ISSN 0084-6198

# Algol Bulletin no. 46

DECEMBER 1980

CONTENTS



LIBRARIANS please take note.

There is a microfiche enclosed with this issue. Please file it away wherever you keep microfiches, and write your catalogue number for it here:

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



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

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

The Editor of the ALGOL BULLETIN is:

Dr. C. H. Lindsey,
Department of Computer Science,
University of Manchester,
Manchester, M13 9PL,
United Kingdom.

Back numbers, when available, will be sent at $4 each. However, it is regretted that only AB32, AB34, AB35, AB36, AB38, AB39, AB40, AB41, AB42, AB43 and AB45 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.

## AB46.0 EDITOR'S NOTES.

Now, as you will see, we have become just like any manufacturer of Cornflakes for, enclosed with this issue, wou will find your Free Plastic Giveaway. It takes the form of a Microfiche containing a full set of ALGOL 68 Syntax Charts. In future issues we hope to bring you the full ALGOL 68 Report, and other related documents - all on microfiche.

Do not be put off by the IFIP copyright notices on the Fiche. These same diagrams also appear in the latest reprint of the Informal Introduction (see AB46.1.4 below), and we had to protect North Holland's rights in the matter. If you want to blow them up so as to produce Students' Handouts, Projector Slides, or even Wall Charts, please feel free to do so - but not for profit, please.

At this point, I usually make my plea for material for the next issue. Please consider it made. If you get the impression that too much of this

issue is the Editor's personal contribution, then you know just how to prevent it from happening again.

## AB46.1 Announcements.

### AB46.1.1 Professor Klaus Samelson.

Klaus Samelson died on May 25, 1980 after suffering from pancreas cancer. He was one of the four members of the GAMM delegation to the Zurich 1958 ALGOL conference and has now followed Rutishauser who died in 1970.

Klaus Samelson was born on December 21, 1918 at Strasbourg. Under the Nazi regime he was subjected to severe discrimination. After the war, he studied mathematics, physics and astronomy at Munich University and received his Ph.D. in 1951. In the 50's he worked with the Munich computer pioneers Hans Piloty and Robert Sauer. He became professor of mathematics at Mainz University in 1959, at Munich Technical University in 1963, where he changed to a chair of mathematics and computer science in 1974.

Among his many contributions to computer science and especially to ALGOL the introduction of the block concept, for example, has found wide recognition; others have been submerged in committee work, patent applications and group research. All his life he was admired by friends, colleagues and students for his outstanding analytical talent. His friendly and honest personality impressed also his adversaries. We mourn the loss of an eminent mathematician and computer scientist.

F.L.Bauer.

### AB46.1.2 Mathematisch Centrum - Change of address.

The Mathematisch Centrum, Amsterdam has moved. Its new address is:

Mathematisch Centrum,
Kruislaan 413,
1098 SJ AMSTERDAM,
The Netherlands.

Tel. (020)5929333
Telex 12571

### AB46.1.3 International Conference on ALGOL 68.

This Conference is to be held on March 30-31, 1981, at the Ruhr-University, Bochum, under the sponsorship of the IFIP WG2.1 Subcommittee on ALGOL 68 Support and the Computing Centre of the University. The Call for Papers was recently circulated to all AB subscribers.

The Topics of the Conference are to include all aspects of ALGOL 68; its implementation and use, future developments and, especially, its use for teaching purposes.

Full details may be obtained from:

Prof. Dr. H.Ehlich,
Rechenzentrum der Ruhr-Universitaet,
Postfach 10 21 48
D-4630 BOCHUM 1
Federal German Republic.

AB46.1.4 Informal Introduction to ALGOL 68.

This book (authors C.H.Lindsey and S.G.van der Meulen) has recently been reprinted, both in hardback and paperback. The new printing corrects various minor errors in the previous (Revised) edition, and also includes a set of Syntax Charts in a new Appendix.

The Publisher is North-Holland Publishing Company.

The Hardback edition is ISBN 0-7204-0504-1 price HFl 75
The Paperback edition is ISBN 0-7204-0726-5 price HFl 40

In the U.S.A and Canada, the distributors are Elsevier/North-Holland Inc., 52, Vanderbilt Avenue, New York, NY 10017, and the prices are US $36.50 and $19.50.

AB46.1.5 International Symposium on Algorithmic Languages - Call for Papers

CALL FOR PAPERS

International Symposium on

ALGORITHMIC LANGUAGES

Organized by the Mathematical Centre, under the auspices of Technical Committee 2 of the International Federation for Information Processing.

OCTOBER 26-29, 1981 AMSTERDAM, THE NETHERLANDS.

The Algorithmic Languages Symposium is organized as a tribute to Professor A.van Wijngaarden on the occasion of his retirement as director of the Mathematical Centre. Professor van Wijngaarden has played an important role in the history of IFIP and, more specifically, in that of TC2. He is well known for his contributions in the area of programming language design (ALGOL 60, ALGOL 68, two-level grammars).

The following list of possible topics illustrates the intended scope:

Programming languages (design, formal definition, implementation, use, environment, importance of style and notation, language features),
programming methodology (design methods, modularity, systematic program development tools),
program correctness (specification, validation, correctness preserving transformations).



CHAIRMAN PROGRAM COMMITTEE
Prof. dr. M. Paul
Technische Universitat Munchen
Institut fur Informatik
Arcisstrasse 21
D-8000 MUNCHEN 2
Federal Republic of Germany

CHAIRMAN ORGANIZING COMMITTEE
Prof. dr. J.W. de Bakker
Mathematical Centre
Kruislaan 413
1098 SJ AMSTERDAM
The Netherlands

TO SUBMIT YOUR PAPER:

1. Send five copies of double-spaced manuscripts, not exceeding 5000 words, by FEBRUARY 1, 1981, to the secretary of the Program Committee:
   Dr.J.C.van Vliet
   Mathematical Centre, Kruislaan 413
   1098 SJ AMSTERDAM, The Netherlands
2. Authors will be notified of acceptance/rejection by JUNE 1, 1981.
3. After acceptance of his/her paper the author will be requested to sign the IFIP copyright transfer form, in connection with his/her contribution.
4. The deadline for submission of the final paper is AUGUST 1, 1981.
5. The proceedings, published by North-Holland Publishing Company, will be available at the symposium.

PROGRAM COMMITTEE MEMBERS

A.Colmerauer (France)
O.J.Dahl (Norway)
R.B.K.Dewer (USA)
E.W.Dijkstra (The Netherlands)
A.P.Ershov (USSR)
C.A.R.Hoare (UK)
G.Kahn (France)
C.H.A.Koster (The Netherlands)
B.Listov (USA)
M.Paul (FRG)
J.E.L.Peck (Canada)
W.L.van der Poel (The Netherlands)
S.A.Schuman (USA)
M.Sintzoff (Belgium)
T.B.Steel,Jr. (USA)
W.M.Turski (Poland)
J.C.van Vliet (The Netherlands)
N.Wirth (Switzerland)
N.Yoneda (Japan)

FURTHER INFORMATION

The symposium will be held at the Free University, De Boelelaan, Amsterdam. Details about lodging and further items about the symposium (including the final program) will become available in Spring 1981. For further information, write to:

ALGORITHMIC LANGUAGES 1981
Mathematical Centre, Kruislaan 413
1098 SJ AMSTERDAM, The Netherlands.

AB46.1.6 Survey of Viable Implementations.

I hope to publish, as a regular department in future issues, a survey in tabular form of "viable implementations" of ALGOL 68. I define a viable implementation to be one which

1) is available (for money or otherwise) to anyone who wants it
2) is already in use on at least two sites

3) has some identifiable person who will provide at least minimal maintainance and/or periodic upgrades.

I am writing to all the implementors I know of who have offerings in this category, but this note is to invite all whom I may have missed to make their submissions. The information I need for each system includes

The hardware it runs on
The operating systems it runs under
Its principal sublanguage features
Its principal superlanguage features
Whether it contains any deviations
Whether a report is available detailing its performance under the MC Test Set (see AB44.1.2)
Whether it costs money, i.e. for nominal charges (<=$200) or is free
Who to get it from
Other useful information (e.g. separate compilation facilities, suitability for teaching, run-time efficiency, etc.).

