion: bool : false ;

m)  on logical file end: (ref file f) bool : false,
       physical file end: (ref file f) bool : false,
       page end: (ref file f) bool : false,
       line end: (ref file f) bool : false,
       format end: (ref file f) bool : false,
       value error: (ref file f) bool : false,
       char error: (ref file f, ref char c) bool : false ;
```

AB52.4.3

A Browse through some Early Bulletins.

by C. H. Lindsey

(University of Manchester)

After IFIP WG2.1 had been formed (initially from the original authors of ALGOL 60) a decision was taken in March 1964 to revive the ALGOL Bulletin, which had lain dormant since the publication of the Revised Report on the Algorithmic Language ALGOL 60 in 1962. Fraser Duncan was appointed as Editor, and AB16 duly appeared in May 1964. As present editor of the AB, I have no access to any issues prior to AB16, but I have managed to piece together a complete set since that date, and they form a fascinating account of what was going on in those years. The following article surveys some of the material published between 1964 and 1972.

ALGOL 60

Of course, ALGOL 60 was not frozen with the publication of the Revised Report in 1962. Much remained to be done as regards subsets, I-O, problems in the Report, and in just trying to understand the beast that had been created. It should be realised that many features in ALGOL 60 seem to have "just happened" and their ramifications (even their implementations) only became apparent as time went on. As John McCarthy said on one occasion, the authors of the original Report were all gentlemen, and did not propose any feature for inclusion that they did not see how to implement sensibly. It was the interactions between the various features which were not so well understood at the time.

Block Structure and Environments.

One of the novel features of ALGOL 60 was, of course, block structure, and the idea that procedures lived in definite environments. This was reasonably well understood by some people by 1964, but not always by implementors. Knuth's famous test case "Man or boy?" appeared in [AB17.2.4] (July 1964 – note how close together issues were in those days). Originally, of course, it was written in ALGOL 60 and used call-by-name, but here it is in ALGOL 68.

```
BEGIN
    PROC a = (INT k, PROC INT x1, x2, x3, x4, x5) INT:
      BEGIN
          LOC INT kk := k;
          PROC b = INT:
            BEGIN
              kk -:= 1;
              a(kk, b, x1, x2, x3, x4)
            END;
          IF kk<=0
          THEN x4 + x5
          ELSE b
          FI
      END;
    print( a(10, INT: 1, INT: -1, INT: -1, INT: 1, INT: 0)
END
```

The point about this program, of course, is that many incarnations of *b* are created in many environments, each of which is able to decrement the particular *kk* in the environment where is was created. Readers are advised NOT to try computing the result by hand. Knuth tried and obtained the result *-121*, which is wrong (at the time, he had broken his right wrist so, as he said [AB19.2.3.4], the calculations were done left handed – but he did then give a formula from which the corect result of any example could be calculated). No wonder he got it wrong! This case recurses

to a depth of 512 in *a* and 511 in *b*. The correct results for various values of *k*, computed on the Electrologica X1 at the Mathematisch Centrum appeared in [AB18.2.5].

k	0	1	2	3	4	5	6	7	8	9	(10	11)
A	1	0	-2	0	1	0	1	-1	-10	-30	(-67	-138)

The X1 actually ran out of its memory (12K of 27-bit words) on *k=10* (memory doubles for each increment of *k*) so the result for *k=10* was from a machine at Kiel and for *k=11* on a KDF9 using the Kidsgrove compiler, which took 12 seconds [AB19.2.3.1]. Other times reported were 1.75 seconds on the ICL Atlas (*k=11*) and times of 20 and 80 <u>minutes</u> on two other machines whose anonymity the Editor agreed to respect [AB20.2.4]. By way of comparison I just ran a PASCAL version of *k=11* on a SUN3, and it took 0.027 seconds.

As a postcript to this whole episode, it may be noted that a couple of years later Bekic was able to use this example to persuade the designers of PL/1 (who really did not understand environments) to mend their ways.

As an aid to understanding those environments, a piece of Science Fiction appeared in [AB17.2.5], written by W. H. Burge, Manager of Systems Programming Research at the Univac Division of Sperry Rand, New York. I think it is worth reproducing it in full.

THE ALGOL MEN

They didn't know where they came from when they were born, they called it "Outside". They did not know when a new person would appear, but noticed that they only appeared when space was available. All people were born with the same amount of experience and property which they called an "environment". When they died, they disappeared, and so did their environment.

