ISSN 0084-6198

Algol Bulletin no. 49

MAY 1983

CONTENTS

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

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

The ALGOL BULLETIN is published at irregular intervals, at a subscription of $11 (or £6) per three issues, payable in advance. Orders and remittances (made payable to IFIP) should be sent to the Editor. Payment may be made in any currency (a list of acceptable approximations in the major currencies will be sent on request), but it is the responsibility of each sender to ensure that his payment is made in accordance with the currency requirements of his own country. Subscribers in countries from which the export of currency is absolutely forbidden are asked to contact the Editor, since it is not the policy of IFIP that anyone should be debarred from receiving the ALGOL BULLETIN for such a reason.

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

Back numbers, when available, will be sent at $4 (or £1.80) 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.

AB49.0 EDITOR'S NOTES.

ALGOL 60 Standardization

The long process of producing as ISO Standard for ALGOL 60 is about to terminate. The draft which was voted upon by the various member bodies of ISO contained a reference to the Modified Report (actually, Clause 6 of the Draft Standard simply stated that the text of the Report was deemed to be inserted at that point). Now, the ISO Secretariat have decided to print the actual text in this place (however, what they print is actually a photograph of the text from the Computer Journal, but with all the published errata incorporated, so there is no need to worry that some suibtle change has crept in). It will be known as ISO Standard 1538 and, if there are no last minute hitches, it should be published about the same time as this bulletin.

This issue of the Bulletin also contains a clarification to the Modified Report. This has been approved by the Working Group. It does not alter the language in any

way, but it may save you some effort in trying to puzzle out what the Report means in the particular situation considered.

ALGOL 68 Standardization

This has now reached its critical stage. After various administrative delays, the formal proposal from IFIP to ISO finally resulted in a letter ballot of the member countries of ISO/TC97. The votes have to be returned by the beginning of May, and we have to get at least 5 countries who will agree to "participate" in the work. Thus, I cannot tell you what is going to happen yet, but keep your fingers crossed!

Exception handling in ALGOL 68

The ALGOL 68 Support Subcommittee of WG 2.1 has been considering exception handling at its last few meetings. There have been two proposals under consideration. One, which is all done with routine-texts, is designed to be readily added to any existing implementation simply by writing some additions to the standard-prelude (maybe even a customer of the implementation could do it himself). However, this causes the syntactic sugar for what the user has to write to be a little bit cumbersome. The other is considered to be more convenient in use, but it seems that, at least in some styles of implementation, it will be necessary to tinker with the compiler in order to implement it.

As well as finding difficulty in choosing between these two schemes, the subcommittee is also in some doubt as to whether it is proper to produce even semi-official extensions to ALGOL 68 at this late stage. Since both schemes are now well-defined and understood, it has been decided to publish them both in this bulletin, but without any official recommendation. Comments from you (and even trial implementations) are now in order, and it may be that the Subcommittee will reconsider the matter at a future meeting.

AB49.1 Announcements.

AB49.1.1 Hans Bekic 1936 - 1982

Hans Bekic, who was for many years a member of IFIP Working Group 2.1, died in a mountaineering accident on October 24th, 1982. Since 1961, he had been a leading member of the IBM Laboratory Vienna, where he was heavily involved, firstly, in the formal, operational, definition of PL/1 and, latterly, in dentotional methods of language definition (and especialy their application to parallel processes).

For his WG 2.1 associations, however, one must look first of all to his work on ALGOL 60 - his implementation, with Peter Lucas, of that language (one of the few to take account of all the concepts of ALGOL 60), and its systematic transformation into a more primitive language (effectively the first successful formal definition of the semantics of a programming language). He was involved in the discussions leading up to the definition of ALGOL 68, but my own chief recollection of him was his attempts, spread over many meetings during the revision of ALGOL 68, to persuade us to relax the scope restriction on routines (so as to enable composition of functions, and the like). He did not win this particular battle but we did, afterwards, prepare a partial-parametrization feature for the language which gave many of the same benefits.

His work was always marked by depth, insight and extremely high personal standards. He published little, but his unpublished manuscripts were circulated and referenced widely. The influence of his achievements will still be felt for many years to come.

C. H. Lindsey
(with help from H. Zemanek and C. B. Jones)

AB49.1.2 Yet another definition of ALGOL 60

Ever since the work by Bekic mentioned above, ALGOL 60 has been used as a Test Bed by those who wanted to try out their latest formal language-definition technique. The latest in this line is a denotational definition using the Vienna Development Method (VDM), and contained in the book "Formal Specification and Software Development" by Dines Bjørner and Cliff B. Jones, published by Prentice Hall (at an exorbitant price for which Cliff denies all responsibility).

AB49.1.3 Book Review : Correctness Preserving Program Refinements: Proof Theory and Applications

by R.J.R. Back.
Mathematical Centre Tracts 131, Amsterdam 1980.
ISBN 90 6196 207 2.

This 118 page monograph presents an interesting theory of program design by stepwise refinement. The author rightly points out that the weakest pre-condition technique described by E.W. Dijkstra does not handle data refinement proofs. (This criticism can also be applied to many other books on program proofs - e.g. "The Science of Programming" by D. Gries.) Here, the author presents a notion of specification and refinement which embraces both decomposition of control structure and refinement of data.