I define a System to "deviate" if it is possible to write some program, valid and with defined meaning both in that System and according to the Revised Report, which will provide results different from those defined by the Revised Report. (A deviation which only occurs if some non-standard progmat is first invoked would, however, be disregarded.) For the tabular presentation, this information must be very brief (the Editor reserves the right to compress it as required). However, more extended articles on particular implementations will also be accepted from time to time (as in AB45.4.1, AB45.4.2, and AB46.4.1 in this issue).

AB461.7 Back numbers.

We can supply back numbers of most issues of the ALGOL Bulletin (see p.1 for list). We recently managed to lay our hands on some spare copies of AB36 and AB40, which had been out of stock for some time. If anyone needs either of these issues to complete his set, they are available from the Editor at the usual back number price $4 or £1.80 each.

AB46.2 Letters to the Editor.

AB46.2.1 A Self-replicating program in ALGOL 68.

South West Universities Regional Computer Centre
University of Bath
Claverton Down
Bath BA2 7AY
United Kingdom

Dear Dr Lindsey,

In response to a recent challenge in the SWURCC newletter to produce a self-replicating program in Algol 68, I have received the following program:

```
(STRINGa="(STRINGa="";print((2*a[:10],2*a[10:])))";print((2*a[:10],2*a[10:])))
```

I hope you will agree that a solution this elegant deserves a wide audience. The program was written by Richard Wendland, who is one of the team currently implementing Algol 68 on Honeywell's Multics.



As you will see, the program depends on its layout to the extent that it must be a single line, and the explicit character position in the print must identify the inner quote within the string.

Yours sincerely,

Martyn Thomas
Deputy Director

AB46.2.2 ALGOL 68S Compiler.

Dept. of Computer Science,
University of Manchester,
MANCHESTER M13 PPL,
United Kingdom.

Dear Editor,

I announce the availability of an ALGOL 68S compiler for CDC Cyber and 7600 machines. It is primarily intended for teaching, insofar as it compiles fast and runs slow, has good diagnostics and checks everything that is checkable (except overflow of course, CDC machines being what they are). The language implemented is the official ALGOL 68 Sublanguage (see SIGPLAN Notices 12 5 May '77, or Appendix 4 of the Informal Introduction), with the addition of heap generators.

To obtain the next Release (due about January), send me a 7-track magnetic tape. There is no charge at present, although this may not always remain so. This release will be more or less complete, the principal likely omissions being binary transport and complex arithmetic.

The compiler is based on the PDP11 compiler written by P.G.Hibbard at Carnegie-Mellon. Although it is written in PASCAL, it is not yet in a portable state - so please do not ask for it on other machines at the present time.

C.H.Lindsey.

AB46.4 Contributed Papers

AB46.4.1 ALGOL 68 Implementations - The ICL 2900 Compiler

The ICL 2900 Series Algol 68 compiler is the result of a collaborative project involving the South West Universities Regional Computer Centre, Oxford University Computing Service, and the Royal Signals and Radar Establishment, and is based on the portable RS compiler front end produced by RSRE (described in RSRE Technical Note 802).

The compiler is released by ICL as a standard 2900 Series product, and has been written to use the same compile-time environment and run-time diagnostic system as other 2900 compilers. It requires a main store quota of 350 Kb.

The language implemented is based on the Algol 68 Revised Report with extensions and restrictions.

The principal language extensions are:

- a powerful modular compilation system which provides full compile-time and run-time checking of the interfaces between different modules.

- 'vectors', special rows which always have a lower bound of 1 and are restricted to one dimension (eg. VECTOR [n] INT v). As well as providing more efficient access than normal rows, this allows communication with programs written in other 2900 languages.

- 'indexable structures' are a generalisation of mode BYTES and group together a fixed number of objects of any specified mode, allowing the facility of indexing without the usual array overheads. (eg. STRUCT 7 REAL s; s[4]:=2.0) Indexable structures may be coerced to vectors, and vectors to rows.

- a FORALL facility allows easy and efficient access to all the elements of an array in turn.

- by default, boolean AND and OR are optimised so that the right operand is only evaluated when necessary. Full evaluation may be requested by a user parameter to the compiler.

- there are no restrictions on the use of transient names.

- priority declarations may be omitted in which case a priority of 1 is assumed.

The principal language restrictions are:

- identifiers must be declared before they are used, except for labels and simply recursive procedures.

- semaphores and parallelity are not implemented.



- no more than two LONGs or one SHORT is allowed on REAL, INT, BITS and COMPL.

- scope checking is not implemented.

- to simplify debugging, SKIP causes a run-time fault if required to yield a non-VOID value.

- for efficiency reasons, the elements of arrays are never copied in identity declarations or equivalent positions.

- BYTES is defined as STRUCT 4 CHAR (see language extensions). LONG and SHORT BYTES are not available.

The compiler has been successfully validated on a subset of the MC (Amsterdam) test set, taking into account the language restrictions described above.

Commercial details such as the terms and conditions of distribution may be obtained through normal ICL channels. It should be noted that special concessionary arrangements apply to all establishments funded by the Computer Board for Universities and Research Councils if they order the compiler soon.

Technical enquiries may be addressed to

Mr Gavin Finnie,
SWURCC,
University of Bath,
Claverton Down,
Bath BA2 7AY.

AB46.4.2

## More Remarks on Algol 68 Transput

by H. Ehlich and H. Wupper, Ruhr-Universitaet Bochum

The last issue of Algol Bulletin contained some critical remarks on Algol 68 transput written by Hanno Wupper in 1978 (AB 44.4.1). In the same issue readers can find the announcement of a new transput model (AB 44.1.1).

An early version of that same model has in fact been the basis of a transput system implemented by the authors, which is now in use at 16 TR440 installations. Other implementors might be interested in our experiences.

The TR440 is a big scientific machine in use at several scientific institutions in Germany. Its operating system supports in a completely orthogonal way several kinds of files - most common those being mentioned in the second paragraph of RR 10.3.1.6. (sic!).

All files can easily be handled by command language and by all programming languages provided by the manufacturer. The main disadvantage of the operating system is that internal interfaces are accessible by assembly language only; therefore all transput systems are vast assembly programs (except, of course, for BCPL where the situation is similar to the one described in this article, only much simpler).

To make Algol 68 available on this machine, we wrote a code generator and run-time system for the portable Algol 68 C system developed and maintained at the University of Cambridge. Algol 68 C is by now, if you carefully avoid some "extensions", a very fine subset of Algol 68 , the most important restrictions being: no FLEX, LONG, SHORT, PAR, FORMAT; STRING a primitive MODE (in several senses).



The Algol 68 C bootstrap-kit provides the machine independent parts of a restricted transput system which is useful for a quick bootstrap but would make Algol 68 a second class language on our machine.

When we intended to provide a more complete transput, we were faced with three questions (cf. AB 44.4.1):

A: How to find out the intended meaning of Algol 68 transput from the Report?

B: What to do with the features more orthogonal in our operating system than in the Report?

C: How to arrive at a both correct and efficient implementation easily?

Just in time H. Wupper met J.C. van Vliet who, at that time, had finished a first version of his model now described in AB 44.1.1. Now Questions A und C could be answered:

Answer A: Not necessary, if the model is used as the basis of an implementation.

Answer C: Use the model. If efficiency is insufficient, replace some small parts by assembly programs.

A fruitful cooperation started: From Hans van Vliet we got two boxes of punched cards, constituting the machine-independent part of Algol 68 transput, written in Algol 68 . (During the journey from Amsterdam to Bochum the second box poured its contents all over the car. Luckily only formatted transput, not being supported by the Algol 68 C compiler anyhow, was afflicted.)

The program could be compiled after a few simple changes mainly of lexical nature (the compiler could not handle blanks in denotations). The lack of FLEX and proper STRING was no problem as the model does not make use of them.

We now had not to find out the intended meaning of complete Algol 68 transput (e.g. when to call which event routine during transput of values of a certain mode) but only of a few routines, the "buffer primitives", to be provided by us.

With these, in the beginning we had some difficulties. In the course of time, however, and after some feedback, Hans van Vliet could simplify this interface a good deal and provide a clear definition.

