ISSN 0084-6198

Algol Bulletin no. 50

DECEMBER 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).

The following statement appears here at the request of the Council of IFIP:

> "The opinions and statements expressed by the contributors to this Bulletin do not necessarily reflect those of IFIP and IFIP undertakes no responsibility for any action that might arise from such statements. Except in the case of IFIP documents, which are clearly so designated, IFIP does not retain copyright authority on material published here. Permission to reproduce any contribution should be sought directly from the authors concerned. No reproduction may be made in part or in full of documents or working papers of the Working Group itself without permission in writing from IFIP."

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 $4.50 (or £2.40) each. However, it is regretted that only AB32, AB34, AB35, AB36, AB38-43 and AB45 onwards are currently available. The Editor would be willing to arrange for a Xerox copy of any individual paper to be made for anyone who undertook to pay for the cost of Xeroxing.

AB50.0 EDITOR'S NOTES.

ALGOL 68 Standardization

I mentioned in the last issue that the proposals to produce an International Standard for ALGOL 68 were about to go to a letter ballot within ISO. This produced an overall majority in favour, and even five coutries willing to participate in preparing the Standard (W. Germany, Belgium, Netherlands, U.S.S.R and Czechoslovakia), but unfortunately two of the would-be participators (U.S.S.R. and Czechoslovakia) were not the right kinds of member of the right ISO committees, and so the project did not get through. Thus we are now struggling in the mire of ISO politics and, unless some additional coutries can be persuaded to support it, or unless the rules within ISO can be changed or circumvented, the proposal will likely fail.

ALGOL 60 Standardization

I said in the last issue that ISO 1538 was about to be published. This still seems to be the situation (ISO politics again).

Activities of IFIP TC2

TC2 is the parent committee of Working Group 2.1 (indeed, the ALGOL Bulletin is, strictly speaking, a TC2 publication). The article in this issue (AB50.3.1) is mainly a public relations exercise on the part of IFIP, but it does at least serve as a useful reference as to what each Working Group is supposed to be about. There is a general permission to reproduce the article in whole or in part, provided the original author is acknowledged, and that it is made clear if any truncation or editing has taken place.

Survey of viable implementations

In AB47.3.3 I published a list of viable implementations of ALGOL 68. I intend to publish an updated version in the next issue, and I therefore solicit details of their offerrings from any implementors of the language who were not included previously. The only conditions for inclusion are that the implementation is available for distribution, and that it is already in use on at least two sites.

In the meantime, you might like to know that ALGOL 68C is now available for the DEC VAX (under Berkeley Unix 4.2) from the ALGOL 68C Distribution Service, Computer Laboratory, Corn Exchange Street, Cambridge CB2 3QG, UK. Also that ALGOL 68RS on the VAX (from S.P.L.) is imminent.

AB50.1 Announcements.

AB50.1.1 The *B* Newsletter

Those interested in the *B* programming language (see AB48.4.1 for an introductory tutorial) may be interested in the *B* Newsletter which will be distributed regularly by the Informatics Department, Mathematical Centre, POB 4079, 1009 AB Amsterdam, The Netherlands. The first issue, dated August 1983, listed the various Technical Reports on the language that are available, and gave information about the pilot implementation (UNIX VAX or PDP11) and the soon-to-be-released portable implementation (written in C).

AB50.1.2 IFIP - its aims & its recent publications

Noth-Holland has recently published a booklet entitled "IFIP, its aims & its recent publications". The brochure presents a detailed description of IFIP (International Federation for Information Processing), as well as full details on 55 books reflecting the interest-sphere of IFIP: Programming, Education, Computer Allpications in Technology, Data Communications, System Modelling and Optimization, Information Systems, Computers and Society, Digital Systems Design.

IFIP publications are available to members of national information processing societies at a 25% discount.

Write for your copy of the brochure to: North-Holland Publishing Company, Attn: Joop Dirkmaat, P.O. Box 1991, 1000 BZ Amsterdam, The Netherlands.

AB50.1.3 Programming Languages and System Design

This book, edited by J. Bormann and published by North-Holland, comprises the Proceedings of the IFIP TC2 Working Conference on Programming Languages and System Design, held in Dresden (GDR) on 7-10 March 1983, under the joint sponsorship of IFIP WG2.1 and WG2.4. It contains the texts of the following papers:

Fast Automatic Liveness Analysis of Hierarchical Parallel Systems, by J. Roehrich.
Concatenable Type Declarations - Their Applications and Implementation, by A. Kreczmar and A. Salwicki.
On the Coherence of Programming Languages and Programming Methodology, by M. Broy and P. Pepper.
On the Design of Data Abstraction Mechanisms for Compiler Description Languages, by H. Ganzinger.
The Remodularization of a Compiler by Abstract Data Types, by K. Bothe.
Experience with Abstract Data Type Specifications in a Compiler Project, by U.L. Hupbach and H. Kaphengst.
XYZ: A Program Development Environment Based on Temporal Logic, by C.S. Tang.
Programming in SETL Environment, by D.Y. Levin.

Design and Verification Oriented Microprogram Transformations, by D. Dembinski.
ELSA - An Extensible Programming System, by C.H. Lindsey.
A Skeleton Interpreter for Specialized Languages, by J. Steensgaard-Madsen.
Comparing PASCAL and MODULA-2 as Systems Programming Languages, by P.H. Hartel.
Early Experioence with the Programming Language ADA, by G. Persch, M. Dausmann and G. Goos.
There are also the transcripts of two discussion sessions.

Price information is not curenmtly available, but presumably the 25% discount to members of national information processing societies (see previous section) will apply.

AB50.1.4 Book Review : Guide to ALGOL 68 - for users of RS Systems

by Philip M. Woodward and Susan G. Bond
160pp. Publ. Edward Arnold, £5.95.
ISBN 0 7131 3490 9.

Those who have used the previous "yellow" and "green" books (H.M.S.O) by Woodward and Bond (which were companions to the original Malvern ALGOL 68R) will imediately recognise the practical style and headlong pace of this new book. Every important fact about ALGOL 68 is there, but is mentioned only once, so do not blink as you read the book.

Thus the book is not for raw beginners - nor even for the hobbyist who thinks computing is just the BASIC provided on one of Mr Sinclair's toys. But for the user of FORTRAN, or PASCAL (or even ADA), whose disillusionment with those languages is not yet quite complete, it is ideal. Although intended primarily for users of the Malvern-developed RS compilers (as implemented on the ICL 2900, the Honeywell Multics and, soon to come, on the DEC VAX), the discrepancies between ALGOL 68RS and the language of the Revised Report are meticuloulsy (and mostly unobtrusively) recorded, so users of other systems need have no fear. Errors of commission are conspicuously absent (I hope the remark about implementors keeping line buffers in their *FILE* structures was not really intended) and errors of omission are few (but I could not find, for example, any mention that I could include the word *LOC* in a variable-declaration, nor any mention of *PRAGMAT*s, nor of mode-equivalence). On the other hand, good programming style is well described (if you excuse Philip Woodward's morbid fear of using the heap), and the section on list processing, with full discussion of the "3-*REF* trick", is particularly thorough.