Specifications are written with "atomic descriptions". These both contain a logical expression and bind variables: this combination takes some time to get used to! An infinitary logic is used in which infinite disjunctions and conjunctions over formulae are allowed. Little detailed justification is given for this choice beyond claiming that such expressions are needed to express the weakest pre-condition of loops.

The semantics of specifications are relations and, following de Bakker, undefined elements are introduced to indicate non-termination of non-deterministic constructs. There is some discussion of unbounded non-determinism. It would have been interesting to have a comparison with D. Park's transfinite approach (cf. Springer LNCS No. 86). The ultimate reason for rejecting unbounded non-determinism is the inability to express such specifications in the chosen logic because of the restriction to a countable number of terms.

This is an extremely readable report which represents a development of the author's thesis. The only general criticism is that the examples are rather small.

C.B. Jones
Manchester

AB49.1.4 Book Review : A Bibliography of Lambda-Calculi, Combinatory Logics and Related Topics

by A. Rezus. Mathematisch Centrum, Amsterdam 1982.
ISBN 90 6196 234X.

This is an extensive bibliography of nearly 80 pages plus two Addenda of a few pages each. As a source of references it will no doubt be of considerable use to experts in the field. There is, however, little to aid the non-expert in finding interesting material. There are no "annotations". Furthermore, there is no attempt to classify the material along the lines suggested in Henk Barendregt's foreword (i.e. pure theory, the theory related to foundations, applications).

C.B. Jones
Manchester

AB49.1.5 Book Review : Deterministic Top-Down and Bottom-Up Parsing: Historical Notes and Bibliographies

by Anton Nijholt.
Mathematisch Centrum, Amsterdam.
ISBN 90 6196 245 5. Price Dfl 16.50.

This bibliography contains over 1000 references, dealing primarily with theoretical problems in the theory of parsing, but also covering some more application-oriented issues, such as compiler construction techniques.

The book is divided into three main sections, covering Top-down Parsing (i.e. LL(k) methods), LR-Grammars and Parsing, and Precedence Parsing. Each section starts with a survey covering the History of the particular method, the Formal Properties of the relevant grammars, the associated Parsing methods, Error Handling, Parser Generators, etc.

The three Bibliographies themselves are exceedingly thorough, entries from the most obscure journals taking their place beside all the classical papers on the subject. However, the entries are given only in strict alphabetical order of first author's name, with only the title of the paper and the bibliographic reference. There is, in general, no way to identify a paper on a particular topic, save for those papers which are explicitly referenced in the introductory surveys (although these do indeed include all the most important papers). How much more useful the book would have been with only one sentence to say what each paper was about (for titles of papers are notoriously unhelpful in this respect).

C.H. Lindsey
Manchester

AB49.1.6 Book Review : ALGOL 68 Preludes for Arithmetic in Z and Q

by Guenter Baszenski
Rechenzentrum der Ruhr-Universitaet Bochum, 1982.
Report No. 8203 ISSN 0341-0358.

This paper defines (and provides implemantations for) two ALGOL 68 preludes - one for performing arithmetic on integers of arbitrary size and one for performing arithmetic on rational numbers.

In the long-integer prelude, the mode *LINT* is defined together with operators + - * % *MOD* ** +:= -:= *:= %:= MODAB *ABS ODD SIGN* plus the usual relational operators. These also work between *LINT* and *INT* and v.v. There are conversion operators *L* and *I*, from and to *INT*, and also a procedure *over*, to give quotient and remainder, and operators *FAC* (factorial), *C* (binomial coefficients), *GCD* and *LCM*. Special transput and conversion procedures are provided for *LINT*s. The implementation is in terms of a sign and a modulus, which is stored as a row of *INT*s on the heap. It runs on the CDC implementation of ALGOL 68, but contains nothing that should prevent porting to other implementations.

The rational prelude defines the mode *RAT*, which is a structure of two *LINT*s. The operators provided are exactly those provided for the mode *REAL* in the ALGOL 68 standard-prelude, and work for all sensible combinations of *RAT*, *LINT* and *INT* (however, a special operator *FR* had to be provided for division of two *INT*s, because / is already defined in the standard-prelude to yield *REAL* in this case). As with *LINT*s, there are special transput and conversion routines.

It is not stated, but I would think it reasonable to suppose that both preludes would be available in machine-readable form from Bochum.

C.H. Lindsey
Manchester

AB49.2 Letter to the Editor

South West Universities Regional Computer Centre

University of Bath
Claverton Down
Bath BA2 7AY

Telephone Bath (0225) 60371
Telex 449097

Director J R Brookes MA FBCS

18 February 1983

Dr C H Lindsey
Editor, Algol Bulletin
Dept of Computer Science
University of Manchester
MANCHESTER
M13 9PL

Dear Dr Lindsey

I would like to inform readers of 'Algol Bulletin', particularly those interested in Algol 68, about the formation of the RS Algol 68 Implementors Group (RIG), and to describe some of the work we have done.

As the name suggests, members of the group are implementors of Algol 68 systems based on the portable Revised Report "RS" compiler front end from RSRE Malvern (described in RSRE Technical Note 802). RSRE themselves are also represented. The group has been meeting regularly since May 1981, its principal objectives being to maintain compatibility between RS implementations, to provide a forum for the discussion of common problems and points of interest between implementors and RSRE, and to promote the use of Algol 68 in general.