In the model, a channel more or less corresponds to a certain access method and consists of a set of primitive routines. For each file-type, physical device etc. it is easy (probably in all operating systems), to provide the appropriate routines. If, later, a new class of physical or logical devices has to be made accessible by Algol 68 , simply another channel is added.

In our case, a considerable part of these primitives could even be written in Algol 68 , though, of course, not machine-independently and with inclusion of several "code segments".

On the other hand it seemed useful to rewrite a small portion of the really time-critical routines in assembly language. Here also the model proved extremely helpful: the logically complex "conversion routines" whole, fixed, float and some others could be hand-translated without much thinking.

During the last years the model has been discussed at lenght in the Subcommittee on Algol 68 Support, also with respect to our Question B. The final version now is less restrictive than the Report and gives hints where the implementor might take greater freedom. As the official language was not to be changed essentially by the model, however, some of the criticisms of Part II of AB 44.4.1, "Algol 68 transput considered insufficient", remain valid.

Apart from that it has been proved that everybody who can make use of an Algol 68 compiler has a good chance to provide the full official transput without great effort.



## Comments upon the proposals for Modules and Separate Compilation Facilities for Algol 68.

A.J. Cowling (Department of Computer Science, University of Sheffield, Sheffield S10 2TN).

A specification [1] has recently been released for facilities to be added to Algol 68 to provide for the definition and use of modules, and to provide for the separate compilation of segments of program. The object of these facilities is to simplify the construction of large programs by breaking them down into segments (called packets), which can be produced, one at a time, using any appropriate combination of the top-down and bottom up methods of working. The purpose of this paper is to suggest that there is an aspect of these specifications which, measured against this objective, is unsatisfactory, and a way of improving this aspect will be suggested. The changes which would be required will then be developed into the outline of an alternative approach to the provision of the required facilities. Most of the discussion in the paper will centre round the examples provided in the "Informal Description" section of the specification [1].

The starting point is provided by the example given to illustrate the statement that "ACCESS is to be regarded primarily as a mechanism for permitting PUBlicised indicators to be made visible:". This example, taken exactly as it stands, is a rather unrealistic piece of code, and the situation which it illustrates is perhaps better represented by the following, which will be called example 1:

```
ACCESS STACK
   (push(1); push (2);
        C etc., followed by several pages of code C
        (PROC push = C something else C, pop = C something else C;
             C several more pages of code C
             ACCESS STACK (push(3); print (pop);
                  C and more code C);
             C and more code C
        ); C and still more C
     pop; pop
     )
```

The unpleasant feature of this is that the two occurrences of the symbol ACCESS are physically separated by a large amount of program, and yet the effect of the second, or inner, one is critically dependent on the presence of the first, or outer, one; that is, it would have a different effect if the outer one were not there. Thus a programmer, (possibly modifying a program originally written by someone else) might suppose that the effect of inserting the inner ACCESS clause (assuming that it had not been there originally) would be to obtain a stack, as defined by the module being ACCESSed. In the absence of the outer ACCESS clause this would be correct, but what is actually obtained, as a consequence of its presence, is that remainder of the original stack which has been left unused by the intervening pages of code - unfortunate if they happen to have already filled it.

In defence against the above, it may be argued that the example exaggerates the number of pages of code that would normally appear in one packet. However, the alternative to this could well be that the inner ACCESS clause is contained within an actual-hole which is then to be stuffed into a formal-hole that is contained within the outer ACCESS clause. Then the two occurrences of ACCESS would be in different packets, even though the significance of one was critically dependent on the presence or absence of the other, which would appear to be an even more undesirable situation.

An alternative argument might well be that, because this sort of situation can arise, the module STACK is therefore badly constructed, and it is interesting to observe that it could not happen with the module STACKS which is presented as the next example in the specification [1]. However, while this may indicate that in some way STACKS is a 'better' module than STACK, it does not alter the fact that modules such as STACK can be written, and in some circumstances may have to be written, and so the problem described above is likely to arise.

It would appear that the solution to this problem is to enable the programmer, writing an ACCESS clause, to specify explicitly whether the corresponding module is to be invoked (and then revoked again), or whether it is sufficient for indicators (which must have been PUBlicized in outer ACCESS clause for this module) to be made visible again. That is, the distinction between a module,



and the set of indicators that it PUBlicizes (hereafter called the layer which it reveals, although the term locale might be more accurate) is made much more strongly, and the two stages of firstly invoking the module to reveal the layer, and secondly accessing the layer, are treated as being separate. In a sense this is analagous with the distinction that is already made between the deproceduring of a procedure and the subsequent use of the value that is returned by it, and this analogy is taken further by suggesting that it be possible to name the layer revealed by the invocation of a module, in the same way that values returned by procedures can be named within definitions. (This analogy must not be pressed too far, since layers have to be nested one within another; thus there cannot be a "collateral invocation" to be analagous to the collateral elaboration of a joined set of definitions, useful though such a concept might be).

Given such an approach, there will then be two alternative versions of example 1; in one the module will be explicitly re-invoked, and in the other it will (equally explicitly) not be reinvoked, and the fact that the programmer has a choice between these two means that the undesirable situation described above can be avoided. Precisely how these two versions are to be written down could be a source of much argument over trivia; one possibility (based on the analogy between a module revealing a layer, and a procedure returning a value) is used below to illustrate the two possible versions of the original example from the specification [1]. First we have the version in which the module is only invoked once, which forms example 2:

```
ACCESS LAYER OLDSTACK = stack
     (push(1); push (2);
          (PROC push = C something else C, pop = C something else C;
          push; pop;
          ACCESS OLDSTACK (print(push(pop)) # prints 2 #)
          ); pop; pop
     )
```

(It will be noted that the analogy with procedures suggests that module indications should be represented by TAGs rather than TABs). Example 3 is then the version in which the module is invoked twice:

```
ACCESS stack #=  since there is no need to name the layer #=
    (push (1); push (2);
         (PROC push = C something else C, pop = C something else C;
         push; pop;
         ACCESS stack #= this one does not need to be named either #=
              (print (push(pop)) #= fails under flow #=)
         ); pop; pop
    )
```

While this effect could be obtained within the module scheme as specified in [1], by turning the code containing the inner ACCESS clause into a procedure which is defined outside the outer ACCESS clause (a technique illustrated in [1] by the "carefully chosen confusing example"), the fundamental problem still remains that it may be necessary to examine a great deal of code in order to determine the precise effect which an ACCESS clause has.
In particular it does not make life any easier for the maintainer of code, if what should be the simple insertion of an inner ACCESS clause into a piece of code will actually mean converting part of it into a procedure in this way in order to force the reinvocation of the accessed module. Furthermore, if the point where the inner ACCESS clause is to be inserted is part of an actual-hole, while the outer ACCESS clause is in the packet containing the corresponding formal hole, then the adoption of this technique would mean that a change which should have been confined to the actual-hole packet will also involve changing the formal-hole packet, which makes this technique seem distinctly unsatisfactory. It is therefore claimed that there is a definite need for the change being suggested.

However, this change does have other effects, and in particular it removes the "calculated ambiguity" of the ACCESS construction, whereby it is possible to write an ACCESS clause for one module within the body of a second module, and then have two different invocations of the second module, of which one will cause the first to be reinvoked, while the other will not. Instead, the result of the change being proposed is that a given ACCESS clause would either be written so that it always caused a fresh invocation, or it would be written so that it never did. (This would then make possible another change, in that modules could be allowed to have parameters, since it would always be known



whether or not invocation of the module, and therefore specification of the corresponding actual parameters, was required. It hardly seems necessary to claim that this consequence would be a good thing, but anyone who is not convinced is invited to formulate a good reason why a stack should always have to have size 100.)