As a work of reference, the book is less successful (in spite of claims in the introduction to the contrary). The facts are all there and will be found on sequential reading, but it appears that *set* is not *possible*. I looked in vain in the index for the word "scope" (they tend to misuse the word "range" when discussing this matter, but even that word in the index did not lead me to the discussion of scope violations). Thus, as a reader of the ALGOL Bulletin and therefore already presumably having a good knowledge of the language, there would not be much gain in buying the book for yourself. But it would make an ideal Christmas present for your friends.

C.H. Lindsey.

AB50.1.5 Preliminary Conference Announcement.

IFIP Working Conference on Problem Solving Environments for Scientific Computing

IFIP TC2 will be holding a working conference on "Problem Solving Environments for Scientific Computing" at the INRIA-SOPHIA-ANTIPOLIS Laboratory in France on the 17th - 21st June 1985. A Problem-Solving Environment (PSE) is an integrated multi-tasking system that supports the solution of a given problem. In many scientific areas, computer software has been developed with specialised high-level languages, complex data structures, graphical displays and post-processors. Such packages allow the user to employ the terminology of the problem area, remove the need to become involved in low-level programming details and maximise productivity.

Work on PSEs has led to the development of facilities directed to specific problem areas. For example, expert systems involve automated reasoning, data base manipulation and question and answer sessions. Statistical PSEs have emphasised problem oriented languages. In the CAD/CAM environment data display and the use of display equipment are crucial. The aim of the conference is to bring together workers on scientific packages and on PSEs to exchange ideas and experiences.

We shall examine PSEs in scientific applications with particular emphasis on the role of numerical computing. Rather than review a number of existing systems in detail, we hope to consider the overall specification, construction and development of a PSE. To this end some topics of interest are:

comparison of self-contained and open systems; applicability of knowledge-based techniques to numerical problems; integration and design of a user interface; achievement of clean dataflow handling; data display; selection and design of implementation language; design, transformation and maintenance of integrated data structures; support, parsing and processing of user dialogue; use of reliable numerical algorithms and diverse hardware; interpretation and summary of results; and the impact of personal computers and scientific workstations.

The meeting will include both invited and contributed papers. One session will be reserved for discussion of issues that arise during the conference. Time will also be included for demonstration of PSEs by attendees, who will be responsible for provision of their own computing resources (although some local assistance with electrical power and modem contact will be available). Substantial time in the programme will be allocated to discussion. The proceedings, including an edited transcript of the discussion, will be published.

In order to preserve a "workshop" atmosphere the number of participants will be limited to approximately 80.

Those interested in attending should write to the Conference Chairman, B. Ford, NAG Central Office, 256 Banbury Road, Oxford OX2 7DE, England preferably including a brief description of their work and interests in the area, to support their application. Please also indicate whether you will be able to finance your own expenses; we are applying for financial support, but this is unlikely to extend beyond partial support for invited speakers.

AB50.3.1

THE ART AND SCIENCE OF PROGRAMMING

IFIP's community of experts tackle key problem areas

(This report has been written for IFIP by Kenneth Owen, former Technology Editor of The Times, London)

Programming arouses strong feelings — of incomprehensibility, alas, to many people outside computing; of partisan vehemence by factions of experts on the inside. In the world of computers, no subject is more basic, none so all-pervasive in its impact (for good or ill) on the performance of computing systems.

For a subject as wide-ranging and ubiquitous as this, who would dare to attempt to set up a framework within which experts would comprehensively monitor and advance the state of this peculiar art? The answer to that question is IFIP, the International Federation for Information Processing, and in particular the Federation's Technical Committee 2 (TC 2).

IFIP's members are national professional and technical institutions. Its aims are to promote information science and technology and to stimulate research, application and international co-operation in this field. TC 2 is one of nine technical committees within IFIP, each acting as a forum for discussion in a particular technical area. Chairman of TC 2 is Professor Manfred Paul of the Institut für Informatik at the Technical University of Munich.

Each IFIP technical committee, made up of national representatives, devolves its technical work to a number of working groups, which invite appropriate experts to participate regardless of nationality. Each working group covers a particular specialization.

The work of TC 2 has evolved in response to developments in the science of programming over the years, Professor Paul points out. In the late 1950s, he recalls, programming really was the struggle to get away from assembler languages and to create higher-level languages such as FORTRAN and ALGOL.

In particular, the preliminary report on ALGOL which appeared in 1958 acted as a focus of intense interest for computer scientists. IFIP itself was created under the auspices of UNESCO in 1960, with two initial technical committees, TC 1 (Terminology) and TC 2 (Programming). For TC 2, the title « programming » at that time meant high-level programming language design, followed closely by the start of work on the automatic translation of such languages. But, even then, it was clear that there was more to programming than simply programming languages.

Working Group 2.1 (ALGOL) was formed in 1962. It began its work by first discussing how to implement a language such as ALGOL. As a result, 2.1 was trying to revise and improve the 1960 ALGOL Report, while, at the same time, it was investigating the problems of translating such a language.

« During that time it became clear that one had to study not only the programming language itself, but also the means by which such languages were defined », Professor Paul says. « And, for that definition also, a formal language is best. » This led to TC 2's first working conference, held in Baden, Austria in 1964, at which languages for defining programming concepts were discussed.

The following year a second working group, WG 2.2 (Formal description of programming concepts) was set up. « Not only did that widen the view of problems in computing », Professor Paul notes, « but it established a formal way for those topics to be discussed. »

A decisive turning point came in 1966 when WG 2.1 (having revised the ALGOL 60 Report) started to think about further concepts which would enable a high-level language to deal more easily with more general, non-numerical algorithms, such as those for text handling, for instance. More general data structures than those of ALGOL 60 were sought, and to this end a new way of defining programming languages was presented.

The working group commissioned a sub-group to define a successor to ALGOL 60 which would incorporate the new data structures and use a more rigid means of defining the language. The result was ALGOL 68, an undoubted intellectual achievement

but one which suffered from the extreme rigidity with which the language was defined, and from a number of programming concepts which were not generally accepted.

In 1969 the ALGOL 68 controversy led to the hiving-off from WG 2.1 of a new working group, 2.3, concerned with programming methodologies. This reflected the expansion at that time of the overall concept of programming. It had become clear that programming was more than just looking for the concepts in the language and for definition methods. It was also about the methodologies and tools needed to transform ideas for algorithms into working programs.

Up to this time the main interest in programming had concerned programs for user applications. But in the early 1970s a new interest emerged in the design of operating systems and systems software generally. A working conference on machine-oriented high-level languages was followed by the formation in 1973 of Working Group 2.4, concerned with system implementation languages. Thus the understanding of programming was growing progressively as different facets of the subject came under scrutiny.

The next development within TC 2 was concerned not with a new aspect of programming but with a very familiar one — that of numerical software. In one sense the numerical analysts were really the people who started it all, anyway : they were the first to use big computers and to write complex algorithms. In 1974 their interests were formally recognised with the setting up of WG 2.5 (numerical software).

Data bases had become very much a hot topic by this time, and the study of data base languages and technologies was taken up by WG 2.6, the second working group to be formed in 1974. Command languages which give an interface to the system had also emerged as a potentially difficult area, needing further study, and this was reflected in the formation in 1975 of Working Group 2.7 (operating system interfaces).