Compatibility is seen as very important for user acceptance of future implementations; we feel it is highly desirable that source programs should be readily portable from any one RS implementation to any other, and that there should be a consistent user view of the RS 'family' of compilers. The RS system itself, on the other hand, permits considerable flexibility in the implementation of the back end translator and the run-time software. Consequently, RIG has agreed on a set of standards and guidelines for implementors to follow so that compatibility is maintained. This compatibility will become even more important early in 1983 with the publication by Edward Arnold Ltd of a new Algol 68 text book, aimed primarily at users of RS systems, and written by Philip Woodward and Susan Bond. (Many readers will recall with affection their 'Algol 68-R Users Guide').

To clarify any misconceptions readers might have, I should point out that RS Algol 68 was designed to be very much closer to the Revised Report than Algol 68-R, its popular predecessor from RSRE. RIG has in fact spent some time reviewing the language implemented by the RS compiler, and as a result, many minor restrictions and deviations from the Report have been removed. (The main remaining restrictions are that, apart from labels and simply recursive procedures, identifiers must be declared before they are used, and that no parallelity features

are provided; on the plus side, RS systems contain a most powerful and secure modular compilation system).

Another activity of RIG has been to produce a standard test set for RS implementations. This consists of the MC (Amsterdam) tests, modified where appropriate to allow for language differences, along with a set of tests developed by Bernard Houssais at the University of Rennes (where they are using the Honeywell Multics implementation of RS Algol 68). The Rennes tests are generated automatically from a description of Algol 68 syntax and have proved remarkably successful at weeding out obscure bugs in the compilers.

Incidentally, the information given in AB47 on available RS implementations is now out of date. The ICL 2900 Series implementation is available under VME/B and VME 2900 (not VME/K) and there is now an implementation for Honeywell Level 68 machines under the Multics operating system. This implementation is available at commercial rates (but at a nominal charge for educational use) and is now in use at six installations in Europe and North America. Further information may be obtained from

Systems Development Manager
South West Universities Regional Computer Centre
University of Bath
Bath BA2 7AY
UK

In addition, SPL are currently developing an RS implementation for the VAX, and RSRE have implemented RS Algol 68 on their own Flex architecture. An implementation for Motorola 68000 is also being carried out as a research project at the University of Cambridge. We are naturally keen to encourage additional implementations and would be pleased to provide information and assistance to anyone interested.

Finally, one objective of RIG that has been relatively neglected so far has been that of promoting Algol 68. Why is is that Algol 68, despite almost invariably becoming the preferred language of anyone who takes the trouble to learn it, has failed to become widely used? To my mind the principle failure has been one of marketing, and here all of us in the Algol 68 community must share the blame; for too long we have been unduly inward-looking, ignoring the outside world of FORTRAN and Pascal programmers who fail to realise the benefits they are missing. Can I therefore exhort all of you who are interested in saving the language to think seriously about what can be done, and to be conscious of any opportunities that may arise to put forward the merits of the Algol 68 case (no pun intended!).

Yours sincerely

Gavin Finnie

Gavin Finnie
Secretary, RIG

AB49. 3. 1

Clarification to Modified ALGOL 60.

The following clarification has been issued by IFIP Working Group 2. 1. and deals with a question which was raised in connection with the Modified Report on the Algorithmic Language ALGOL 60. This clarification is not to be construed as a modification to the text of the Modified Report.

Interpretation of "call by name" where the actual and formal parameters differ in type.

1. Introduction

1. 1 The Modified Report on ALGOL 60 (section 4. 7. 3. 2) says that, when a formal parameter is called by name, "If the actual and formal parameters are of different arithmetic types, then the appropriate type conversion must take place, irrespective of the context of use of the parameter." A query having been raised about the exact meaning of this requirement, it seems worth while to try to clarify it. The aim is solely to make a clarification, not to change the intention.

1. 2 First it should be noted that ALGOL 60 has only three types, of which ***Boolean*** is not arithmetic, so it is only conflicts between ***real*** and ***integer*** that are involved. The possible cases are (1) ***real*** (***integer***) actual parameter with ***integer*** (***real***) formal parameter (2) ***real procedure*** (***integer procedure***) actual parameter with ***integer procedure*** (***real procedure***) formal parameter.

1. 3 A preliminary version of this note was published, in translation, in the Russian version of the Modified Report (Translator A. F. Rar, Editor A. P. Ershov, Moscow, 1982). However, that preliminary version has been found to be inadequate in that it did not allow for the case where a formal parameter is used as an actual parameter in a further procedure statement, nor the case of an assignment statement with more than one left part.

2. Interpretation

2. 1 If an actual parameter, called by name, and the corresponding formal parameter are of different arithmetic types, then the formal parameter is said to be ill-matched. If an actual parameter, called by name, is itself an ill-matched formal parameter, then the corresponding formal parameter is also said to be ill-matched.

2. 2 For the following explanation, let the functions ρ and ϕ be defined as:

```
integer procedure ρ(h);
value h; real h;
ρ := h;

real procedure φ(k);
value k; integer k;
φ := k;
```

Note: the Greek letters ρ and ϕ have been used for these functions to avoid any clash of identifier with those of the program, and to make clear that these are not standard functions that can be used by a programmer. For ease of human understanding, ρ may be pronounced 'round' and ϕ may be pronounced 'float' if desired.