There was a ritual called "enter block" which they could perform in order to use a new body. They stored what was left of the old body and its environment secretly, so that no one could interfere with it, and lived their life in a new body. The new body came with a certain amount of experience and property called its "locals". This was added to the environment of the old body to produce an environment for life in the new body.

When an incarnate body died, there was another ritual called "block exit" in which the old body was disinterred, and life continued in this body where it had left off. The incarnated body disappeared together with its locals.

The process of incarnation was difficult and required much concentration. There were many people who were born and died without achieving it. These were called "FORTRANS" or "COBOLS". Some of these were compensated for this by having other abilities, for instance, the FORTRANS could run fast and the COBOLS could describe data by a carefully guarded technique only known to them.

Some people who were lucky found that among their possessions in their environment were bodies called "procedure bodies". They could live in these at will by using the "enter block" ritual. It was possible to pass property on to be used in their lives as procedure bodies. This property was called an "argument".

It required considerable training to keep track of all the interred bodies, and to choose the correct one to reincarnate by the block exit ritual. This was made even more difficult by the existence of so-called "recursive" procedure bodies or Doppelgängers. These bodies found themselves on their own environment. It was clear that when a body incarnated itself, it could not both bury and use its body (the technique of copying bodies was known but its use was deprecated because it wasted space) and so instead of burying the body, they buried information which said where it was.

When a person got tired of his present life and hankered after one of his previous incarnations, he could invoke a "go to" ritual which would take him back to one of his previously stored bodies. This would disinter and reactivate a body which was not necessarily the last one used. This process invariably lost all the intervening bodies. This go to ritual occasionally surprised people. They would enter a procedure body in order to gain something of value, and find themselves projected by a go to into a previous incarnation, instead.

The ALGOL Men were not only able to change their immediate environment or locals but could also change the environments of their interred bodies. Some people thought it was sacrilege to interfere with the possessions of their interred bodies (or non locals), especially when living in procedure bodies. Others thought this practice useful, and used it.

It seemed that the course of a person's life was mapped out for him in advance, although they couldn't tell for sure. Some said that they must do the things they did in a fixed order, others said that the order was not fixed. Occasionally some of the latter found that changing the order made them do things they didn't intend to. Others argued that they had free will because they could make decisions. Although they tried hard, they could never manufacture a new body. The bodies and environments which appeared seemed to have been lying dormant in their genes, waiting to be activated.

The ALGOL Men had a great book which contained guidance called the "[Revised] Report on the Algorithmic Language ALGOL 60". This contained rules and parables which stated how they should behave, and commandments which prohibited certain behaviour. The interpretation of certain sections of this Report was the subject of much scholarly and theological argument. Part of the Report was written in a peculiar language called "Backus normal form". No one had ever deciphered this part, but it seemed to be explaining how their bodies were constructed. The Report prescribed that if certain rules of behaviour were disobeyed then the offender would be put into a state called "undefined" or "hell". There were legends about certain adventurous spirits who, because they possessed some defect of character, had tried these prescribed acts and as a result had vanished together with all their interred bodies and possessions.

People who were deep in incarnations used a lot of space to hold their bodies and possessions and as a consequence people were taxed on how deep they were. Sometimes people would wish to incarnate but find they could not because there was no space left on the world. They had to go into a state of "suspended animation" waiting for space to become available.

There were some people who lived the same parts of their life over and over again. Some of these were harmless and were called "ghosts". There seemed to be no way of finding out whether they would ever break out of their loop or not. Others were dangerous to the community and were called "space thieves". For these each new life cycle produced new possessions and these possessions threatened to overrun the world. A rule was introduced to detect space thieves and throw them Outside. This was called a "debugging rule". There was a similar rule against "body snatching". This rule said that if any person interfered with the interred bodies or property of another then both would be thrown Outside. A person was said to be "running wild" when he did this. This was unfortunate for the victim but his removal was a safety precaution. Since he had been interfered with, there was the chance that he too might run wild. Later, special locks were provided for the graves called "lockouts". These locks could only be opened by the owner. This prevented the whole population from running wild.

A movement sprang up to hoard property in a body's life so that it should be available if that life were to be re-entered. This hoarded property was called the "own" property of the body. There was some confusion about the precise procedure to be follwed when hoarding and there was little guidance in the Report about how to do

this. Some people, because they used incorrect methods, produced hoards which they never used again.

There were tales that certain people, called "mystics", were able to obtain information from Outside to guide their lives, and that there were poeple who could transmit information to the Outside by prayer. The Report, however, does not mention this. Some people found themselves incarnated into another form when they used what are called "non-ALGOL code" procedure bodies. Although this is mentioned in the Report, no details are given, and people were constantly surprised by the forms they took.