TODAY'S TOPICS The pressing issues in the field of programming today, Professor Paul says, are centred on concurrent programming, distributed systems, and expert systems in the widest sense. But, within these areas, the questions are the same as before : how to go about solving problems with the help of the systems that you have. The systems are now more sophisticated, they may be distributed via local area networks, they are likely to contain huge data bases.

« You have to know how to deal with huge amounts of data, for example, 10^6 or 10^7 different objects that have to be grouped according to certain criteria. That's what data bases do and, if you have the right control and method to use the system purposefully and through an attractive man – machine – communication, it may develop into an expert system. »

And, the TC 2 chairman adds, there is the collection of problems which are being addressed in the Japanese fifth-generation computer programme. Professor Paul does not believe that all the aims and goals of the Japanese programme will be reached within the planned decade. He doubts whether some will ever be reached.

« But I'm sure that some of the interesting issues of artificial intelligence — for example, artificial vision, pattern recognition in the area of audible transmissions, or robotics — all come together to form a very complex bunch of questions and problems which have to do with programming. »

Programming is an engineering discipline, Professor Paul comments. While the classical engineering fields deal with matter, the software engineer's « matter » consists of information.

Similarly, with programming, a new dimension is added to engineering. « Programming is partly a fundamental science, and partly it is an engineering discipline. » And the work of TC 2, the chairman insists, covers both theory and practice.

The committee has rejected suggestions that it should adopt the phrase « software engineering » in its formal terms of reference or in those of one or other of its working groups. This is not because the subject is not important — it is regarded as very important indeed — but because the present definition of the committee's scope (as quoted later in this article) is seen as already embracing the subject.

« So far », says Professor Paul, « I think all the actual problems that have come up in programming have been incorporated into the work of our groups, although sometimes the shift of interest to tackle new problems may take perhaps two years to achieve.

« The framework has proved remarkably good, and also flexible in that we have encouraged young people with new ideas to participate. We believe our structure covers the field at present. »

GROUP ACTIVITIES Against this background of the historical development of programming and today's topics of interest, TC 2's working groups are pursuing active programmes in their respective areas – and in their various ways. To hold working conferences of experts and publish the proceedings is one well-established practice within IFIP, but no standard working style is laid down for the working groups. In the TC 2 groups in particular, the style reflects the membership.

Starting point for the TC 2 work is the formal definition of the committee's scope. This says comprehensively that the committee's work includes :

- general considerations concerning programming principles and techniques, such as concept development, classification and description;
- the investigation and specification of particular programming languages;
- the investigation and specification of programming systems; and
- the identification, investigation and specification of programming techniques and their applications.

On occasions the committee as distinct from a particular working group will sponsor working conferences; a recent example (in Keskemet, Hungary, in May 1983) covered system description methodologies.

The committee's longest-established working group, WG 2.1, takes as its general scope at present the continuing support of ALGOL 60 and ALGOL 68; and « the exploration and evaluation of new ideas in the field of programming, possibly leading to further languages. » Within this latter area the group has now embarked on a particular line of study which has significant implications.

Over the past two years the group has explored some of the concepts involved in programming by transformation – that is, the search for methods to transform a formal specification into a runnable program. They did not believe this could be done completely automatically, but interactive methods might help in a step-by-step sequence.

Now the group has narrowed down from the search for general concepts to a scrutiny of particular languages in the context of transformational programming. One example is the language CIP-L (CIP stands for Computer aided, Intuition guided Programming) developed by scientists at Munich Technical University.

Working Group 2.2 (formal description of programming concepts) describes its scope as « to explicate programming concepts through the development, examination and comparison of various formal models of these concepts. » Last year it held its second working conference on the formal description of programming concepts in Garmisch, F.R.G. Just as the first working conference on the same topic in St. Andrews, Canada, it was very well received.

Working Group 2.3, set up in 1969 by a minority of 2.1 members who had opposed the publication of ALGOL 68 (« feeling that programmers needed tools other than bigger and better programming languages », to quote Mr. M. Woodger of the UK, chairman of 2.3), has a record of distinguished contributions to computer science by its members – and of a light, informal working style.

Its subject is programming methodology, and its defined aim could hardly be more all-encompassing – « The work of the group is directed towards increasing programmers' ability to compose programs. » Eight topics are listed to illustrate the scope of the group's work, but again the net is cast deliberately wide.

The group sets out to provide an international forum for the discussion of programming methodology. Informal discussion meetings rather than formal conferences are the rule, with the result of this interaction appearing in the normal scientific literature rather than in special published proceedings.

Machine-oriented higher-level languages, otherwise known as system implementation languages, are the concern of WG 2.4. In general these are characterized by :

- their intended application area (software development);
- their machine orientation (they may be used as assembler replacements);
- their concern with the efficiency of the object program; and
- their use of control features (but not necessarily data or operation features) of general purpose programming languages.

Members of the group have been much concerned with the Ada language in recent years (about half the members were involved in Ada program development). Now the emphasis is changing towards programming environments, concurrent systems, machine architecture and compiler technology, with the goal of deriving requirements for future system programming languages.

As a group, the members aim to explore the techniques involved in their kind of languages, rather than to design a specific language of their own. In March 1983 at Dresden, G.D.R., a joint 2.1/2.4 open conference was held on programming languages and systems design. WG 2.4 members are now preparing for a 1984 working conference on « system programming languages – experiences and assessment », which will be held in Canterbury, England.

Working Group 2.5 aims to improve the quality of numerical computation by promoting the development and availability of sound numerical software. Most of its activities take the form of projects, in which one or more members pursue a chosen subject in collaboration with other scientists in the field.

Subject areas which have received the attention of 2.5 include the transportability of numerical software, languages for numerical software, programming environment for the development of numerical software, hardware requirements for numerical software, evaluation of numerical software, and numerical software for special areas.

Software for solving partial differential equations was the subject of a working conference in Sweden in August 1983. The group is working towards a closer collaboration between the designers of numerical software and of statistical software.

Although the general-purpose scientific languages form the basis of general-purpose scientific computation, there is a need for more specialized languages and computing environments (e.g. for computer aided design). These can free the user from the necessity to learn a sophisticated programming language, and can address the problem area directly using its own vocabulary. A forthcoming working conference will explore the implications.

Working Group 2.6 (data bases), whose scope is « to investigate, evaluate and develop data base languages and technologies », has been relatively inactive in recent years. Under a new chairman it is now planning to launch a new programme, starting with a working conference on conceptual schema design methodology.

Working Group 2.7 aims to investigate the nature and concepts of the interfaces of operating systems. Within this broad scope the group is now working on a project to produce a framework in which user interfaces to an operating system can be described and modelled. Such a framework must be capable of modelling both existing and future command and response languages, used both locally and in networks.

The thinking behind this is that many people have tried to define command and response in terms of the actual languages, and without considering the underlying concepts. As a result, the user interfaces to computer systems are difficult for non-experts to understand. The 2.7 project should provide a simpler and better organized framework within which the user interface can be tailored to different types of users.