Unfortunately, there are situations where this "calculated ambiguity" of meaning of the ACCESS clause is extremely useful. One of the examples in the specification [1] is a library of modules which can be summarised as in example 4:

```
MODULE MATMODE    = DEF C something C FED;
MODULE MATRICES   = ACCESS PUB MATMODE DEF C something C FED;
MODULE VIBRATIONS = ACCESS MATRICES, PUB MATMODE DEF C something C FED;
MODULE STRESSES   = ACCESS MATRICES, PUB MATMODE DEF C something C FED;
```

An example of a particular program using these is also given, and can be summarised as example 5:

```
ACCESS VIBRATIONS, STRESSES
BEGIN C.....C;
      ACCESS MATRICES (C.....C);
      C.....C
END
```

If such a library was to be reexpressed in the forms so far suggested, then it would be necessary to decide for each ACCESS clause whether the modules to be ACCESSed are to be explicitly invoked, or whether it can be assumed that this has already been done, so that the ACCESS will simply be a use of a previously defined layer. In so far as it is possible, the latter is obviously preferable, since it will avoid unnecessary invocations; however, this does mean that the particular program must then arrange for the explicit invocations not only of VIBRATIONS and STRESSES, but also of MATMODE and MATRICES, and furthermore it must ensure that the name given to the LAYERs defined by MATMODE and MATRICES are those names by which VIBRATIONS and STRESSES actually refer to them. Thus the particular-program would have to be concerned with a great deal of messy detail concerning the internal workings of the library (or set of libraries) which really should not affect the interface between particular-program and libraries at all.

Clearly, in this sort of situation the "ambiguity" of the ACCESS clause, whereby invocations are constructed where they are needed, is extremely useful, although the fact that invocations may sometimes not be constructed when they are wanted can equally be dangerous in other situations. At first sight it might appear that there is actually a need for two different constructions, one for each of these two situations; in fact it will be demonstrated that they can both be regarded as special cases of one concept which encompasses not only modules and ACCESS chances, but also the construction of holes and nests as well.

This unifying concept is the concept of specifying explicitly the interface between two clauses which are to be nested one within the other. The interface in this context consists of the set of properties which the outer clause makes available for identification to the inner one, and in Algol 68 as originally defined [2] is always specified implicitly, in that the interface "seen" by the inner clause is the same as that defined by the outer clause. What is now being proposed is that a clause can specify, either wholly or in part, the interface which must be provided by any clause into which it is to be nested, (which will hereafter be called the outward interface) and all the indicators specified in that outward interface will then be visible within that clause. Correspondingly, a clause will be able to specify one or more points into which other clauses may be nested, and will then also specify the interface which those clauses will see (hereafter referred to as the inward interface). It can be seen that the function of the present ACCESS clause is effectively to specify an outward interface for the enclosed-clause, and much the same is true of the EGG definition. What is now being proposed is that these two should be expressed by a single construction, namely the outward interface specification; corresponding to this will be a second construction, viz. the inward interface specification, and the compilation system will then be expected to match inward and outward



interfaces (constructing module invocations where necessary) when a number of clauses (possibly compiled separately) are to be assembled into a single particular program (by nesting them, one inside the other, in an appropriate fashion).

An interface will thus be constructed from two types of element: one will be a named layer, and the other will be a complete named nest. For the inward interface, there will then be two cases: one will be that a clause will define a single point wherein other clauses may be nested, and that at this point it makes available to the nested clause a single named layer (in addition to the other properties which would be available to that clause anyway); this case corresponds to the definition of a module, and indeed such clauses will continue to be described as modules. The other case is that a clause may define any number of points wherein other clauses may be nested, and at each of these points the nested clause has available to it a named nest; thus each of these points will correspond to a formal-hole. Such a clause will hereafter be referred to as a segment, and this term is to be understood as including those clauses which do not specify an inward interface at all; thus all EGGs which are not modules will be segments (indeed, it seems not unreasonable to regard modules as a special case of segments, so that all EGGs will be segments). Furthermore, a particular program will be a segment; it will be converted into a program (using the term in the sense in which it is defined by the syntax of the language) by being "assembled" with a suitable collection of segments which are to be nested into it, and into which it is to be nested (the latter including, of course, an appropriate particular prelude and particular postlude). These other segments will have been constructed in the form of segment definitions (of which module definitions are a special case), although (with the exception of modules) the names that

they are given in these definitions will be irrelevant, since the way in which segments are to be nested together is determined purely by the matching of the interfaces. Thus it is envisaged that a minimum packet for presentation to the compiler will be either a segment or module definition, or a segment (i.e. particular program); in the latter case it is implied that successful compilation is to be followed by assembly of a complete program (effectively link editing). This will presumably require for practical purposes some way of specifying where these other segments (in their compiled form) are to come from, but the mechanism for doing this will obviously depend on the requirements of local operating systems, etc. However this mechanism will not form part of the program text as such, and so no suggestions for its specification (or even a suggestion that it be specified in a standard fashion) will be made.

Clearly the practical operation of this concept depends on the way in which specified inward and outward interfaces are to be matched together when clauses are nested one within another. The simplest case of this interface matching is a clause which has an outward interface specified as a single named nest, and this clearly can only be nested into a point in a clause where the inward interface is specified as being a nest with the same name. A restriction which follows from this is that, within a single program (i.e. collection of segments), no two inward interfaces that are specified as nests may have the same name. It may also be observed that a segment which specifies no outward interface (such as a particular-program constructed according to the original specification of the language [2]) can be regarded as having an implicit outward interface which will be a nest with the same name as that defined by the inward interface of a segment containing an appropriate particular-prelude and particular-postlude.

The next simplest case is that of a clause which has an outward interface which is specified as being the name of a layer (i.e. a construction



corresponding to the ACCESS clause as proposed in [1]). Such a clause can only be nested within a point where the inward interface contains this named layer - usually implicitly (i.e. the clause forms part of another which has a revelation of this named layer in its nest, and so on); an illustration of this situation is the clause beginning ACCESS OLDSTACK in example 2. The revelation of this layer may have arisen in one of two ways, of which the most obvious is that a module has been explicitly invoked in an outer clause, and that the layer which the module reveals in its inward interface has been given this name; this is the case in example 2. The second way in which a revelation of a named layer may arise is only applicable to layers which are revealed by modules that do not have parameters, and it corresponds to the mechanism which is required for extracting modules from libraries, as illustrated by examples 4 and 5; it is proposed that in such a case an "automatic" invocation of the module should be constructed (by a mechanism analagous to coercion) around the outermost clause which explicitly requires that layer, and of course the layer so revealed is given the name defined in the module's inward interface. Thus, in example 5 the clause BEGIN.... END would be surrounded by automatic invocations of the modules defining the two layers VIBRATIONS and STRESSES (which would actually be nested one inside the other, but in an undefined order). These would in turn be nested inside an automatic invocation of the module revealing the layer MATRICES (and a revelation of this layer would therefore be in the nest seen by the particular program, and so would be implicitly in the inward interface at the point where the clause (....) specifies it in its outward interface). Lastly, this automatic invocation would be nested within an automatic invocation of the module revealing the layer MATMODE.

It is of course possible for an outward interface to specify both a nest and one or more layers; if revelations of any of the named layers exist within the named nest then those items can be considered as satisfactorily matched.

If not, then (with the proviso that definitions of the appropriate modules exist - either within the named nest, or as other segments which have the same named nest specified in their outward interfaces) automatic invocations of the modules will be constructed around the clause in question. As an example of what happens, it may be observed that all the segments defined in examples 4 and 5 have included in their outward interfaces an implicit reference to a nest defined by a segment containing an appropriate particular-prelude and particular-postlude (which is assumed to be the same nest for each segment). It should also be added that, for the automatic invocation mechanism to be able to operate, a restriction must be imposed that no two modules may define layers with the same name if the definition of one module is accessible from the point where the other is defined.

Finally a notation for expressing interfaces will be needed. One will be suggested here, in order that some examples can be presented, but it is not intended to be regarded as definitive. One immediate comment is that the special brackets DEF....FED are unnecessary, and will be abandoned. It is suggested that the outward interface specification be written before the clause to which it applies, preceded by the symbol ACCESS; the items which may appear in them (which would normally be separated by commas) would obviously be layer and nest names (which will be represented by TABs), or explicit module invocations (in the form module name and actual parameters, if any), and any of these items could, if desired, be renamed by preceding them with either LAYER name = or NESTname = , as appropriate (not that any use can be envisaged for the latter: it is included purely for the sake of uniformity). Since it is proposed that segment and module names be represented by TABs, then it can be seen that examples 2 and 3 are in fact expressed in this notation already. This choice of notation gives rise to one restriction, namely that no program may contain a module defining a



a layer which has the same name as any of the nests which may be defined as inward interfaces, and it is also necessary to extend the restrictions on layer names revealed by modules to cover layer names defined in outward interface specifications. These restrictions are not burdensome, although the first of them could be removed by insisting that every item in an outward interface specification be prefaced with either NEST or LAYER as appropriate. It would also seem reasonable to allow items to be separated by semicolons rather than commas if it is required to express the order of nesting of module invocations, where this would otherwise be undefined.