2. 3 If a formal parameter is ill-matched, then each use of it in the procedure body, other than as a destination or as an actual parameter called by name,

is treated as if enclosed in parentheses and preceded by ρ. If furthermore it is specified as ***real*** (or as ***real procedure***) the function designator so formed is treated as if enclosed in further parentheses and preceded by ϕ. In the case of a typed procedure its actual parameter part is, of course, also contained within the parentheses.

2. 4 If a formal parameter is ill-matched and is used as a destination, the arithmetic expression whose value is to be assigned is treated as if enclosed in parentheses and preceded by ρ.

2. 5 Automatic type changes across an assignment may occur, according to the usual rules, after the above operations have been applied.

2. 6 The above operations, together with the other operations mentioned in sections 4. 7. 3. 2 and 4. 7. 3. 3 of the Modified Report, may lead to a left part list containing destinations of different types, in violation of the first sentence of section 4. 2. 4. In such a case, the strict interpretation would be that the procedure statement is undefined because it has not led to a correct ALGOL statement, as required by section 4. 7. 5.

2. 7 However, it may be found more convenient to allow such a construction as an extension, in which case the process should take place in three steps as laid down in section 4. 2. 3, except that the value assigned should be the value of the expression for all destinations that are of ***real*** type and not ill-matched, ρ(the value of the expression) for all destinations of ***integer*** type and not ill-matched, $\phi(\rho$(the value of the expression)) for all ill-matched destinations. The expression is to be evaluated once only, however, not three times.

3. Notes and Examples

3. 1 In the program

```
begin real s, t;
procedure f (x, z); integer x; real z;
      begin
      z := z+x; x := z+x
      end f;
s := 3.1; t := 5.1;
f(s, t); print(s); print(t)
end
```

the procedure statement f(s, t) is treated as

```
begin
t := t+ρ(s); s := ρ(t+ρ(s))
end
```

Consequently the values printed are *11.0* and *8.1*.

3. 2 It may seem surprising that, if the actual parameter is ***integer*** and the formal parameter is ***real***, both ρ and ϕ are applied. This is necessary however in special cases, and does no harm in other cases. Consider:

```
begin real x, z;
procedure b (y); real y; z := y;
procedure a (j); integer j; b(j);
x := 12.7; a(x); print(z)
end
```

In this program $z := y$ is interpreted as $z := \phi(\rho(x))$. The ρ is necessary as z is given the value *13.0*, not *12.7*, because it has passed through the ***integer*** parameter *j*. The ϕ is necessary to make *y* of ***real*** type within the *b* procedure. In the above case ϕ is not essential (but does no harm) as the change of

type of the assigned value happens automatically as it is assigned to *z*, but the fact that *y* is ***real*** is important in its own right (for example to disallow *y+2*, as integer division is allowed only for operands of ***integer*** type).

3.3 It may also seem surprising that, when a formal parameter is used as a destination, ρ is always used but not ϕ. This is because, if there is ill-matching at any stage, the integral value is required. If the expression is already integral ρ does no harm. ϕ is not required, as a change of type is automatic, if needed, across an assignment. The argument in 3.2 above, concerning integer division, does not apply here because the entire expression is having the function applied to it.

3.4 The program

```
begin integer i;
procedure f (y); real y;
      begin real x;
      x := y := 13.3; print(x); print(y)
      end f;
f(i); print(i)
end
```

is technically incorrect because

```
x := i := 13.3
```

is incorrect. Compiler writers, however, will probably find it considerably easier to allow it as an extension than to detect it as erroneous. If such an extension is permitted, the three values to be printed should be *13.3* (for *x*), *13.0* (for *y*) and *13* (for *i*). (The *print* procedure might not, of course, distinguish between the last two in the form of its printing.)

AB 48.4.1

A Proposal for Exception Handling in ALGOL 68.

by C. H. Lindsey
(University of Manchester)

1. Informal Description.

Even in programs which are logically correct, exceptional things can happen. When presented with inappropriate data, time and space can become exhausted, numbers can go out of range, and matrices can turn out to be singular. Sometimes, these exceptions are detected by hardware or by the implementation (we call these "system exceptions"). Sometimes, they are detected by tests incorporated by the programmer (we call these "user exceptions"). Sometimes, they are not detected at all (e.g. because some run-time check has been turned off – we call these "undetected exceptions"). The Report does not distinguish between system and undetected exceptions – it merely states that at such a point the further elaboration is undefined (R1.1.4.3.b, 2.1.4.3.h, 2.2.2.b). In actual implementations, a system exception usually causes immediate suspension of the program with suitable diagnostic messages. An undetected exception allows the program to continue with erroneous results, possibly triggering a system or user exception later on.

In many situations, suspension of the program could be most embarassing. A database might be left in an inconsistent state. Some piece of equipment being controlled might fail. Results already accumulated might be lost. The user, particularly an interactive one, might have preferred to ignore the data that caused the trouble and to continue with the next input. The system to be described allows the programmer to specify traps for both system and user exceptions.