Hence WG 2.7's draft « reference model for command and response languages », which is now being further refined by the group. The model itself does not define the syntax of the language, but describes the underlying system. The user can then define the commands in one way or another – lines typed at a keyboard, for example, or spoken commands, or sensor devices, or network interfaces.

FUTURE DIRECTIONS One area which is of increasing interest to TC 2, as mentioned by the chairman, is that of distributed data processing. Mr. T.B. Steel Jr. of the USA, a former TC 2 chairman, now acts as liaison officer between the committee and the Open Systems Interconnection (OSI) activity of the International Standards Organization (ISO), which is concerned with the underlying network aspects of distributed systems.

The link with the ISO has a double benefit : it keeps TC 2 members aware of the progress of the OSI work; and it enables them to influence some of this work if necessary. It also provides a base from which to address the wider aspects of distributed systems.

On top of the network itself, there are problems of integrating with OSI such things as genuinely distributed data bases, programming languages, graphics facilities and command and response languages – in very general terms, the distribution and in-depth user interface aspects of such systems.

To achieve fully integrated distributed systems, Mr. Steel suggests, will take at least ten years' work. For TC 2 to address this subject in a more formal way – possibly by holding a working conference which might lead to the formation of a new working group – would be a logical future development.

Another development could be to bring in to TC 2's activities more experts from the artificial intelligence community, since what they are doing is very much « programming » and their field is advancing rapidly. The subject is clearly of interest to a number of TC 2's existing working groups (such as 2.1, 2.3 and 2.6), but a sharper focus to the committee's interest in this subject would be another logical possibility for the future.

There is no doubt that, over many years, members of TC 2 and its working groups have significantly influenced the development of computing science, both collectively and as individuals. The committee's membership embraces both traditionalists, active in refining familiar techniques; and radicals, keen to investigate new concepts. Perhaps one of their future directions might even lead to the foolproof and almost fully reliable program.

Published by the IFIP Secretariat, 3 rue du Marché, CH-1204 GENEVA, Switzerland
October 1983

For further information, please contact your National Computer Society or the IFIP Secretariat.

AB50.4.1

An Executable Prolog Semantics.

Lloyd Allison
Department of Computer Science
University of Western Australia
Nedlands 6009
25/5/83

Abstract.

A Denotational Semantics of the logic programming language Prolog is expressed in Algol-68. The result is a formal definition that is also executable. It is presented as an example of high-order programming in Algol-68; the eventual aim is to use this to compare differing brands and implementations of Prolog formally and experimentally.

Introduction.

Pagan [1] suggested the use of Algol-68 as a metalanguage to write denotational definitions in, but he recognised that to translate the highly curried functions in a 1-1 manner would require partial parameterisation[2]. For example, the domain of functions A->B->C or proc(A)proc(B)C cannot be used in Algol-68 if the proc result depends on local objects as it usually does. In [3] however the definitions were uncurried, to AxB->C or proc(A,B)C, and then expressed in Pascal, as it happens, to define a very small language with jumps.

Here the technique is applied to a definition of Prolog[4]. The notions of Standard Denotational Semantics are used. The advantage of using a uniform flavour or style of semantics is the ability "to discuss very different languages (for example Pascal and Lisp) within a single framework"[5]. The eventual aim here is firstly to bring Prolog within this framework. Then, Prolog is recognised as only a first approximation to the goal of programming in logic. Its declarative semantics are to be understood as first-order logic but Prolog implementations invariably include non-logical features for various reasons, notably for efficiency but

also to make some programs work at all. These features can only be understood in terms of Prolog's procedural semantics which specify how a program is executed; their use is also called specifying the "control" component of a program. One might devise elegant methods for the programmer to specify the control information[6, 7] or better for this to be generated automatically[7]. The second aim is to define these procedural semantics in the denotational style so as to compare various control mechanisms. It is hoped that the formal theory of Denotational Semantics will illuminate the essential features of the mechanisms. By coding the semantics in Algol-68 it is possible to run the definitions as interpreters and to experiment with them.

Using Algol-68 as a metalanguage for Denotational Semantics exploits the fact that Algol-68 minus assignment is a useful functional language. The technique is perhaps less "clean" than using a true Semantic Compiler-Compiler[8,9,10] but it needs no software other than a compiler.

The semantics given here is for a very basic Prolog, strictly left to right depth-first search and cut is not defined. Jones and Mycroft[11] give a denotational definition at about this level of detail which does include cut. Lassez and Maher[12] give a denotational definition much closer to the declarative interpretation of Prolog.

Prolog.

A very simple example is given here to illustrate some of Prolog. A clause is a statement of fact such as "male(fred)." or

"brother(X,Y):-male(X),parents(X,A,B),parents(Y,A,B).".

This demonstrates atoms such as "fred", variables such as "X" and compound terms such as "brother(X,Y)". The first clause can be thought of as a basic fact in that it is true without further proof. The second is a rule and can be read, X is a brother of Y if(:-) X is male and the parents of X are A and B and the parents of Y are A and B. Note, the implicit association of these clauses with real family relationships is an interpretation supplied by the programmer and not by Prolog.

A question has the form "?male(fred)." which, given the above clauses would result in a "yes" response. "?male(bill)." would give "no" as things stand. A question may also include variables as in "?male(X)." which would result in "X=fred yes" or some similar indication that the question can be solved by binding X to fred. Precise details vary between implementations but there is some way of running through all possible solutions to a question.

Informally, Prolog is implemented by some form of backtracking, usually left to right depth-first search on the terms in the current goal clause. At the heart of the search is a pattern-matching algorithm called unification which attempts to match the current term with the head or left hand side of a clause from the set. This may involve binding variables in the term and/or the head, and is done in "the most general" way possible.

Domains.

The domain of function A->B will be coded as proc(A)B. The high-order domain A->B->C or A->(B->C) must be uncurried to AxB->C and coded proc(A,B)C. The disjoint sum A+B is coded union(A,B) and the product AxB becomes struct(A,B).

An atom can be an integer or a string as in "fred". Lists of values can appear in compound terms. A Prolog program will be represented by a tree-like data-structure value:
mode node=union(int,string,compound,list,location);
mode compound=struct(string op,list args); mode list=struct(value head,tail);
mode value=ref node. For the case of a clause there will be a compound with op ":-" and for a question "?".

Prolog has variables but these should be understood as particular, as yet unspecified, values. Once a variable is bound its value will not change, except that it may contain other as yet unbound variables. The environment in a Prolog program maps variable identifiers onto values
mode env=proc(var id)value. A variable identifier may be reused in several procedures and a procedure may be recursive so to avoid name clashes the approach taken here is to map unbound variables to locations which will hopefully later attain a value in the store, mode store=proc(location)value. Some implementations use systematic renaming of variables which is equivalent to the use of locations.

Top-level semantics.