It is a shame that this brief account cannot include descriptions of other tribes similar to the ALGOL Men such as the Macro People who constructed bodies (or Frankensteins) and activated them, the Ipulvees who listed their property and had an oracle called the Interpreter, and the Lisps who constructed magnificent structures and took them to pieces, and had a strong garbage collectors' union.

The tragedy of the ALGOL Men was that they could not communicate with one another nor could they store things which would be useful to future generations. All men were born equal but did different things with their lives. When a man died he vanished, left nothing, and released the space he occupied. It is clear that what these people needed was the ability to store things produced by people in their lives in a library so that other people could make use of them. It is said that the FORTRANS and COBOLS have a tradition of this sort.

ALGOL 60 Developments.

The two great issues with regard to ALGOL 60 were Subsets, and I-O.

The "Report on SUBSET ALGOL 60 (IFIP)" appeared in AB16.3.1.1, complete with approval from the Coucil of IFIP and all the usual trimmings including permission to reproduce, but only in full. The Subset was in fact a dramatic piece of surgery, and a less drastic subset was produced by ECMA, the European Computer Manufacturers' Association (historically, the ECMA subset actually came before the IFIP one, however IFIP < ECMA < full ALGOL 60). For the ECMA subset, and a discussion of the whole issue, see [AB20.2.5] (July 1965).

Firstly, here are the restrictions common to both subsets:
- disappearance of *own*
- disappearance of integer labels
- <formal parameter>s to have their types specified
- types of all expressions to be compile-time determinable
- no lower-case letters
- *go to* an undefined <switch designator> to be undefined, not a dummy

Those all seem quite innocuous, but the next two will be surprising to modern readers, and reflect the fact that many compilers written up to that time had taken these shortcuts.
- no recursion
- only the first 6 characters of an identifier to be significant

Secondly, here are the additional restrictions in the IFIP subset. Some of these were made for reasons of tidiness rather than difficulty of implementation.
- an identifier may not appear twice in a <formal parameter list>
- no *for* *a[i]* := ... *do* ...
- no *go to* *if* *B* *then* *L1* *else* *L2* (and similarly in <switch-list>s)
- no *switch* *s* := *L1*, *t[i]*, ... (where *t* is another switch)
- no raising of *integers* to -ve powers
- no integer division
- actual-parameters called by name to be variables (but, not only does this prevent *jensen(i, i↑2)*, it also prevents *read(a[i])*)
- functions to have no side effects

Note the passion people had in those days for efficient implementation of extraordinary *go to*-like constructs which would never even be admitted into a modern language. Note also that many of these restrictions were rejected by ECMA because they eliminated more than they strictly needed to.

The "Report on Input-Output Procedures for ALGOL 60" appeared at the same time [AB16.3.1.2]. Its provisions were extremely primitive, but were only intended as a basis on top of which libraries of more useful facilities could be constructed. The Report blithely concluded by saying that "WG2.1 does not propose any further means for input-output operations". In the meantime, a separate proposal for Input-Output Conventions in ALGOL 60 had been prepared by an ACM committee chaired by Donald Knuth and published in CACM 7 (1964). This was at the opposite extreme, being format-based with every possible bell and every possible whistle (anyone wanting to know whence the formatted transput in ALGOL 68 originated need look no further). Of course this lead to complaints and counter proposals, including pleas by Naur [AB19.3.11.2, AB20.3.2.1] and Garwick [AB19.3.8] to separate number conversion from the actual I-O and a full-blown 15-page alternative scheme from the IBM Vienna Lab [AB20.3.5], and a mere 11 pages from ECMA [AB27.3.1]. A common factor of most of these proposals, judging by some of the comments made, seems to have been a lack of provision for recovery from input errors.

The ALGOL 60 Standard.

There now commenced that struggle to get ALGOL 60 adopted as an ISO Standard. It all started in fine style in May 1964 [AB17.1.1] with a report that ISO/TC97/SC5 had decided to "proceed immediately with ... an ISO Draft Proposal". This was to include the Report, the IFIP Subset, both the IFIP and the Knuth I-O, and a hardware representation. All very straightforward and timely. Of course, we now know with hindsight that this process was not finally completed until 20 years later. By October 1964 a First Draft was being circulated [AB18.1.3] – and the ECMA subset had crept in. After further meetings in September and October 1965 [AB22.2.1], it was all ready for final approval – bar the hardware representation. After that – silence! What happened? Apparently, there was a further draft in April 1967 which was finally approved in April 1968. Next, the text got lost in the ISO system. And then the text got mangled by ISO bureaucrats who didn't understand what they were doing, and it was finally published, complete with omissions and errors, in March 1972. There followed much argument between WG2.1 and ISO which culminated in the withdrawal of the document in 1976. After that, a mere 8 years to get another Standard (now based on the Modified ALGOL 60 Report) through the system is neither here nor there.