The inward interface specification involves rather more obvious changes from the formats proposed in [1 ], since there it appears in two radically different forms. It may be observed that the specification actually has to fulfill a number of functions, viz. to indicate that this clause will have an inward interface; to specify whether the interface(s) consist(s) of a layer, or one or more nests; to name the layer or nests; and to indicate the point(s) where the interface(s) occur(s). To achieve the first requires a construction similar to that used for the outward interface specification, and it would seem appropriate to use the symbol DEF to introduce it; this form of construction will also encompass the second and third functions if the specification is restricted to a single item of the form LAYER name or NEST list of names. It is then proposed that the actual point where an interface occurs should be treated as a unit, consisting essentially of the name of the nest or layer. It may then be necessary to either preface this with a suitable symbol, or surround it by special brackets; the latter will be suggested, consisting of ENC (short for enclose), with CNE as the corresponding right bracket.

Additionally, within a module it is necessary to specify which of the indicators are to be included in the revealed layer, and for this the PUB

notation will be used, as before. It could also be useful to extend this notation to a clause which is defining a nest (or nests) as its inward interface, although this could get tedious if only a few indicators were to be excluded from the interface - perhaps a symbol PRI (for private) could be introduced to cover this case, and it might also be necessary to allow PUB and PRI to be qualified by the names of the nests to which they applied, as well as allowing an entire nest to be specified as PUB or PRI. Similar remarks also apply to modules publicising the indicators from other modules which they require; it seems desirable to allow this to be done selectively, by means of a PUB unit within the clause itself, and so the case where all the indicators are to be made visible will be treated the same way, by requiring a unit in the body of the clause, rather than prefacing items in the outward interface specification with PUB. Thus example 4 could be written in the notation being suggested as follows (example 6):

```
MODULE matmode      = DEF LAYER MATMODE BEGIN
                        C something C;
                        ENC MATMODE CNE;
                        C and a postlude C
                      END;

MODULE matrices     = ACCESS MATMODE DEF LAYER MATRICES BEGIN
                        PUB MATMODE;
                        C something C;
                        ENC MATRICES CNE;
                        C and a postlude C
                      END;

MODULE vibrations   = ACCESS MATRICES, MATMODE
                      DEF LAYER VIBRATIONS BEGIN
                        PUB MATMODE;
                        C something C;
                        ENC VIBRATIONS CNE;
                        C and a postlude C
                      END;
```



```
MODULE stresses     = ACCESS MATRICES, MATMODE
                      DEF LAYER STRESSES BEGIN
                          PUB MATMODE;
                          C something C;
                          ENC STRESSES CNE;
                          C and a postlude C
                      END;
```

and example 5 (using the above) could remain unchanged. Alternatively, example 5 could be split into two parts, to be compiled separately, as either example 7:

```
SEGMENT outerbit    = ACCESS VIBRATIONS, STRESSES
                      DEF NEST MIDDLE
                      BEGIN C......C;
                           ENC MIDDLE CNE;
                            C......C
                      END
```

---

```
ACCESS MIDDLE, MATRICES (......)
```

or as example 8

```
ACCESS VIBRATIONS, STRESSES
DEF NEST MIDDLE
BEGIN C......C;
      ENC MIDDLE CNE;
      C......C
END
```

---

```
SEGMENT middlebit   = ACCESS MIDDLE, MATRICES (......)
```

A number of observations should be made on these examples: firstly that the use of bold rather than brief brackets is not being suggested as an integral part of the notation; secondly that there is no suggestion that modules must have postludes if these are unnecessary; thirdly that a PUB unit may come anywhere within the serial clause (but presumably before the interface unit). There are also a number of restrictions which would be needed, as in the original proposals [1]. For instance, allowing PUB to be used with individual indicators would require that no operator or mode revealed within a layer could have the same name as the layer, nor the

same name as any other layer which appeared in any outward interface specification along with that layer. (In practice it is probably simpler to insist that no nest or layer name used within a program could be the same as any other nest, layer, operator or mode name used anywhere else in that program but in some situations this might prove over-restrictive). There would also need to be a restriction prohibiting two layers in the same outward interface specification from revealing the same indicator, unless this indicator was actually defined in a third layer which they both required, so that they were only PUBlicising but not defining it (i.e. in the way that vibrations, stresses and matrices in example 6 all PUBlicise the indicators from matmode); or unless the order of the invocations of the relevant modules was defined, so that it was known which definition of the indicator was hidden by the other.

It might also be thought necessary to restrict the points at which an inward interface which is defined as a layer may occur, so as to prohibit something like example 9.

```
MODULE m        = DEF LAYER M BEGIN
                    PUB PROC p = VOID: (ENC M CNE)
                  END
```

although there could be advantages to permitting a construction like example 10:

```
MODULE m        = DEF LAYER M BEGIN
                     C a lot of setting up of definitions, etc. C;
                     IF all set up ok
                     THEN ENC M CNE; C normal postlude C
                     ELSE C print error messages, and clear up the debris C
                     FI
                  END
```

and possibly even one like example 11:

```
MODULE repeat   = (INT n times): DEF LAYER REPETITIONS BEGIN
                     FOR i TO n times DO
                        PUB INT rcount = i
                        ENC REPETITIONS CNE
                     OD
                  END
```



However, no attempt will be made here to go into detail as to how these restrictions (or the other features which have been proposed) might be defined within the syntax of Algol 68, or an extension to it. The reason for this is that the purpose of this paper has been to present concepts rather than detail, and the author is not yet convinced that the required concepts have been established sufficiently clearly to justify spending time on the detail. As an illustration of this point, it may be observed that neither the original specification of [1], nor the proposals that have been made above, are adequate for coping with the following example 12 (expressed in the notation of [1]:

```
MODULE A    =    ACCESS B DEF
                    PUB MODE AMODE = C something using BMODE C;
                    C define procedures and operators upon AMODEs, some of
                      which may need the procedures and operators upon BMODEs,
                      and PUBlicise some of these definitions C
                 FED;

MODULE B    =    ACCESS A DEF
                    PUB MODE BMODE = C something using AMODE C;
                    C define procedures and operators upon BMODEs, some of
                      which may need the procedures and operators upon AMODEs,
                      and PUBlicise some of these definitions C
                 FED;
```

This is not to say that the required effect cannot be achieved, but it cannot be done in a fashion that reflects the "natural" modularisation of the problem. Of course, it may well be that to provide even the interface mechanism that has already been proposed (to say nothing of a mechanism capable of coping with the above example 12) would involve doing so much violence to the original definition of Algol 68 that, whatever the result might be, it would not be an extension of Algol 68. In which case, it should perhaps be acknowledged that considerable time has passed since 1968, and that a need is beginning to emerge for Algol eighty-something, which would have modularisation facilities designed in from the start.

REFERENCES

1. A Modules and Separate Compilation Facility for ALGOL 68, C.H. Lindsey & H.A. Boom, Algol Bulletin 43, (December 1978) pp 19-53.