A "trap" is a routine to be called only when the associated exception happens. Different traps may be associated with different kinds of exception, and the association lasts throughout a specific range (unless reassociated within an inner range, of course). In our proposal, this range is some routine-text, and we will illustrate the method with an example of a user exception for handling singular matrices.

```
EXCEPTION singular = new exception; # EXCEPTION is a new mode #
PROC gauss = (REF [,] REAL a, REF [ ] REAL rhs) VOID:
      COMMENT a procedure to solve a set of simultaneous
            equations COMMENT
      BEGIN C the usual algorithm for gaussian elimination which, at some
                  point, may discover that a is singular C;
            IF C it makes this discovery C
            THEN RAISE singular
            FI;
            C rest of algorithm C
      END;
```

Within some given range of his program, the user decides how he wants to handle this situation:

```
[1:n, 1:n] REAL matrix, [1:n] REAL r;
handle (VOID:
      BEGIN # of range in which the proposed trap is to apply #
            C compute matrix etc. C;
            gauss(matrix, r);
            C process the results in r C
      END # of range of trap # ,
      TRAP (singular, VOID: (
                  print("matrix was singular; proceed with next case");
                  GOTO next case) )
```

where *next case* is a suitable label elsewhere in the program. Here *handle* is a procedure of mode *PROC(PROC VOID, [ ] TRAP)*, where

MODE TRAP = STRUCT(EXCEPTION exc, PROC VOID handler).

Calls of *handle* will usually be written in this form, with routine-texts written in situ for all the *PROC VOID*s. In general, a row-display of *TRAP*s providing handlers for various exceptions would be provided, but in this case there was just one, and so the rowing coercion took care of it (observe the cast *TRAP (..., ...)* which is syntactically necessary for the rowing to work properly, and is desirable for clarity in other cases).

This particular handler was a very simple one, but was typical insofar as it finally terminated with a jump. One could imagine a much more complex handler which made some subtle alteration to *matrix* and called *gauss* again, possibly inside a different call of *handle* to deal with any further singularities. But, in general, a handler should finally terminate with a jump (which, according to the syntax of the language, must be to some label outside the call of *handle*). Of course, there may exist at some moment several nested handlers for a given exception. For example, the programmer may have provided some general handler for *singular* which enclosed most of his program; but at some particular inside call of *gauss*, where he foresaw some particular possibility of singularity arising, he might provide a more local one. When the exception occurs, it is always the most local handler (in the dynamic sense) that is entered. If it terminates with a jump, then the matter is presumed to be resolved, and the program continues from the label jumped to. If, however, it tries to return to its caller, it is presumed that the matter is not resolved and the next outer handler is entered. Eventually, there will be no outer handlers left and the program is aborted (presumably with whatever diagnostic printout the system normally produces).

Since a handler will frequently do nothing but jump to a place where there is a sensible continuation, it is possible to make use of the automatic "proceduring" of jumps (R5.4.4.2.Case B) and the option of omitting the *GOTO* (neither feature available in ALGOL 68S, however). This combines well with the use of the completion-symbol (*EXIT*) in a serial-clause (R3.2.1.b).

```
REAL x =
     BEGIN
          INT j, REAL y;
          PROC reciprocal = (INT i) REAL: 1.0/i;
          handle (VOID:
               BEGIN read(j);
                    y := reciprocal(j);
               END ,
               (TRAP (arithmetic error, overflow),
                TRAP (others, other error) )
               );
          y
     EXIT
     overflow:
          max real
     EXIT
     other error:
          print("bad input");
          reciprocal(max int)
     END
```

arithmetic error and *others* are built-in exceptions. In this case, *arithmetic error* would catch division by zero, and *others* might catch troubles in *read* (assuming no suitable event routine had been provided). Whatever happens, some value or other is bound to get ascribed to *x*. The example also shows how a construct involving *handle* can be made to return a result (unfortunately, *handle* itself must always return *VOID*), and also how objects should be declared in order that they may be visible either inside the handler or, as in this case, at the place that the handler jumps to.

The full list of built-in exceptions is as follows. Not all implementations will necessarily be able to raise them all. They have deliberately been left general since, although one piece of hardware might be able to distinguish "floating point overflow" and "division by zero" as two distinct cases, another might give the same interrupt for both. It is thought that no system should have difficulty in selecting the appropriate exception from the following list. The errors listed within square brackets after some of the items are specific transput errors recognised by J.C. van Vliet's implementation Model for ALGOL 68 Transput (Mathematical Centre Tracts 110, Part 2); this suggested alloocation may help to clarify the intent of those exceptions.

time exhausted
space exhausted
: there might not be much that a handler could actually do in these cases

undefined value
: cases where an operation on a value requires it to be well defined; e.g. the destination of an assignation or an object to be dereferenced

arithmetic error
: all kinds of overflow, division by zero, square roots of negative quantities, etc.

bounds error [*wrongmult, posmax, posmin*]
: including errors in subscripts and in trimmers, incompatibilities when assigning complete multiple values, and a few transput errors as indicated

scope error

transput impossible [*nowrite, noread, noestab, noset, noreset, nobackspace, noshift, noreidf, nobin, noalter, nomood, notopen, badidf, notavail*]
: mainly for when something is attempted for which the appropriate *x possible* returns *FALSE*

file end [*nocharpos, noline, nopage, smallline, wrongpos, wrongset, wrongbacksp*]
: logical or physical file end

char error [*no digit, wrongchar*]

value error [*wrongval, wrongbin*]

format error [*noformat, wrongformat*]
: observe that the last five exceptions all correspond to specific calls of *undefined* in the standard-prelude, and the last four are in general only raised if no user's event routine has been provided (or when same has returned *FALSE*)

abort
: to be *RAISE*d by users in order to abort their programs deliberately; unless it had been explicitly trapped, the system's usual postmortem action would then ensue

others
: any of the above or any user exception, for which no more specific handler has been provided.