A program is processed one clause or question at a time sequentially. Each clause can be thought of as (part of) a declaration of a procedure, such as procedure "brother" above. This sequential processing is specified by

```
exec: prog->pnv->dcont->answer
      pnv=term->prok
      dcont=pnv->answer
      answer=({yes}+value)^*

if prog1 and prog2 are each a clause or question then

exec "prog1.prog2." p dc
   = exec "prog1." p newdc
     where newdc=(p')answer:
                 exec "prog2." p' dc
exec  "q:-r."  p  dc =  dd  "q:-r."  p  dc
exec  "?q."    p  dc =  << pp "?q." p emptyenv yes emptystore,
                           dc p
                        >>
```

A declaration continuation dcont is something to execute after the given clause or question. To execute prog1 prog2 in succession, execute prog1 in the given procedure environment p and with a new continuation which will execute prog2. prog1 may update p to p'; execute prog2 with p' and

eventual continuation dc. Clauses are defined by dd and questions by pp.

In Algol-68 exec must be uncurried:

```
proc exec=(value prog,pnv p,dcont dc)answer:
   case prog in
      (list l):exec(head of l, p,
                          (pnv p2)answer:
                           exec(tail of l, p2, dc)
                         ),
      (compound c):
         if op of c = ":-" then
              dd(args of c, p, dc)
         else # must be question#
              cons( pp(args of c,
                       p,empty env,yes,empty store),
                    dc(p)
                  )
         fi
      out undefined
      esac

where
   mode dcont=proc(pnv)answer;
   proc undefined=answer:
          (print("undefined prog"); goto stop; skip);
   env empty env=(string id)value: unbound;
   store empty store=(location l)value: unset
and
   informally, yes gives answer "yes"
   pp processes questions and dd declarations or clauses.
```

Clauses.

Clauses are defined by dd,

```
dd:clause->pnv->dcont->answer
dd "q:-r." p dc = dc newp
   where newp=(term t)prok:
                ( p t  ||
                  unify t q emptyenv callr
                )
      where callr=pp r newp
```

To evaluate "q:-r." given procedure environment p and declaration continuation dc, apply dc to an updated p which will also attempt to unify terms t with q and, should that be successful, ask "?r". Note emptyenv occurs because variable identifiers are local to a clause, and || defines the order of search amongst the rules, usually sequential. It is here that the branching search is defined; both the old p and the new unification may

produce answers.

A term such as brother(,) denotes a prok=cont->store->answer. cont is defined below. pnv=term->prok =term->cont->store->answer or uncurried mode pnv=proc(term,cont,store)answer.

Note that a fact "q." is taken to be a rule "q:-." with an empty right hand side or nothing to prove.

In Algol-68 we have,

```
proc dd=(value dec,pnv p,dcont dc)answer:
  ( pnv newp=(value t,cont success,store s)answer:
      cons( p(t,success,s),
            ( pcont callr=(env e2,store s2)answer:
                        pp(tail of dec, newp, e2,
                           (env e3,store s3)answer:
                                success(s3);
                           s
                          );
              unify(t,head of dec, emptyenv, callr, s)
      ) );
   dc( newp )
  )
```

Questions.

We need to define the backtracking search process of a typical Prolog implementation. A particular term "?male(hilary)." may succeed or fail depending on whether it is a fact or not. If this is a subgoal of a bigger goal, success means continue with the attempted proof, failure means give up (backtrack). In fact a goal "?male(X)." might succeed in several ways so in general we have a branching process.

A (procedure) continuation is something further to try in case of success pcont=env->store->answer or mode pcont=proc(env,store)answer, it uses the env and store, which are growing during the forward search, to produce an answer.

When a subgoal has been proved, we continue with the main proof but variables are local to the subproof so such a continuation is defined as cont=store->answer or mode cont=proc(store)answer. The environment created in the subproof is discarded on return, but any necessary values are passed on in the store. A clause denotes a prok=cont->store->answer. The cont is to be evaluated in success. A question might be a single term "?q." or a list "?q,r,s." - are q and r and s true?

```
pp:quest->pnv->env->pcont->store->answer
pp "?." p e pc s = pc e s          # the nil rhs#
pp "?q,r." p e pc s
   = pp "?q." p e dorest s
     where dorest=(env e')store->answer:
                  pp "?r." p e' pc
```

```
pp "?q." p e pc s
   = p  map(q,newe,news)  pc(newe)  news
```

To evaluate an empty goal just evaluate the given continuation. To prove a list of terms, prove the head of the list with a continuation which seeks to prove the rest of the list; this defines the left to right search within a clause. If there is a single term, it may contain variables. The updated environment and store, newe and news, contain bindings of any such variables to free locations so as to avoid name clashes; "map(q,newe,news)" is q with the variables so replaced. In any case, test the set of procedures p to see if q is provable. Note that in evaluating "?q,r.", any bindings in the unification of q are passed, in e', to the evaluation of "?r.".

In the Algol-68 we have

```
proc pp=(value quest,pnv p,env e,
                 pcont pc,store s)answer:
 if quest is nil then
    pc(e,s)
 else
   case quest in
      (list l): pp(head of l,
                       p, e,
                       (env e2, store s2)answer:
                          pp(tail of l,p,e2,pc,s2),
                       s
                  ),
       (compound c):( env newe= not v.interesting;
                       store news=     ditto        ;
                       p( map(c,newe,news),
                       (store s2)answer:
                              pc(newe,s2),
                       news
                  ) )
   out undefined
   esac
 fi
```

Note that these semantics do not allow for system procedures such as assert which may update the procedure environment p.

Unification.

Unification is responsible for the matching of terms to the heads of clauses. If no variables are involved this is a simple (tree) equality test; an atom matches itself and structures match if their components match.

By this stage, the term contains no variables - any still not determined are replaced by unique locations. An unset location unifies with a value by being bound to it. A location unifies with a variable by the variable becoming bound to the location, or sharing with it. A variable may appear more than once in a term as in "d(X,X,1).". This may result in

an unset location unifying with an unset location; in this case they are both bound to a new unset location if they differ.

Unification usually proceeds left to right across a term. As it progresses the environment and the store may be updated. At any time matching may fail and backtracking occur. Unification can be defined in the same way as the rest of the execution of Prolog. The definition is long because of the number of cases involving locations and variables in the term and/or the head of the clause and whether they are first occurrences or not. Two of the controlling cases only are given:

```
unify: (term x clause)->env->pcont->store->answer

1) unify  "q(args1)" "r(args2)" e pc s
   =  if q=r then unify "args1" "args2" e pc s

2) unify "a1,a2" "b1,b2" e pc s
   =  unify "a1" "b1" e  newpc  c
      where newpc=(env e')store->answer:
                  unify "a2" "b2" e' pc
```

In Algol-68 this becomes

```
proc unify=(value term,clause,env e,pcont pc,
                          store s)answer:

1) if op of term = op of head of clause then
      unify(args of term,args of head of clause,
            e, pc, s)

2) unify(head of term, headad of clause, e,
         (env e2,store s2)answer:
            unify(tail of term,tail of clause,e2,pc,s2)
        )
```

Output.

Prolog provides many standard system predicates or procedures. One of these is write.

```
?male(X), write(X,is,male).
```

might result in

```
fred is male   yes
```

Such system procedures can be provided in an initial

```
pnv start pnv=(value t,cont success,store s)answer:
   if t is nil then
      success(s)
   elif op of t = "write" then
      cons( map(args of t,nilenv,s), success(s))
   fi
```

The argument of write should contain no variable identifiers but it might contain locations which should be mapped to values in the store.

Conclusion.