ALGOL 60 Trouble Spots.

Of course there were always worries about what the ALGOL 60 Report really meant. In January 1965 [AB19.3.7], Knuth published his famous "List of the remaining trouble spots in ALGOL 60" (later published in CACM 10 October '67), which documents the well-known problems with regard to such things as numeric labels. It was not proposed to fix these things at that time (the controversies still raged), but rather to point out to users features of the langauge to steer clear of. Of course, this was not the end of the worries. Further worries were reported by Bekic [AB20.3.6] and Medema [AB20.3.7]. In [AB27.2.1] Bryan Higman pointed out the reason why making the value of the controlled variable undefined on exit from a <for statement> achieved precisely nothing for the implementor, since its value at the end of each <for list element> must be defined in case it is needed in the next one, as in

for i := 1 step 2 until 9, i+1 do print(i)

which must clearly print *12* on the last iteration.

Miscellany.

Here, from J. Nievergelt [AB31.3.5] is what I regard as the ALGOL 60 equivalent of the FORTRAN Venus Probe Joke (you know, the one that starts *DO 1 I = 1.2 ...*). The difference between the two following is just one ";" – clearly just a <dummy statement> which will make no difference to the meaning.

begin procedure p(k) ;   integer k ; k := 1 ; end

begin procedure p(k) ;; integer k ; k := 1 ; end

It ain't so simple, however. You haven't spotted it? Clue: where does the declaration of the *procedure p* finish?

A rather unlikely semantic problem was reported in [AB26.1.3]. In England, the newly installed automatic barriers on railway level crossings were provided with a warning

"Stop while lights are flashing."

But it seems that the local dialect in parts of Northern England, particularly in Yorkshire, ascribes to "while" the meaning which the rest of the world understands by "until", with the obvious possibility of disastrous accidents at level crossings. The wording on the notices had had to be changed. Many AB readers lived in Yorkshire, and there were expatriate Yorkshiremen throughout the world, so perhaps there were problems with the *while* of ALGOL 60? As a Northener myself (but from Lancashire rather than Yorkshire) I can assure you that "while" is indeed often used with an "until" meaning (I even do it myself in the right company), but the ambiguity can in fact always be resolved by the context.

Another nice touch (with hindsight) was a news item [AB20.1.1] entitled "Release of syntactic ALGOL compiler for PASCAL". Mystified? Well, this was 1965, and it seems that Philips manufactured a machine called "PASCAL" in those days.

Another long-standing tradition was established in [AB27.3.2] (see also [AB28.2.5, AB29.2.1]) with a paper by Brian Wichman on "Timing ALGOL Statements". This was, of course, many years before his invention of the "Whetstone", the systems compared here being the two KDF9 compilers, the ICL 1905, the Elliott 4120 and 4130, and the CDC 3600. Timings for *x := y↑z* ranged from 99 μ-secs to 47700 μ-secs.

ALGOL X and ALGOL Y.

It was always the intention of WG2.1 to proceed to a more advanced language, and the first mention of ALGOL X (which eventually became ALGOL 68) and of the mythical ALGOL Y (originally conceived as a language which could manipulate its own programs, but in fact degenerating into a collection of features rejected for ALGOL X) was in a paper entitled "Cleaning Up ALGOL 60" by Duncan and Van Wijngaarden [AB16.3.3] which proposed a type *string*, which is reasonable enough, but also a type *label* and a removal of the restriction forbidding a *go to* from outside into a block (clearly, such things were not considered in the least bit harmful in those days).

It should be noted that it was always the intention in those early days for ALGOL X to be a strict upwards extension of ALGOL 60. It was only gradually that the folly of this view became plain.

Features Proposed for ALGOL X.

There followed a long series of wish-lists for the new language, of which a long series of important articles by Tony Hoare will be examined in more detail below. A long list by Peter Naur [AB18.3.9] asked for environment enquiries, *short* and *long* modes, operator-declarations, *string*s (but crude), yet more *label*s and *switch*es, non-rectangular arrays (they nearly did make it into ALGOL 68), operators such as *mod*, *abs*, *round*, *odd*, procedures with variable numbers of parameters, a hint of *struct*s, and [AB19.3.11] a type *character* coupled with a separation of conversion from I-O, and a type *bits*, and [AB22.3.7] a suggestion to replace the ALGOL 60

procedure $p(a,b)$; *integer* a; *real* b; ... by

procedure $p($*integer* a, *real* $b)$; ...