Note that there is no reason why a user should not *RAISE* a built-in exception (one can even envisage sensible applications of this).

2. Formal Definition

1. Standard prelude

The following forms are added to the standard-prelude. In these forms, the phrase "the calling of A", where A is an identifier or an operator, stands for "the calling (R5.4.3.2.b) of the routine ascribed, during the elaboration of these forms, to A".

```
a)  MODE EXCEPTION = STRUCT(INT F) ;
b)  ? PROC makexception = (INT i) EXCEPTION:
        (EXCEPTION e; F OF e := i; e) ;
c)  ? INT last exception := 1 ;
d)  PROC newexception = EXCEPTION: makexception(lastexception +:= 1) ;
e)  MODE TRAP = STRUCT(EXCEPTION exc, PROC VOID handler) ;
f)  PROC handle = (PROC VOID user program, [ ] TRAP traps) VOID:
        user program ;
g)  OP RAISE = (EXCEPTION e) VOID:
        BEGIN
            C consider the environ in which the closed-clause suggested by
            this pseudo-comment (R10.1.3 Step 7) is being elaborated C;
            WHILE
                (   C the considered environ is not the first environ
                    established (according to a declarative) during a
                    calling of handle C
                OR
                    C the considered environ has been considered
                    previously during some other calling of RAISE from
                    which the elaboration of the closed-clause suggested
                    by this pseudo-comment is descended (R2.1.4.2.b)
                    C
                )
                AND
                    C the considered environ is not the primal environ
                    (R2.2.2.a) C
            DO  C consider instead the environ upon which the considered
                environ was established (R3.2.2.b) (or around which it was
                established if no version to be established upon had been
                specified) C
            OD;
            IF  C the considered environ is not the primal environ C
            THEN
                COMMENT the locale of the environ now considered
                    corresponds (R2.1.1.1.b) to the parameters of the
                    newest, not heretofore considered, calling of handle
                COMMENT
                [ ] TRAP traps =
                    CO the traps parameter of that handle, i.e. CO
                    C the (multiple) value accessed inside the locale of
                    the considered environ by 'STOWED letter t letter r
                    letter a letter p letter s' (traps), where 'STOWED' is
                    the mode specified by TRAP C;
                PROC others handler :=
                    VOID: RAISE e;
                    # may be called if traps does not include others #
                PROC handler :=
                    VOID: others handler;
                    # will be called if traps does not include a specific
                    TRAP for e #
                PROC choose trap = ([ ] TRAP traps) VOID:
                    # examines the traps collaterally #
                    IF UPB traps >= LWB traps
                    THEN (
                            (   TRAP firsttrap = traps[LWB traps];
                                F OF exc OF firsttrap = F OF e
                            |   handler := handler OF firsttrap
                            |:  F OF exc OF firsttrap = 0
                            |   others handler := handler OF firsttrap
                            )
                        ;   choose trap ( traps[LWB traps + 1:  ]
                        )
```



```
                    FI;
                choose trap(traps);
                    # to assign a suitable routine to handler #
                handler;
                COMMENT if the elaboration of handler is terminated (with
                    a jump), then all elaborations taking place within the
                    considered environ, or any newer environ, are
                    terminated (since the jump can only be to a label not
                    contained within the considered call of handle);
                    otherwise, the elaboration of handler may RAISE an
                    exception not trapped within itself, or it may return
                    (whereupon the RAISE e which follows is
                    elaborated);
                    in either event some other handler is called or
                    ultimately, when the primal environ is reached (no
                    intermediate handler having been terminated),
                    undefined is called).
                COMMENT
                RAISE e
            ELSE undefined
            FI
        END ;
h)  EXCEPTION others = make exception(0) ;
i)  EXCEPTION time exhausted = new exception ;
        # the operating system proposes to terminate the elaboration peremptorily
        #
    EXCEPTION space exhausted = new exception ;
        # there is insufficient storage space to elaborate some generator or
        call #
    EXCEPTION undefined value = new exception ;
        # some value is undefined (or nil) and the further elaboration depends
        upon it #
    EXCEPTION arithmetic error = new exception ;
        # the result of some arithmetic operation cannot be computed (or is
        meaningless) #
    EXCEPTION bounds error = new exception ;
        # one of the requirements in R5.3.2.2.a.Case A or Case B or
        R5.2.1.2.a.Case B is not satisfied #
    EXCEPTION scope error = new exception ;
        # the scope requirement in R5.2.1.2.b or R3.2.2.a is not satisfied #
    EXCEPTION transput impossible = new exception ;
        # a call of undefined consequent upon one of the possible procedures
        (R10.3.1.3.b..h) returning FALSE, or upon an incompatibility of one
        of the mood fields of a file (R10.3.1.3.a), or the attempted use of
        an unopened file #
    EXCEPTION file end = new exception ;
        # a call of undefined consequent upon one of the get good procedures
        (R10.3.1.6.e,f,g) returning FALSE #
    EXCEPTION char error = new exception ;
        # a call of undefined consequent upon char error mended OF some
        FILE returning FALSE, or the suggested character being unsuitable #
    EXCEPTION value error = new exception ;
        # a call of undefined consequent upon value error mended OF some
        FILE returning FALSE #
    EXCEPTION format error = new exception ;
        # any of the calls of undefined contained in get next picture
        (R10.3.5.b) (or in do fpattern - see Commentary 12) #
    EXCEPTION abort = new exception ;
        # an exception which may be raised by the user for the purpose of
        intentionally aborting the particular-program #
```

(Further declarations such as these may appear in the **library-prelude**.)