Using Algol-68 to code Denotational Semantics gives a formal definition that is mechanically checked and is executable. Such an interpreter is a reference implementation and is very useful for experimentation. The semantics presented here has not been compiled or run as is, but an Algol-68S[13] version has been used to run simple Prolog programs. This Algol-68S does have heap but lack of union, multiples in structs and restrictions on string make this version less elegant. A shortcut of using the standard output file to take the answers produced was also used. Unify is 100 lines long; pp, ee and exec together take 75 lines. The syntax of Prolog was coded into a recursive-descent parser to build the tree for the semantics to walk.

References.

[1]. F.G.Pagan "Algol-68 as a metalanguage for Denotational Semantics."
Computer Journal V22 No1 (Feb 1979) p63-66

[2]. C.H.Lindsey "Partial Parameterisation"
Algol Bulletion No 37 p24-26 (1974)

[3]. L.Allison "Programming Denotational Semantics"
Computer Journal Vol 4 1983 p164.

[4]. W.F.Clocksin, C.S.Mellish Programming in Prolog
Springer Verlag 1981

[5]. M.J.C.Gordon The Denotational Definition of Programming Languages.
Springer verlag 1979

[6]. K.L.Clark, F.McCabe "The control facilities of IC-Prolog."
in Expert Systems in the Micro-Electronic Age
D.Michie (ed) Edinbugh U.P. 1979

[7]. L.Naish "MU-Prolog 3.0 reference manual"
Melbourne University May 1983

[8]. L.Paulson "A Semantics Directed Compiler Generator"
9th Annual Symposium on Principles of Programming Languages Jan 1982 p224-233

[9]. M.R.Raskovsky "Denotational Semantics as a Specification of Code generators"
Proc' 1982 Sigplan Conference on Compiler Construction
June 1982 p230-244

[10]. R.Sethi "Control Flow aspects of Semantics Directed Compiling."
Proc' 1982 Sigplan Conference on Compiler Construction
June 1982 p245-260

[11]. N.D.Jones, A.M.Mycroft "Stepwise Development of Operational and Denotational Semantics for Prolog."
Draft version April 83
Copenhagen University/Edinburgh University

[12]. J-L.Lassez, M.J.Maher "Closure and Fairness in the Semantics of Programming Logic."
University of Melbourne 1983

[13]. C.H.Lindsey "Algol-68S system"
University of Manchester 1982

AB50.4.2 A Library Mechanism for the CDC Algol 68 Compiler

by David A.J. Outteridge
(Martin Marietta Corporation, Denver, Colorado, U.S.A.)

Summary.

Algol 68 has no machine-independent construct to allow the use of library facilities. However in the CDC implementation use is made of pragmats to offer the user the opportunity both to access separately compiled code and to make additions to the standard prelude. This article describes a method - implemented and in use for over three years - whereby these available facilities are used to create a system that enables a user to create, use and maintain extensive modular libraries in a simple way. An Algol-like construct that could be considered machine-independent is used in programme source code to access required modules from the desired library.

Introduction.

A library of operators appears to be a sucession of small pieces of code; and one would expect to draw on only a few operators in a particular programme. CDC describe their Algol 68 compiler prelude-addition facility as library addition; hence one produces a "library-prelude". These facts led the author, at that time engaged in modelling a physical engineering problem and blissfully unaware of what was happening in the computer, to create a large, mostly dormant, library-prelude. So large a prelude that one day a particular programme filled up all available memory on the Cyber in use at the time. The relevant characteristic is that all the additional (user) code is loaded at assembly-time rather than the small percentage that is actually used; all the code is compiled into one module.

After a short period of dicussion, during which the learning rate was positively astronomical, the neophyte accepted with gratitude the offer from the Data Systems Division of the company to provide a more suitable system. This system has grown significantly, gradually becoming more sophisticated in capability so that now it provides a library use and manipulation mechanism that is flexible and yet retains all the checking of the CDC compiler.

Advantages of the system include the ability to write number-crunching routines in optimised Fortran or Compass (assembly code) in a way that is user-transparent and that may be done in a selective manner so that run-time checking may be carried out until there is no further point. As an example: an Algol matrix multiplication routine can do bound checking and matching and then call optimised Fortran Code to do the work. Perhaps it is necessary to bring this approach into perspective by pointing out that it can reduce run time by a factor of four; CDC optimised Fortran is fast.

The library system is believed to be totally secure at compile-time (as opposed to a do-it-yourself access to separately compiled code) as will be explained. However at the time of writing this paper there is no check made at assembly time that the module being loaded is indeed the one

that was in the library at compile time (i.e. the one intended to be used). It is hoped to introduce a "date-stamping" system similar to that used in Algol 68R - but presumably then there will be the frequent necessity to re-compile mentioned in AB 48.4.2; however it is thought that this is preferable to disastrous results, and an up-date compilation of one module is possible which eases the problem somewhat.

Despite the objections that may be made about the library system when viewed from an academic viewpoint - it is messy, involving a pre-processer and a significant amount of operating system control - perhaps its greatest virtue is that it is here and it works. The library system enables practical engineering analysis of significant problems in a way that would not be possible without it.

CDC Syntax.

To create a library-prelude it is necessary both to submit source code to the compiler in special format and to signal that one wishes to compile in prelude addition mode. The latter is effected simply by including an "N" in the compiler parameter list. The special format for the addition source code is as follows:

```
MODULE: ('C' PRELUDE SOURCE CODE  'C';
          'PR' PROG 'PR'
        'C' POSTLUDE SOURCE CODE 'C')
```

MODULE is used as the module name for later identification, when the library is used the PROGramme will be embedded in the pre/postlude as indicated.

The desirability of adding to the postlude is not immediately apparent. An example is: suppose a library were created for a series of programmes that call Fortran sub-routines that involve transput. Since no Fortran main programme exists it would be necessary to open and close the required system files within the Algol main programme. Suppose the procedures that do this opening and closing are "enable fortran transput", "required fortran tapes ('C' list 'C')" and "disable fortran transput". In this case it might be appropriate to make the first procedure call in the prelude, the second in each particular programme with a set of file-name parameters, and the third in the postlude. Not closing the files might well result in lost data.

Separately compiled code is treated as the definition of the unit of a routine. The whole routine then may be ascribed to an operator or procedure identifier as in the examples below; a programme not using separately compiled code is given first for comparison.