(I am amazed that such a, now generally taken for granted, feature should have been so late in appearing). Quite a lot of familiar stuff there! The wish list of the ALCOR group [AB19.1.1] mentioned complex arithmetic, variable precision, strings, "simultaneous statements", simpler loop-statements, restricted call-by-name, collateral assignment of array elements, and much else besides. Rutishauser [AB19.3.10] was asking for elaborate mechanism to denote lists, with associated features in the for-statement. Van de Laarschot and Nederkorn [AB19.3.2.1] wanted to ensure that *string*s would be first class citizens (no length limitations, proper assignments, etc). Samelson [AB20.3.3] wanted anonymous routines (i.e. λ-expressions or routine-texts) chiefly so that they could be in-situ actual-parameters. In [AB21.3.1] Seegmüller proposed *reference* types (principally as an aid to parameter passing) but, because coercion had not been invented yet, he needed a special "undereferencing" operator *ref*. Thus *integer reference* ii; *integer* i; *ref* ii := *ref* i (to assign a reference to i), or $ii := i$ (to assign the *integer* in i). Genuine coercion (or at least the widening coercion as we now know it) came from Dijkstra [AB21.3.3]. David Hill [AB22.3.9] was pressing for the "operate and becomes" operators, such as "+:=" (except that they were spelt "+:"). O-J Dahl [AB24.3.5] and Král [AB25.3.2] made strong pleas for Multi-programming (though I doubt whether the *par* clause of ALGOL 68 was quite what they had in mind).

Tony Hoare's contributions.

A series of articles by Tony Hoare had a great influence on the development of ALGOL X, and they are worth looking at in more detail.

Case expressions (and statements) [AB18.3.7]. These were just like the case-clauses that eventually got into ALGOL 68, except that the alternatives of the list were separated by *else*s and there was no *out* part (the effect of an out-of-range *case* being undefined, even for <case statement>s). The intention was most definitely to provide an alternative to the ALGOL 60 *switch*es.

Record handling [AB21.3.6]. Records were conceived much as the *struct*s in ALGOL 68 to which they gave rise. However, they had three substantial (and deliberate) restrictions which ALGOL 68 does not impose.

1. Records could only be created, on demand, on the heap. There were to be no locally declared record variables. Thus there were to be *reference* variables and *reference* fields and these were the sole methods of accessing records.
3. However, *reference* values could not point to local variables (so that no scope problems could arise).
2. Records could not have other records as fields (although *array* fields were envisaged).

The proposal envisaged both a garbage-collector and a PASCAL-style *destroy*. There was also provision for *reference*s to *union*s of other types, and a special construction equivalent to the ALGOL 68 conformity-clause for taking the *union*s safely apart. There was further discussion of these ideas in [AB23.3.2] which included, by way of an example, the earliest publication known to me of Dijkstra's well-known algorithm for finding the shortest path between two nodes of a graph.

It is interesting to trace the origins of these ideas for records and references to them. Already, in [AB18.3.12], McCarthy had proposed *cartesian*s (records) and *union*s, but no dynamic allocation and no *reference*s. The first language really to provide complex data structures out of records and pointers was Doug Ross' AED-0. SIMULA 67 also provided much input to Hoare's proposal.

It would seem that the concept of *reference* and its relationship to records, variables and procedure parameters caused much discussion within the Working Group much if it based, so far as I can see, on the difficulties of recognising the underlying fundamentals and inventing suitable terminology for them. A letter from Doug Ross [AB26.2.2] to Van Wijngaarden illustrates this point. It discusses various features, most of which I can recognise in ALGOL 68, but I am sure that Doug thought he was proposing something radically different.

A separate proposal, also contained in Hoare's paper, was for enumeration types (as now provided in PASCAL, but originally introduced with the word *set*).

Cleaning up the for statement [AB21.3.4]. This proposed implicit declaration of the control variable by its mere mention after a *for*, it was to behave like a *value* parameter (so it could not be assigned to), and the expressions for the initial, step and final values were to be evaluated only once. This all seems very familiar today, but the story did not end there. In [AB22.3.1] Galler suggested that the for statement should become an expression, returning the final value of the control variable (I think I like this), and in [AB25.3.2] David Hill was complaining that making the control variable an implicit declaration would scupper Jensen's Device.