2. Revised Report on the Algorithmic language Algol 68, ed. A. van Wijngaarden et. al. Springer-Verlag 1976.

AB46.4.4

# An ALGOL 68 Indenter.

by C.H.Lindsey
University of Manchester

First let it be said that this program is an Indenter, not a Prettyprinter. What, you may ask, is the difference? Well, a Prettyprinter is a program which performs virtually a complete syntax check of a program source text, and determines its entire layout. It may insist that exactly one blank occurs at certain points (e.g. on either side of a ":="). It is very likely to insist on putting each statement on a fresh line, and to start the if, then and else parts of a conditional-clause on separate lines. If it tries to be clever and to keep these things on one line in suitable cases, it has to make some sort of value judgement as to what is "suitable", and it can only do this on the basis of the length of the items concerned. These are all areas in which the User's judgement is likely to be superior. If he chooses to put several statements on one line, it will be because they have some logical connection with each other, not because they happen to fit. By keeping small snappy conditional-clauses (especially where they are conditional expressions) on one line, the User can use his main indentation structure to emphasise his global program stategy, thus conveying to his human reader the way in which he wants his program to be vizualized. When it comes to comments, the conventional Prettyprinter is in a real quandary. How can it know that some comments are short remarks to amplify the proceeding statement (and should therefore follow it on the same line) and that some are long essays explaining what is about to happen next (which should clearly start on a line of their own)? Also, a conscientious User may well wish to leave blank lines at important divisions in his program, but what Prettyprinter can manage that?

Therefore this program is a pure Indenter. It leaves the User's line divisions strictly alone. With a few minor exceptions, it does not alter his spacing within lines. All it does is to adjust the number of blanks at the beginning of each line.

Clearly, it endeavours to align fis under their matching ifs, but many prettyprinters (and textbooks) seem to think the then should be indented more than the if:

```
if condition                          if condition
   then unit;                         then unit;
        unit;                              unit;
        unit       as opposed to           unit
   else                               else unit;
       unit;                              unit;
       unit;                              unit
       unit
fi                                    fi
```

I cannot see the sense in this. It will run out of page width much sooner and, to my taste, the condition, the then-part and the else-part are all just one level of refinement deeper than their surroundings, and hence on a par with each other. Therefore my indenter aims to achieve the second style (which we also used in the standard-prelude and the examples of the Revised Report).



Students, especially, seem to find difficulties in not putting semicolons after the last unit of a serial-clause. You know that a semicolon is a separator, I know that a semicolon is a separator, and the students should know that a semicolon is a separator (because you, and I, have told them so repeatedly). But still they get it wrong. Also, those of us who still input programs on cards, or who are affected with editors designed only to modify FORTRAN or Basic programs on a line-at-a-time basis, have often been frustrated at having to punch two lines when adding an extra statement at the end of a serial-clause (a quite common requirement, apparently). There is a simple answer to both these problems - put the semicolon at the beginning of the next line (if any). Actually, programs are very readable this way (see the examples below) as the semicolons help to delineate the various indentation levels. I have taught students to program in this way, with benefit. It helps to emphasise that they are really called "go-on-symbols" and, when developing a program in front of a class, to say "go-on" as each semicolon is written. It is also a good discipline when writing programs not to write a go-on-symbol until you are ready to write the unit to be gone-on to. Then when you have run out of units to write, you are ready for your closing symbol immediately. The system works best if you always write closed-clauses (i.e. in general the complete program or the body of a procedure) in the brief style, with the "(", the ";"s and the ")" in vertical alignment.

This Indenter is designed to facilitate this style. It cannot force you to put your go-on-symbols first, but if you do it will align them nicely (if you don't, it will still produce an acceptable indentation). It is also pleased if you put and-also-symbols (commas) at the start of lines in displays, data lists, etc. Here is an example program written in both styles, and indented by the program.

```
( .LOC .INT N
; READ(N)
  # PRINT THE PASCAL TRIANGLE
    UP TO LAYER N #
; .LOC [0:N%2] .INT LINE
; .STRING GAP = 3*" "
; LINE[0] := 1
; .FOR I .TO N
  .DO [] .INT OLDLINE = LINE[ :(I-1)%2 0]
    ; .FOR J .FROM 1 .TO (I-1)%2
      .DO LINE[J] := OLDLINE[J-1]+OLDLINE[J] .OD
    ; LINE[I%2] := LINE[(I-1)%2]
    ; .FOR J
      .FROM I-N
      .TO 2*(I-1)
      .DO PRINT((
                ( J<0 .OR .ODD J
                ! GAP
                ! WHOLE(LINE[
                          ( J<I
                          ! J%2
                          ! I-1-J%2
                          )]
                      , -3)
                )
              ))
      .OD
    ; PRINT(NEWLINE)
  .OD
)
```

```
.BEGIN
    .LOC .INT N;
    READ(N);
    .COMMENT PRINT THE PASCAL TRIANGLE
        UP TO LAYER N
    .COMMENT
    .LOC [0:N%2] .INT LINE;
    .STRING GAP = 3*" ";
    LINE[0] := 1;
    .FOR I .TO N
    .DO [] .INT OLDLINE = LINE[ .(I-1)%2 0];
        .FOR J .FROM 1 .TO (I-1)%2
        .DO LINE[J] := OLDLINE[J-1]+OLDLINE[J] .OD;
        LINE[I%2] := LINE[(I-1)%2];
        .FOR J
        .FROM I-N
        .TO 2*(I-1)
        .DO PRINT((
                  .IF J<0 .OR .ODD J
                  .THEN GAP
                  .ELSE WHOLE(LINE[
                                ( J<I
                                ! J%2
                                ! I-1-J%2
                                )],
                              -3)
                  .FI
                ))
        .OD;
        PRINT(NEWLINE)
    .OD
.END
```