A SIMPLE PROGRAMME:
#*****************

```
 LEADING LABELS ARE OPTIONAL, BUT IF PRESENT THE FIRST SEVEN NON-SPACE
 CHARACTERS ARE USED BY THE COMPILER TO NAME THE MODULE; ELSE: A68PROG #

'BEGIN'
       'REAL'  X;   READ (X);
        WRITE (("X IS: ",  FIXED (X, -8, 2),  NEWLINE))
'END' # OF A SIMPLE PROGRAMME #
```

```
                    A SEPARATELY COMPILED ROUTINE:
                    #**************************#
'BEGIN'
       'PROC'  DOUBLE = ('REAL'  Y) 'REAL': 'PR' XDEF TWICE 'PR'

             # TWICE IS THE ENTRY POINT OF MODULE ASEPARA #

              'BEGIN'          # OF UNIT #
                     2 * Y     # YIELD   #
              'END'            # OF UNIT #

              'PR' FEDX 'PR';  # END OF PROCEDURE DEFINITION #

       'SKIP' # AN ALGOL PROGRAMME MAY NOT END WITH A DECLARATION #
'END' # OF THE SEPARATELY COMPILED CODE #


                          CALLING PROGRAMME:
                          #****************#
'BEGIN'
       'REAL'  X;   READ (X);

       'PROC'  DOUBLE = ('REAL'  Y) 'REAL': 'PR' XREF TWICE 'PR''SKIP';

                                            # XREF THIS TIME #

        WRITE (("TWICE X IS: ",  FIXED (DOUBLE (X), -8, 2),  NEWLINE))
'END' # OF THE CALLING PROGRAMME #
```

The separately compiled code may be written in another language, usually Fortran or Compass:

```
      FUNCTION FOURX (X)
      FOURX = 4 * X
      RETURN
      END
```

```
Called by: 'PROC' QUADRUPLE = ('REF' 'REAL'  X) 'REAL':
                              'PR' XREF A68FTN, FOURX 'PR' 'SKIP';
```

Note that the Fortran routine requires the address of X.

Before continuing with the description of how these two constructs are used in the library system structure it is pointed out that with the separate compilation mechanism described, responsibility for correctly matching the interfaces of the calling and called modules lies entirely with the code writer. Not only is no check carried out at load time, but neither is it possible to check at compile time - the modules are compiled separately after all. This is unfortunate for it means that just at the time when it is likely that mis-match errors will occur they are not checked for.

Also note that the library system about to be described does not preclude a user from using the pragmat system shown above; it is supplementary.

Library Structure.

The fundamental ideas behind the library system are:

i) to extend the standard prelude to the minimum degree possible consistant with achieving the required effect, this minimises the amount of space used at run-time. (However the user has full control over the size of prelude; he can reject this objective).
ii) to provide a (large) number of modules which may be accessed via the separate compilation pragmat as required. By calling only those modules required a user may minimise the amount of space used at run-time.
iii) to provide a simple mechanism to automate the accessing process for the user to avoid as many errors as possible.

Using a Library, User Source Code.

A user has an index of available modules in a given library, together with the complete prelude, on the "headers" file; see appendix A. A means of readily accessing this library information is available and normally the user would have a hard-copy for reference. Having decided which modules he needs, a user introduces them where he chooses in his particular programme via the "include" statement:

```
'INCL'  MODULE NAMES SEPARATED BY COMMAS  'LCNI'
```

This statement may be put anywhere it is legal to declare identifiers, and duplicate calls to modules are ignored so that only one declaration of a given module is made (this is important since library modules may call other library modules). From the user's point of view the effect is the declaration - serial, not co-lateral - of the required procedures and operators in place of the include statement. Multiple use of the include statement results in multiple sets of declarations, and duplication in this case is dealt with by the scope rules of the language. Naturally the correct prelude must be used when compiling particular programme source code; this is simply a matter of specifying the module name when calling the compiler, the default being the standard prelude.

Thus, three things are required of a user who wishes to use modules on a library:

i) inclusion of a list of the required modules in particular programme source code.
ii) specification of the required library at compile time via operating system commands.
iii) ensuring the library availability at both compile and load time.

Appendix B contains a sample programme and the resultant listing after pre-processing and submission to the compiler.

Mechanism of the Pre-Processer.

Joint examination of appendices A and B will unveil any mystery about the code conversion the pre-processer effects. Consider for example the declaration of the operator + on lines 4 and 5 of the listing in appendix B. The pre-processer simply has inserted the operator declarative and

formal parameter parts of the module headers entry from appendix A into the source code, and then stuffed the module name into the appropriate pragmat that announces the module as separately compiled. The include statement as such is removed.

The reader should not miss the bonus that this linkage provides: by introducing source code from the library into the particular programme code the pre-processer relieves the particular programme writer of the responsibility of checking the code interface. The compiler is able to do full checking to ensure correct usage; this is considered to be a significant improvement over a method relying on human actions.

Library source code format.

Libraries are required to have special layout, partly to enable the production of the library headers and partly because a few restrictions make the pre-processer a lot simpler. Appendix C contains part of the vector library used in the example; however further detail of the special layout and special rules for library writers will not be spelt out here.

Library Manipulation.

The encouragement of source code generality and subsequent public availability of useful routines is hardly a new idea; however a side-benefit of the library system described is a furthering of this end.

A collection of operating system command procedures has been written with the objective of assisting use of Algol 68 on the CDC Cyber; and, from a practical point of view, it is pointless not to use the library generation routines. The resultant common usage results in common library design; leading in turn both to a common header's format which aids human comprehension, and the capability of manipulating (operating on) libraries with other operating system procedures. Thus, in addition to the vector library, there exist other basic libraries such as a graphics library, a matrix library and a transput library. And these basic libraries may be combined as required using the appropriate operating system routine to provide the library most suitable for the job in hand. The only restrictions are those concerning compatibility of the constituent libraries; for example since the prelude/postlude source codes are assembled into one source code deck and re-compiled it is necessary that there be no duplicate declarations.

Once assembled this tailor-made library is machine-indistinguishable from any other library and the manipulation process may continue. One useful library operation enables the extension of an existing library with further source code. Hence there exists a library used for investigation of satellites that is based on the vector library and one other, but which contains an orbit-specific prelude/postlude and many extra modules.

It is re-iterated that the advantages of this system of library control and manipulation are ease-of-use and complete compiler checking of interfaces at particular programme compilation time.

Other uses.

It appears that by using a number of system-dependent pre-processers a fair amount of code portability could be maintained, although the human management of such a scheme would not be insignificant. It appears that the only requirements of the implementation are some - codable - means of accessing external modules and the ability to extend the prelude.

Acknowledgements.