File processing [AB25.3.3]. *file*s were to be ordered sequeces of *record*s, and in its initial form the proposal provided the effect of the PASCAL *file of sometype*. However, there was also to be provision for different record types within the one file, and means to locate specific records (and even to store such locations as fields of other records).

Set manipulation [AB27.3.4]. This proposal was the forerunner of the *set*s in PASCAL, it being explicitly envisaged that they would be stored as bit patterns. However, it went somewhat beyond what eventually appeared in PASCAL, for example there was to be an operation to iterate through a set, and a *shift* operator. Hoare's proposal was roundly criticised by Landin [AB27.3.6] because it was too ad hoc – being designed as a clever way of using bitpatterns rather than an attempt to embody the mathematical concept of sets. However, the interesting part of Landin's critique was his (then) novel idea of trying to identify types with the operators that could be applied to them, complete with an axiomatic description of the type *integer*.

Subscript optimisation and subscript checking [AB29.3.6]. The intention was to allow, for some array A,

for a *in* A *do* $sum := sum+a$;

The motivation was to optimise the efficiency of loops that scanned arrays and, in particular, to make it impossible for subscript bound errors to arise, at the same time eliminating (or much reducing) the necessity for such things to be checked at run time.

Text processing [AB29.3.7]. I do not undersdtand this proposal. It suggests a type *character* and the handling of textual data in *character array*s, but these are conceived as of fixed size and their is no discussion of any need for strings of arbitrary length such as the real world is full of. There are proposals for slicing such arrays (just as in ALGOL 68), and also for fixed collections of characters like the ALGOL 68 type *bytes*. All this seems to be so far behind ALGOL 68 as it then stood (MR93 had been published and Tony was well aware of it) that, even though he disliked ALGOL 68, the lack of discussion – critical if necessary – of these issues I find very odd.

The History of ALGOL X.

The Early Days.

Work on ALGOL X started in earnest at the Princeton meeting of WG2.1 in May 1965 [AB21.1.1]. *strings* were seen as important, but a small majority preferred to compose them out of *characters* using existing data structuring tools. Trees were envisaged (but note that this was before Hoare's Records paper). There was considerable influence from Wirth's Euler language, but still confusion about parameter passing. Strong typing was clearly envisaged. Draft proposals for a full language were solicited for the next meeting.

At the next meeting in October 1965 at St Pierre de Chartreuse [see excellent report by Mike Woodger in AB22.3.10], there were three such drafts on the table, by Wirth (with extensive comments by Hoare incorporating his Record Handling), by Seegmüller, and by Van Wijngaarden – the famous "Orthogonal design and description of a formal language" wherein W-grammars first appeared (did you know that metanotions originally consisted of just one capital letter and that there were exactly 26 of them). The four of them (Van Wijngaarden, Wirth, Seegmüller and Hoare) were commissioned "to agree among themselves" and to produce a proposal for "final approval" (sic) at the following meeting. BUT, it had also been decided that Van Wijngaarden's method of description should be used, so that he would get to write the text. Nevertheless, although Van Wijngaarden's proposal was more concerned with method of description than with language content, it seems that the Wirth/Hoare proposal had been effectively rejected and permission was given for it to published independently, which was done in CACM in June 1966, together with the remark that it had been felt that "the report did not represent a sufficient advance on Algol 60, either in its manner of language definition or in the content of the language itself". The CACM version did, however, use just a little bit of 2-level grammar, with acknmowledgment to Van Wijngaarden's document.

Decisions made at this meeting were
- *label* variables were out. *strings* (but maybe with declared maximum length) *complex*, *bits* and various *long* modes were in.
- Hoare's Records were accepted.
- Row-displays of some form were envisaged.
- Collateral elaboration to discourage side effects.
- Parrellel-clauses (but no semaphores as yet).
- Blocks (containing declarations and statements) could stand as expressions, returning their last expression (or the one before a completion-symbol, which was "." in those days rather than the current *exit*).
- Identifiers could represent values (if declared *val*) or locations (*loc*) or variables (*var*).
- Conditional-expressions and Hoare's case-expressions.
- The control variable of a *for* to be implicitly declared and constant, *step* and *until* parts to be elaborated only once, and *while* forms to have no control variable at all.
- I-O transmission and data conversion to be separated.