The Indenter is written in PASCAL (for my own convenience - it needs to run efficiently on our machine since we put lots of students' programs through it). It is easily translateable into ALGOL 68. Our operating system likes to put line numbers at the start of program lines. If the source text starts with a digit, it therefore assumes the program is line-numbered. It assumes point stropping (.BEGIN). Three constants are defined. 'smallindent' is best left at 2, but you might like to experiment with 'midindent' (but 2 seems about right) and 'largeindent' (4 seems to work well). Each "opener" ((, if, case, etc. and starts of comments) increases the indentation by 'smallindent' (for brief-openers) or by 'largeindent' (for bold-openers) and each "closer" (), fi, esac, etc. and ends of comments) decreases it again. "Middlers" (!, then, else, in, out etc. - and also exit, semicolon and comma) leave the indentation alone, but align themselves beneath their openers (assuming they are at the start of a line, of course). A phrase that extends over a line boundary acquires an extra 'midindent', which is maintained until the next middler or closer. At least one space is insisted upon (strictly, 'smallindent-1' spaces) after each semicolon or comma (mainly to ensure good vertical alignment at the start of lines, but it is a good discipline anyway), and also after any "(", "!" or "#" when at the start of a line. The "string break" convention for string-denotations extending over a line boundary is accepted, but strings-denotations overflowing a line without a string break are left strictly alone. If a brackets mismatch is found, it makes a crude attempt to repair it.

```
PROGRAM INDENT(INP, OUT, OUTPUT);
CONST
  SMALLINDENT=2; MIDINDENT=2; LARGEINDENT=4;
TYPE
  STATETYPE =
```



```
      (OPENER, MIDDLER, CLOSER, PRAGMENT, DOER, QUOTE, COLON, GO, OTHER);
  CLAUSETYPE =
      (BRIEF, CONDCL, CASECL, CLOSEDCL, LOOPCL, INDEXER, ROUTINE,
      JUMP, EXIT, SEMICOMMA, STRING,
      HASH, CO, COMMENT, PR, PRAGMAT, ANY);
  TREEP=↑TREE;
  TREE=RECORD
  (*TREE TO HOLD RESERVED WORD DICTIONARY*)
    C: CHAR;
    LEFT, RIGHT, NEXT: TREEP;
    TIP: BOOLEAN;
    ST: STATETYPE; CL: CLAUSETYPE;
    END;
  STACKP=↑STACK;
  STACK=PACKED RECORD
    C: CLAUSETYPE; G: BOOLEAN;
    NEXT: STACKP
    END;
(*ALFA=PACKED ARRAY [1..10] OF CHAR;*)
VAR
  INP, OUT: TEXT;
  ROOT: TREEP;
  TOS: STACKP;
  VETTEDCHARACTER: RECORD
    WORD: ARRAY [1..80] OF CHAR; (*THE LONGEST CONCEIVABLE BOLDWORD!*)
    INDEX: 0..80;
    END;
  STARTOFLINE,
  LINENUMBERS: BOOLEAN; (*TRUE IFF THE SOURCE TEXT INCLUDES LINE NUMBERS*)
  I: INTEGER;
  INDENT, (*EXPECTED INDENT FOR SUBSEQUENT LINES*)
  TEMPINDENT: INTEGER; (*INDENT FOR CURRENT LINE*)
  INSTRAGMENT: BOOLEAN;
  GONEON: BOOLEAN; (*TRUE IFF THE LAST TOKEN WAS AN OPENER OR A MIDDLER*)
(**)
(**)
PROCEDURE SETUPTREE;
(*TO CREATE THE DICTIONARY*)
  PROCEDURE INSERT(WORD: ALFA; S: STATETYPE; B: CLAUSETYPE);
    VAR TREEPTR: TREEP; INDEX: INTEGER; FOUND: BOOLEAN;
      BEGIN TREEPTR := ROOT; INDEX := 1;
      WHILE WORD[INDEX]<>' ' DO
        BEGIN
        WITH TREEPTR↑ DO
          BEGIN
          IF TREEPTR↑.NEXT=NIL THEN
            BEGIN NEW(NEXT); WITH NEXT↑ DO
              BEGIN C := WORD[INDEX];
              LEFT := NIL; RIGHT := NIL; TIP := FALSE; NEXT := NIL
              END
            END;
          TREEPTR := NEXT
          END;
        FOUND := FALSE;
        WHILE NOT FOUND DO WITH TREEPTR↑ DO
          IF WORD[INDEX]<C THEN
            BEGIN
            IF LEFT=NIL THEN
              BEGIN NEW(LEFT); WITH LEFT↑ DO
                BEGIN C := WORD[INDEX];
                LEFT := NIL; RIGHT := NIL; TIP := FALSE; NEXT := NIL
                END;
```

```
            FOUND := TRUE
            END;
          TREEPTR := LEFT
          END
        ELSE IF WORD[INDEX]>C THEN
          BEGIN
          IF RIGHT=NIL THEN
            BEGIN NEW(RIGHT); WITH RIGHT↑ DO
              BEGIN C := WORD[INDEX];
              LEFT := NIL; RIGHT := NIL; TIP := FALSE; NEXT := NIL
              END;
            FOUND := TRUE
            END;
          TREEPTR := RIGHT
          END
        ELSE FOUND := TRUE;
      INDEX := INDEX+1
      END;
    WITH TREEPTR↑ DO
      BEGIN TIP := TRUE; ST := S; CL := B END
    END (*INSERT*);
(**)
  BEGIN (*SETUPTREE*)
  NEW(ROOT); ROOT↑.NEXT := NIL;
  INSERT('(          ', OPENER  , BRIEF    );
  INSERT('.IF        ', OPENER  , CONDCL   );
  INSERT('.CASE      ', OPENER  , CASECL   );
  INSERT('.BEGIN     ', OPENER  , CLOSEDCL );
  INSERT('[          ', OPENER  , INDEXER  );
  INSERT('!          ', MIDDLER , BRIEF    );
  INSERT('.THEN      ', MIDDLER , CONDCL   );
  INSERT('.IN        ', MIDDLER , CASECL   );
  INSERT('.ELIF      ', MIDDLER , CONDCL   );
  INSERT('.ELSE      ', MIDDLER , CONDCL   );
  INSERT('.OUSE      ', MIDDLER , CASECL   );
  INSERT('.OUT       ', MIDDLER , CASECL   );
  INSERT('.EXIT      ', MIDDLER , EXIT     );
  INSERT(';          ', MIDDLER , SEMICOMMA);
  INSERT(',          ', MIDDLER , SEMICOMMA);
  INSERT(')          ', CLOSER  , BRIEF    );
  INSERT('.FI        ', CLOSER  , CONDCL   );
  INSERT('.ESAC      ', CLOSER  , CASECL   );
  INSERT('.END       ', CLOSER  , CLOSEDCL );
  INSERT(']          ', CLOSER  , INDEXER  );
  INSERT('#          ', PRAGMENT, HASH     );
  INSERT('.CO        ', PRAGMENT, CO       );
  INSERT('.COMMENT   ', PRAGMENT, COMMENT  );
  INSERT('.PR        ', PRAGMENT, PR       );
  INSERT('.PRAGMAT   ', PRAGMENT, PRAGMAT  );
  INSERT('.FOR       ', DOER    , LOOPCL   );
  INSERT('.FROM      ', DOER    , LOOPCL   );
  INSERT('.BY        ', DOER    , LOOPCL   );
  INSERT('.TO        ', DOER    , LOOPCL   );
  INSERT('.WHILE     ', DOER    , LOOPCL   );
  INSERT('.DO        ', DOER    , LOOPCL   );
  INSERT('.OD        ', CLOSER  , LOOPCL   );
  INSERT('.GO        ', GO      , JUMP     );
  INSERT('"          ', QUOTE   , STRING   );
    (*':' AFTER BOLD , COLON   , ROUTINE  ); *)
  END;
(**)
(**)
```



```
PROCEDURE PUSH(CL: CLAUSETYPE);
  VAR TEMP: STACKP;
    BEGIN TEMP := TOS; NEW(TOS); WITH TOS↑ DO
      BEGIN C := CL; G := GONEON; NEXT := TEMP END
    END;
(**)
(**)
PROCEDURE POP;
  VAR TEMP: STACKP;
    BEGIN
    IF NOT GONEON AND NOT INSTRAGMENT THEN INDENT := INDENT-MIDINDENT;
    TEMP := TOS; GONEON := TOS↑.G; TOS := TOS↑.NEXT; DISPOSE(TEMP)
    END;
(**)
(**)
PROCEDURE VET;
(*MOVES NEXT INTERESTING TOKEN TO VETTED CHARACTER,
  AND SETS INDENT AND TEMPINDENT ACCORDINGLY*)
  VAR TEMP: STACKP;
      TREEPTR: TREEP;
      CH: CHAR;
      STATE: STATETYPE;
      CLAUSE: CLAUSETYPE;
      BOLD, FOUND: BOOLEAN;
      I: INTEGER;
(**)
  PROCEDURE GAP;