2. Semantics

a) When, during the elaboration of a **particular-program**, an action is interrupted (R2.1.4.3.h) or the further elaboration becomes undefined (e.g. R1.1.4.3.b), and the implementation is able to detect that such a situation has arisen, then a "system exception" may be (indeed, it should be) raised (: if the situation is not detected, it might be said that an "undetected exception" has occurred, and if a **formula** containing *RAISE* is elaborated, it might be said that a "user exception" has occurred).

b) The raising of a "system exception" consists of the elaboration, in the environ of the interrupted or undefined action, of a **MONADIC-formula** (R5.4.2.1.b),
. whose **applied-operator** identifies the **defining-operator** (*RAISE*) contained in form 1.g above, and
. whose **operand** yields the value (of the mode specified by *EXCEPTION*) that was ascribed, during the elaboration of the **standard-prelude** or the **library-prelude**, to some "appropriate" **identifier**.

c) It is not further defined which such **identifiers** should be considered the most "appropriate" (since the manner in which interrupts etc. are classified and the possibilities for further action thereafter vary so much from one implementation to another). However, the **defining-identifiers** contained in form 1.i above are accompanied by comments which suggest the circumstances envisaged as being appropriate for each.

3. Implementation

It should be possible to incorporate the exception handling feature into an existing compiler by modifying only the run-time system, leaving the compiler itself untouched. The implementer has only to devise means of performing the following operations:

1. to follow the dynamic chain, starting from the stack frame of the current routine and locating, in order, all stack frames between there and the bottom of the stack, which must itself be recognizable as such;

2. to recognise any stack frame corresponding to a call of *handle*; if the implementation already provides a field within each stack frame to identify the code that is being called, this is no problem; in general, implementations can be expected already to provide some such feature, since they mostly are able to print out some identification of the active routines upon program failure; once a stack frame for *handle* has been recognised, it is a simple matter to find the value of its *traps* parameter;

3. to mark a stack frame for *handle* as having been "considered previously"; if a spare bit can be found in the stack frame, this is easy; alternatively, it would be sufficient to replace the *traps* parameter of *handle* with a flat multiple value (allowing the gartbage collector to dispose of the old parameter);

4. to invoke the implementation's normal failure action, which is usually to print a (more or less complete) list of the active routines, their local variables, etc.; this invocation should take place at the call of *undefined* at the end of *RAISE*.

Acknowledements

The underlying idea behind this proposal came from Hanno Wupper. Many other members of the WG 2.1 ALGOL 68 Support Subcommittee also contributed ideas - notably Martyn Thomas, Chris Thomson and Martin Cole.

AB49.4.2

An Exception-Handling Mechanism for ALGOL 68

by Martyn Thomas
(South Western Universities Regional Computer Centre, Bath, UK)

Objectives

This proposal is designed to provide an exception-handling mechanism within ALGOL 68 without any language changes. The mechanism should provide the following facilities:
- programmer-created traps for system exceptions;
- traps for programmer-defined exceptions.

Exception-handlers should be bound dynamically to their exceptions, to allow a handler to be set up before a library procedure is called, to handle any untrapped exceptions which are raised inside the procedure.

The Proposal

The handlers would normally be set up to trap exceptions within a closed-clause, as follows:

```
BEGIN
      on(overflow, overflow handler);
      on(bound check, bound check handler);
      . . . .
      . . . .
      C body of the closed-clause C
      . . . .
      . . . .
EXIT
overflow handler:
      C handle overflow exceptions C
EXIT
bound check handler:
      C handle bound check C
END
```

User-exceptions are created by

```
EXCEPTION my exception = new exception;
raise(my exception)   CO this signals the new exception CO
```

The following are included in the standard-prelude:

a) *MODE EXCEPTION = STRUCT(INT ?unique, ...);*
b) *PROC new exception = EXCEPTION:*
   C guaranteed to yield an unique *EXCEPTION* value *C;*
c) *PROC raise = (EXCEPTION e) VOID:*
   C signal *e C;*
d) *PROC reraise = VOID:*
   C raise the current exception outside the range of the current handler. *raise* the standard exception *no exception* if there is no valid current exception *C;*
e) *PROC on = (EXCEPTION e, PROC VOID l) BOOL:*
   C set a trap for *e*, calling *l* when *e* is *raised*, calling *system action* if *l* returns. Return *FALSE* if *e* may not be trapped (for example, *'job cancelled by operator'*); otherwise, return *TRUE C;*

f) *EXCEPTION*
overflow = new exception,
bound check = new exception,
underflow = new exception,
. . . .
. . . .;

g) *PROC system action = VOID:*
C handle untrapped exceptions or any return from a user exception handler *C;*

Discussion

When an exception is raised, either by the run-time system detecting some exceptional condition or by the programmer calling *raise*, a search is made for the most recently established, in-scope handler for this exception. Control then passes to the handler's *PROC VOID* (which will most commonly be a procedured jump). If there is no valid handler, or if the *PROC VOID* returns, the standard postmortem action is invoked by *system action*.