- Call by value and by *name* (but name calls to be indicated at point of call also, and actuals of kind *loc* to be passed by reference). Confused? Try the following example:

  real val f (*real val x*, *real loc y*, *real var z*) = <expression>;

  This defines a function whose calls may supply:
  - an *integer* or *real* expression for *x* (input),
  - a *real* or *complex* variable for *y* (output),
  - but only a *real* variable for *z*.

  Still confused?

The Wirth/Hoare language, as described in the CACM article, was in due course implemented on an IBM 360 by Wirth, under the name "ALGOL W", during a visit to Stanford in 1966, and in [AB24.3.3] he describes a few refinements of the language that he found necessary. See also [AB26.3.4] for comments on this language by the Japanese ALGOL Working Group. The Japanese were also active in language design, and produced their own "ALGOL N" [AB30.3.2], a simplified form of ALGOL 68 with a simplified method of description.

In the next issue [AB23.1.1] it was merely announced that the next meeting of the Working Group had been postponed from April 1966 to October 1966 "to allow the sub-committee ... more time for their work". Apparently, some interim document was produced at that meeting, and also a substantial proposal for transput [AB25.1.1]. It was confidently predicted that the main business of the following meeting, in May 1967, would be ALGOL Y. The next news [AB25.0.1] in March 1967 apologised for the fact that the draft report on the new language did not accompany that issue. "The work at Amsterdam, which includes implementation studies, has taken longer than anticipated." It would be distributed direct from Amsterdam shortly and there would be time for readers to comment before the meeting in September.

Came August and AB26, but still no Draft Report. It seems that at the May meeting, despite the confident predictions, "Discussion of ALGOL Y was rather limited", although there was still no hint of the furore that was to come. The September meeting had been postponed until "not less than 3½ months after distribution of the draft". In the event the Draft, the (in)famous MR93, did not appear until February 1968. Comments were invited [AB27.1.1] and were to be considered by the WG at its meeting in June. The possibility was still being envisaged of obtaining final IFIP approval of the document at the IFIP Congress in Edinburgh in August. And at the end of AB27 appeared the first example of a long tradition of errata to the Draft Report – a mere 9 pages of them.

The Troubles.

The appearance of MR93 was the cause of much shock, horror and dissent, even (perhaps especially) amongst the membership of WG2.1. Generally, only those members who had been present at the meetings in October 1966 and May 1967 had seen the earlier drafts, and the absentees included such notable names as Naur, Dijkstra and Garwick (for both meetings) and Hoare and Woodger (May '67 only) [AB25.1.1, AB26.1.2]. It was said that the new notation for the grammar and the excessive size of the document made it unreadable. However, it could be read and understood, as I demonstrated myself by extracting the underlying language from it and serving it up as "ALGOL 68 with Fewer Tears" [AB28.3.1] – however, I must confess that the task occupied me fully for 6 man-weeks. "Fewer Tears" was a paper written in the form of an ALGOL 68 program, and its definitive version can be found in the Computer Journal <u>15</u> 2 May '72.

A small (and stormy) meeting of the Working Group in June 1968 instructed the authors to undertake considerable revisions to be submitted to the Group for final acceptance or rejection in December.

Examples of the difficulties which people had with the Report can be found in various articles. Turski [AB29.2.4] found inconsistencies in the definition of the underlying "paper computer". Hoare [AB29.3.4] wanted a more primitive core language with assignment, fancy subscripting, and the like being added by means similar to the addition of new operators, and he called for simpler coercions and no multiple *refs*. Many others submitted detailed suggestions, both to the AB and directly to Amsterdam.

In May 1968, the tenth anniversary of ALGOL 58, a colloquium had been held in Zurich [AB28.1.1], where the recently distributed Report came in for much discussion, being "attacked for its alleged obscurity, complexity, inadequacy, and length, and defended by its authors for its alleged clarity, simplicity, generality, and conciseness". Papers given at the colloquium included "Implementing ALGOL 68" by Gerhard Goos, "Successes and failures of the ALGOL effort" by Peter Naur [AB28.3.3], and some "closing remarks" by Niklaus Wirth [AB29.3.2]. Naur's paper contained criticism of MR93, as providing "the ultimate in formality of description, a point where Algol 60 was strong enough", and because "nothing seems to have been learnt from the outstanding failure of the Algol 60 report, its lack of informal introduction and justification". IFIP seemed to be calling for immediate acceptance or rejection (thus precluding further development of MR93), and thus IFIP was the "true villain of this unreasonable situation", being "totally authoritarian" with a committee structure and communication channels entirely without feedback and with important decisions being taken in closed meetings. "By seizing the name of Algol, IFIP has used the effort ... for its own glorification." Strong words indeed! Wirth's contribution was also critical of MR93, and of the method whereby the Working Group, after a week of "disarray and dispute", and then "discouragement and despair" would accept the offer of any "saviour" to work on the problems until next time. Eventually the saviours "worked themselves so deeply into subject matter, that the rest couldn't understand their thoughts and terminology any longer". Both Naur and Wirth resigned from the Group at this time.