  (*ENSURE THAT AT LEAST (SMALLINDENT-1) BLANKS ARE PRESENT IN OUT*)
    VAR I: INTEGER;
      BEGIN
      I := SMALLINDENT-1;
      WHILE NOT EOLN(INP) AND (INP↑=' ') AND (I>0) DO
        BEGIN GET(INP); I := I-1 END;
      IF NOT EOLN(INP) THEN
        FOR I := 2 TO SMALLINDENT DO WITH VETTEDCHARACTER DO
          BEGIN WORD[I] := ' '; INDEX := I END
      END;
(**)
  PROCEDURE CHECK(CLAUSE: CLAUSETYPE);
      BEGIN WITH TOS↑ DO
        IF C<>CLAUSE THEN (*ATTEMPT TO FIX BRACKETS MISMATCH*)
          IF NEXT↑.C=CLAUSE THEN (*ASSUME CLOSER WAS OMITTED*)
            BEGIN
            IF C IN [BRIEF, INDEXER] THEN INDENT := INDENT-SMALLINDENT
            ELSE INDENT := INDENT-LARGEINDENT;
            POP;
            IF GONEON THEN
              BEGIN GONEON := FALSE; INDENT := INDENT+MIDINDENT END
            END
          ELSE (*ASSUME OPENER WAS OMITTED*)
            BEGIN
            IF CLAUSE IN [BRIEF, INDEXER] THEN INDENT := INDENT+SMALLINDENT
            ELSE INDENT := INDENT+LARGEINDENT;
            IF NOT GONEON THEN
              BEGIN GONEON := TRUE; INDENT := INDENT-MIDINDENT END;
            PUSH(CLAUSE)
            END
      END;
(**)
```

```
    BEGIN (*VET*)
    (*ASSERT: INP↑ IN [(!)[],.#";]*)
    CH := INP↑;
    TEMPINDENT := INDENT;
    VETTEDCHARACTER.INDEX := 0;
    BOLD := CH='.';
    TREEPTR := ROOT↑.NEXT; FOUND := FALSE;
    WHILE (TREEPTR<>NIL) AND NOT FOUND DO WITH TREEPTR↑ DO
      IF C=CH THEN WITH VETTEDCHARACTER DO
        BEGIN
        INDEX := INDEX+1; WORD[INDEX] := CH;
        GET(INP); CH := INP↑;
        IF NOT(CH IN ['A'..'Z']) OR NOT BOLD THEN FOUND := TIP;
        IF NOT FOUND THEN TREEPTR := NEXT
        END
      ELSE IF CH<C THEN TREEPTR := LEFT
      ELSE TREEPTR := RIGHT;
    IF FOUND THEN WITH TREEPTR↑ DO
      BEGIN STATE := ST; CLAUSE := CL END
    ELSE
      BEGIN
      WHILE (CH IN ['A'..'Z']) DO WITH VETTEDCHARACTER DO
        (*ABSORB REMAINDER OF UNRECOGNIZED BOLDWORD*)
        BEGIN INDEX:=INDEX+1; WORD[INDEX]:=CH; GET(INP); CH:=INP↑ END;
      IF (CH=':') AND NOT INSTRAGMENT THEN WITH VETTEDCHARACTER DO
        (*START OF ROUTINE-TEXT*)
        BEGIN STATE := COLON; CLAUSE := ROUTINE;
        INDEX := INDEX+1; WORD[INDEX] := CH; GET(INP)
        END
      ELSE BEGIN STATE := OTHER; CLAUSE := ANY END
      END;
(**)
    IF INSTRAGMENT AND (CLAUSE=TOS↑.C) THEN
      (*MATCHING CLOSE-STRAGMENT-TOKEN FOUND*)
      BEGIN
      POP;
      INSTRAGMENT := FALSE;
      IF CLAUSE=HASH THEN INDENT := INDENT-SMALLINDENT
      ELSE IF CLAUSE<>STRING THEN INDENT := INDENT-LARGEINDENT;
      TEMPINDENT := INDENT
      END
    ELSE IF NOT INSTRAGMENT THEN
      BEGIN
      IF STATE IN [MIDDLER, CLOSER] THEN (*MAYBE END OF ROUTINE-TEXT*)
        WHILE TOS↑.C=ROUTINE DO
          BEGIN
          POP; INDENT := INDENT-SMALLINDENT;
          IF GONEON THEN
            BEGIN GONEON := FALSE; INDENT := INDENT+MIDINDENT END
          END;
(**)
      IF STATE=GO THEN (*.GO OF .GO .TO*)
        BEGIN PUSH(JUMP); STATE := OTHER END
      ELSE IF STATE=DOER THEN (*CHANGE IT TO MIDDLER OR OPENER*)
        IF TOS↑.C=JUMP THEN (*.TO OF .GO .TO*)
          BEGIN POP; STATE := OTHER END
        ELSE IF (TOS↑.C=LOOPCL) AND NOT GONEON THEN STATE := MIDDLER
        ELSE STATE := OPENER;
(**)
```

```
IF STATE=COLON THEN (*START OF ROUTINE-TEXT*)
  BEGIN
  IF NOT GONEON THEN
    BEGIN GONEON := TRUE; INDENT := INDENT-MIDINDENT END;
  PUSH(CLAUSE);
  INDENT := INDENT+SMALLINDENT
  END
ELSE IF STATE=OPENER THEN (*START OF A NEW INDENT*)
  BEGIN
  PUSH(CLAUSE);
  IF CLAUSE IN [BRIEF, INDEXER] THEN
    BEGIN INDENT := INDENT+SMALLINDENT; IF STARTOFLINE THEN GAP END
  ELSE INDENT := INDENT+LARGEINDENT;
  GONEON := TRUE
  END
ELSE IF STATE=MIDDLER THEN
  BEGIN
  IF NOT (CLAUSE IN [EXIT, SEMICOMMA]) THEN CHECK(CLAUSE);
  IF NOT GONEON THEN
    BEGIN GONEON := TRUE; INDENT := INDENT-MIDINDENT END;
  IF CLAUSE=SEMICOMMA THEN
    BEGIN TEMPINDENT := INDENT-SMALLINDENT; GAP END
  ELSE IF TOS↑.C=BRIEF THEN
    (* ! OR !: OR .EXIT AFTER ( *)
    BEGIN TEMPINDENT := INDENT-SMALLINDENT;
    IF STARTOFLINE AND (INP↑<>':') THEN GAP
    END
  ELSE TEMPINDENT := INDENT-LARGEINDENT
  END
ELSE IF STATE=CLOSER THEN (*END OF INDENT*)
  BEGIN
  CHECK(CLAUSE); POP;
  IF CLAUSE IN [BRIEF, INDEXER] THEN INDENT := INDENT-SMALLINDENT
  ELSE INDENT := INDENT-LARGEINDENT;
  TEMPINDENT := INDENT;
  IF GONEON THEN
    BEGIN GONEON := FALSE; INDENT := INDENT+MIDINDENT END
  END
ELSE IF STATE=PRAGMENT THEN
  BEGIN


  TEMPINDENT := INDENT;
  PUSH(CLAUSE);
  INSTRAGMENT := TRUE;
  IF CLAUSE=HASH THEN
    BEGIN INDENT := INDENT+SMALLINDENT; IF STARTOFLINE THEN GAP END
  ELSE INDENT := INDENT+LARGEINDENT
  END
ELSE IF STATE=QUOTE THEN
  BEGIN
  IF GONEON THEN
    BEGIN GONEON := FALSE; INDENT := INDENT+MIDINDENT END;
  PUSH(STRING);
  INSTRAGMENT := TRUE
  END
```

```
      ELSE (*STATE=OTHER*)
        IF GONEON THEN
          BEGIN GONEON := FALSE; INDENT := INDENT+MIDINDENT END
      END
    END (*OF VET*);
(**)
(**)
BEGIN (*INDENT*)
INDENT := 0; INSTRAGMENT := FALSE;
GONEON := TRUE;
SETUPTREE;
RESET(INP); REWRITE(OUT);
LINENUMBERS := INP↑ IN ['0'..'9'];
TOS := NIL; PUSH(ANY); PUSH(ANY);
WHILE NOT EOF(INP) DO
  IF EOLN(INP) THEN BEGIN GET(INP); WRITELN(OUT) END
  ELSE
    BEGIN
    STARTOFLINE := TRUE;
    IF LINENUMBERS THEN
      BEGIN
      WHILE INP↑ IN ['0'..'9'] DO
        BEGIN WRITE(OUT, INP↑); GET(INP) END;
      IF INP↑=' ' THEN (*FIRST BLANK AFTER LINE NUMBER IS OBLIGATORY*)
        BEGIN WRITE(OUT, ' '); GET(INP) END
      END;
    IF TOS↑.C=STRING THEN
      (*DO NOT TINKER WITH BLANKS INSIDE STRING-DENOTATIONS*)
      BEGIN
      WHILE INP↑=' ' DO
        BEGIN WRITE(OUT, ' '); GET(INP) END;
        STARTOFLINE := FALSE
        END
    ELSE WHILE INP↑=' ' DO GET(INP); (*GET RID OF EXISTING INDENTATION*)
    WHILE NOT EOLN(INP) DO
      BEGIN
      IF (INP↑ IN ['(','!',')','[',']',',',',','.','#','"']) OR (INP↑=';') THEN
        (*CHARACTER WHICH MIGHT AFFECT INDENTATION*)
                  (*N.B. ';' CANNOT BE A SET ELEMENT IN CDC PASCAL*)
        BEGIN
        VET;
        IF STARTOFLINE THEN FOR I := 1 TO TEMPINDENT DO WRITE(OUT, ' ');
        WITH VETTEDCHARACTER DO
          FOR I := 1 TO INDEX DO WRITE(OUT, WORD[I])
        END
      ELSE
        BEGIN
        IF STARTOFLINE THEN FOR I := 1 TO INDENT DO WRITE(OUT, ' ');
        IF (INP↑<>' ') AND NOT INSTRAGMENT AND GONEON THEN
          (*PREPARE TO INDENT ANY CONTINUATION LINE*)
          BEGIN GONEON := FALSE; INDENT := INDENT+MIDINDENT END;
        WRITE(OUT, INP↑); GET(INP);
        END;
      STARTOFLINE := FALSE
      END;
    GET(INP); WRITELN(OUT)
    END;
END.
```