The main implementation problems with this proposal result from the attempt to define a sensible range for a handler (i.e. when does a handler established by a call of *on* cease to be linked to its exception, so that a handler established at a lower block-level is reactivated?). It is clearly desirable that nested handlers are permitted and safe, so that a library procedure can trap its own exceptions without destroying the handlers set up in the environment embracing the call. For safety, and for excellent philosophical reasons, it should be impossible for a mistake in the called procedure to destroy the embracing handlers, so the reestablishment of the old environment must be automatic once the range of the handler is left.

Unfortunately this implies some system action at the end of any range which includes a call of *on*, and this probably ensures that this proposal will not be adopted by any compiler currently in use. If the range of the handler is defined in a way which is easier to implement without altering the runtime systems of existing compilers, the behaviour of nested handlers becomes far harder for the programer to understand and control.

Nevertheless, this proposal seems to be the simplest to use and to understand in the common cases; it handles the exception conditions in transput in a straightforward way and, as a final bonus, it provides a wholly acceptable use for the completer *EXIT*. Comments on this proposal would be welcomed.

Acknowledgement

This proposal arose from discussions with members of WG 2.1 ALGOL 68 Support Subcommittee, notably Chris Thomson, Charles Lindsey and Lambert Meertens.

AB49.4.3

A Self-replicating Program in ALGOL 68C

by E. F. Elsworth
(University of Aston, Birmingham, UK)

After reading [AB47.4.1] on self-replicating programs, I decided to try the problem using Algol68C. The solution given for (standard) Algol 68 was the following, entered as a single line:

```
(.STRING a="(.STRING a="";print(2*a[:12]+2*a[12:])";
  print(2*a[:12]+2*a[12:]))
```

However Algol68C has two features which prevent the direct use of this solution:

i) Mode STRING is not equivalent to any ROW mode and so STRING values can't be trimmed. (But the construction 'iELEMs' can be used to obtain the ith character of a STRING 's'.)

ii) * is used as an escape character in STRING and CHAR denotations, so that to represent a literal quote character "*"" (and not """") is required, and "**" is required to represent literal *.

Nevertheless, we can still make use of the general principle behind the above solution, which is:

```
                head      tail
STRINGa="STRINGa="";print(head+head+tail+tail)";
print(head+head+tail+tail)
```

By adding some STRING-construction features to the *head* part and using these to produce a satisfactory *tail* part, I was able to find an Algol68C solution which gets round problems (i) and (ii) above. Once again this needs to be input as a single line; laid out for clarity, its form is:

```
STRING a="STRING a=",
       x="***"xt=,",
       t="*";print(a,2ELEMx,a,2ELEMx,6ELEMx,
                   3ELEMx,5ELEMx,2ELEMx,1ELEMx,1ELEMx,x,2ELEMx,6ELEMx,
                   4ELEMx,5ELEMx,2ELEMx,1ELEMx,t,t)";
       print(a,2ELEMx,a,2ELEMx,6ELEMx,
             3ELEMx,5ELEMx,2ELEMx,1ELEMx,1ELEMx,x,2ELEMx,6ELEMx,
             4ELEMx,5ELEMx,2ELEMx,1ELEMx,t,t)
```

Note that in Algol68C a program is a series, not a closed clause, and that 'print' only ever requires single brackets.

I now decided to test this using our local Algol68C compiler, but ran into a snag - it will not accept source lines longer than 132 characters! Seeing no way to get an Algol68C solution down to 132 characters, and always being keen to test theory in practice, I now had to tackle the problem of constructing a multi-line self-replicating program. Here is my solution:

```
1. OP% =(INTi,STRINGs)CHAR:iELEMs;STRINGa="OP% =(INTi,STRINGs)CHAR:iELEMs;
1' STRINGa=",x="***"xt=,",t="*
2. *";print(a,2%x,a,2%x,6%x,3%x,5%x,2%x,1%x,1%x,x,2%x,6%x,4%x,5%x,2%x,1%x,
2' newline,1%x,t,1%x,newline,t)*
3. ";print(a,2%x,6%x,3%x,5%x,2%x,1%x,1%x,x,2%x,6%x,4%x,5%x,2%x,1%x,
3' newline,1%x,t,1%x,newline,t)
```

Notes: i) 1 and 1´, 2 and 2´, 3 and 3´ above should be entered as the single lines 1, 2 and 3 of the program.

ii) To continue a string denotation onto a new line, Algol68C requires a * before the end-of-line.

iii) The operator declation defining % as equivalent to ELEM is necessary to save enough characters to get the 'print(....)' part into one line. Even if a solution with this part split can be found, it will certainly be significantly more complicated. (The space after OP% is necessary to terminate the operator symbol).

This program has been successfully compiled and tested using our Prime Algol68C system — Can anyone come up with a shorter Algol68C solution whose correctness can be demonstrated by actual compilation and execution?

Reference

AB47.4.1 C. Thomson, 'Self-Replicating and n-cycle Programs', pp19-20, Algol Bulletin no. 47, August 1981