Now at that time the ALGOL Bulletin, which was (and still is) an official IFIP publication, was printed by the Mathematisch Centrum in Amsterdam, and when AB28 (originally dated July 1968) containing Naur's paper was sent there for printing, Van Wijngaarden refused to do so. At the next WG meeting in North Berwick in August there was a furore, with words like "mud" being applied to Naur's paper, and Fraser Duncan refusing to compromise his editorial freedom. AB28 did eventually get published without expurgation, but some months late (and, to be fair, there were also floods and mislaid manuscripts which contributed to the delay).

The North Berwick meeting, which was the first that I myself attended, was five days of continuous politics. I have just been re-reading the minutes, and they display a sorry tale. A poll was taken concerning the future work of the group, and the best part of a whole day, both before and after the poll, was taken up with its form, and how its results were to be interpreted. One can see that mutual trust between the two parties had been entirely lost, and jockeying for position was the order of the day, and of every day. Barely half a day was spent in technical discussion of the Report.

The meeting at Munich in December 1968 for making the final decision was also political, but there was much more willingness to reach a compromise. There had now been three further drafts, MR95, MR99 and MR100, and there was much more technical discussion than there had been at North Berwick, resulting in the final document MR101. The meeting devoted much time to drafting a covering letter which would be acceptable to all, and which "did not imply that every member of the Group, including the authors, agreed with every aspect of the undertaking" but decided that "the design had reached the stage to be submitted to the test of implementation and use by the computing community". This was still too strong for some members, however, and on the last afternoon of the meeting a short Minority Report signed by eight members was presented. The Formal Resolution submitted to TC2 clearly implied that the Covering Letter should accompany the final publication of the Report, together presumably with the Minority Report. In January 1969, TC2 duly forwarded the Report and Covering Letter (but not the Minority Report) to the IFIP General Assembly, which in May authorised publication of the Report, but without the Covering Letter. The text of the Minority Report may be found in [AB31.1.1]. It spoke of the effort which had gone into ALGOL 68 as an experiment which had failed, and claimed that the language's "view of the programmer's task" was ten years out of date and that, doing little to help "in the reliable creation of sophisticated programs", the language "must be regarded as obsolete". It is a pity that its production at the very last moment precluded a more constructive minority report, such as Doug Ross had been trying to organise in the months preceding the meeting (as documented in [AB30.2.3]). The Munich meeting had also recommended to TC2 the setting up of a new WG2.3 on Programming Tools, and after the meeting 11 members resigned, to become the nucleus of the new group.

Post Munich.

ALGOL 68 now entered on its "maintenance phase". Of course people found problems, and there were worries about the way infinite modes had been defined in the Report. Fraser Duncan had been warning about the mathematical unsoundness of these for a long time, but articles by Meertens [AB30.3.4] and Pair [AB31.3.2] did not make their points clearly enough to be noticed (it was not until much later that Hendrik Boom showed the lurking ambiguity plainly for all to see, leading to the fixing of the problem in the Revised Report). In [AB30.3.3] Koster introduced his famous algorithm for determining the equivalence of two modes (essentially, they are equivalent if you can show that they match given the hypothesis that they are the same mode).

The original Covering Letter had envisaged that a revision of the Report might be needed in due course, and in 1971 the WG issued a general call for suggestions [AB32.2.8], stressing however that "for the present, the language definition shall remain stable", and that "there should be only one revision". There then followed a variety of reports by subcommittees of the Working Group, which had been charged with collating such material. Examples are reports on Data-processing and Transput [AB32.3.3, AB33.3.4] containing quite elaborate proposals for "record transfer" to mass-storage devices, and others on General Improvements to the language [AB32.3.4, AB33.3.3], on Conversational Languages and on Operating System Interfaces [AB32.3.5, AB32.3.6, AB33.3.6], and on Sublanguages [AB33.3.5]. All these formed the input to the meeting at Novosibirsk in August 1971 at which a timetable for the production of a Revised Report was laid down.

Also at this meeting, Fraser Duncan resigned as Editor of the ALGOL Bulletin, having been responsible for the production of 18 issues spread over 8 years. I was appointed to take his place, and this seems as good a reason as any for terminating this account here.