Development of the CDC Library Mechanism described has been brought about with the facilities provided by the Martin Marietta Corporation, and by assistance in various forms from the following colleagues: Beth Biddle, George Heyliger, Damon Ostrander, Jim Randolph, Dwight Rudolph, Wayne Simon and John Ulrich.

```
VECLIB   :(# VECLIB.  LAST REVISION 19 MAY 1983, ADDED MIN REAL.

            THIS IS A LIBRARY OF OPERATORS AND PROCEDURES FOR USE WITH THE
            MODE 'VEC'. A NORMAL INTERPRETATION OF WHICH IS THAT OF A THREE-
            DIMENSIONAL SPATIAL VECTOR. #

          'PRIO'   ><          =  7,
                   'SCALEDTO'  =  9,
                   'MADEPERPTO'=  6,
                   'PARLTO'    =  5,
                   'PERPTO'    =  5;

          'REAL'   MIN REAL  =  3.0 E -293;                    THE CDC IS NOT SYMMETRIC, THIS IS A TEMPORARY
                                                               FILL-IN.

          'REAL'   SMALL NUMBER CLOSE TO MACHINE LIMIT  =  1E6 * MIN REAL;   USED IN OPERATORS 'SMALL'.

          'MODE'   'VEC'         =  'STRUCT' ('REAL'  XCOORD,  YCOORD,  ZCOORD); MODE DECLARATION.

          'PROC'   ON INDETERMINATE VECTOR := ('VEC' V)        EVENT ROUTINE FOR RUN-TIME CREATION OF AN
                                                               INDETERMINATE VECTOR SUCH AS THE UNIT VECTOR
                   'VOID': ( WRITE ((NEWLINE,  "ATTEMPTED CREATION OF AN",  OF THE ZERO VECTOR; USER ALTERABLE.
                             " INDETERMINATE VECTOR.",  NEWLINE,  "INPUT ",
                              "VECTOR IS PRINTED BELOW AND PROGRAMME",
                             " TERMINATED.", NEWLINE, NEWLINE, V));  STOP);

          'PR'PROG'PR'                                         END OF VECLIB PRELUDE.
          'SKIP')                                              END OF VECLIB POSTLUDE.

FTANPI    'PROC'  PIARCTAN  =  ('REAL'  SIN, COS) 'REAL':      THE ARCTANGENT OF SIN/COS ON (-PI, PI).
XCORD1    'OP'    'X'       =  ('VEC'  U) 'REAL':              X CO-ORDINATE OF U.
XCORDN    'OP'    'X'       =  ({} 'VEC' V) {} 'REAL':         THE X CO-ORDINATE ROW OF A ROW OF VECTORS.
YCORD1    'OP'    'Y'       =  ('VEC'  U) 'REAL':              Y CO-ORDINATE OF U.
YCORDN    'OP'    'Y'       =  ({} 'VEC' V) {} 'REAL':         THE Y CO-ORDINATE ROW OF A ROW OF VECTORS.
ZCORD1    'OP'    'Z'       =  ('VEC'  U) 'REAL':              Z CO-ORDINATE OF U.
ZCORDN    'OP'    'Z'       =  ({} 'VEC' V) {} 'REAL':         THE Z CO-ORDINATE ROW OF A ROW OF VECTORS.
POSVEC    'OP'    +         =  ('VEC'  U) 'VEC':               MONADIC POSITIVE FOR A VECTOR.
SUMVEC    'OP'    +         =  ('VEC'  U, V) 'VEC':            THE SUM OF TWO VECTORS.
NEGVEC    'OP'    -         =  ('VEC'  U) 'VEC':               MONADIC NEGATIVE FOR A VECTOR.
DIFVEC    'OP'    -         =  ('VEC'  U, V) 'VEC':            THE DIFFERENCE OF TWO VECTORS.
ISTVEC    'OP'    *         =  ('INT'  I,  'VEC'  U) 'VEC':    PRODUCT OF INTEGRAL SCALAR AND VECTOR.
RSTVEC    'OP'    *         =  ('REAL' R,  'VEC'  U) 'VEC':    PRODUCT OF REAL SCALAR AND VECTOR.
VECTIS    'OP'    *         =  ('VEC'  U,  'INT'  I) 'VEC':    PRODUCT OF VECTOR AND INTEGRAL SCALAR.
VECTRS    'OP'    *         =  ('VEC'  U,  'REAL' R) 'VEC':    PRODUCT OF VECTOR AND REAL SCALAR.
VECINN    'OP'    *         =  ('VEC'  U,  V) 'REAL':          THE INNER OR DOT PRODUCT.
VECPRD    'OP'    ><        =  ('VEC'  U,  V) 'VEC':           THE VECTOR OR CROSS PRODUCT.
VECQIS    'OP'    /         =  ('VEC'  U,  'INT'  I) 'VEC':    QUOTIENT OF VECTOR AND INTEGRAL SCALAR.
VECORS    'OP'    /         =  ('VEC'  U,  'REAL' R) 'VEC':    QUOTIENT OF VECTOR AND REAL SCALAR.
PABVEC    'OP'    +:=       =  ('REF' 'VEC'  U,  'VEC'  V) 'REF' 'VEC':  PLUS-AND-BECOMES FOR A VECTOR.
PBBVEC    'OP'    'PLUSAB'  =  ('REF' 'VEC'  U,  'VEC'  V) 'REF' 'VEC':  PLUS-AND-BECOMES FOR A VECTOR.
MABVEC    'OP'    -:=       =  ('REF' 'VEC'  U,  'VEC'  V) 'REF' 'VEC':  MINUS-AND-BECOMES FOR A VECTOR.
MBBVEC    'OP'    'MINUSAB' =  ('REF' 'VEC'  U,  'VEC'  V) 'REF' 'VEC':  MINUS-AND-BECOMES FOR A VECTOR.

                                  AND SO ON ....
```

```
'BEGIN'
       'INCL'
              VECPRO,    # THE VECTOR PRODUCT #
              SUMVEC,    # THE SUM OF TWO VECTORS #
             'SKIP'
       'LCNI';

       'VEC'  V1,  V2,  V3,  V4;        READ ((V1,  NEWLINE,  V2,  NEWLINE,  V3));

        V4 := V1 >< (V2 + V3);

        WRITE (("THE ALGORITHM RESULT IS: ",   V4));

'SKIP'
'END' # OF A DEMONSTRATION PROGRAMME #
```

```
                                    * SOURCE LISTING *          A68 1.3.1 82330      83/09/01. 12.07.33.      PAGE     1


      1.            'BEGIN'
      2.
      3.
      4.           'OP'    +         =   ('VEC'  U, V) 'VEC':
      5.               'PR' XREF SUMVEC     'PR' 'SKIP';
      6.           'OP'    ><        =   ('VEC'  U,  V) 'VEC':
      7.               'PR' XREF VECPRO     'PR' 'SKIP';
      8.
      9.
     10.
     11.                   'VEC'  V1,  V2,  V3,  V4;        READ ((V1,  NEWLINE,  V2,  NEWLINE,  V3));
     12.
     13.                    V4 := V1 >< (V2 + V3);
     14.
     15.                    WRITE (("THE ALGORITHM RESULT IS: ",   V4));
     16.
     17.            'SKIP'
     18.            'END' # OF A DEMONSTRATION PROGRAMME #

PROGRAM LENGTH     000223B WORDS
REQUIRED  CM 052600. CP    .410 SEC.
SPECIFIED OPTIONS      ICLBDP
```

```
VECLIB   :(# VECLIB.  LAST REVISION 19 MAY 1983, ADDED MIN REAL.
            THIS IS A LIBRARY OF OPERATORS AND PROCEDURES FOR USE WITH THE
            MODE 'VEC', A NORMAL INTERPRETATION OF WHICH IS THAT OF A THREE-
            DIMENSIONAL SPATIAL VECTOR. #

         'PRIO'   >< = 7,
                  'SCALEDTO' = 9,
                  'MADEPERPTO' = 6,
                  'PARLTO' = 5,
                  'PERPTO' = 5;


         'REAL'   MIN REAL = 3.0 E -293;                                  THE CDC IS NOT SYMMETRIC, THIS IS A TEMPORARY
                                                                          FILL-IN.

         'REAL'   SMALL NUMBER CLOSE TO MACHINE LIMIT = 1E6 * MIN REAL;   USED IN OPERATORS 'SMALL'.


         'MODE'  'VEC'          = 'STRUCT' ('REAL'  XCOORD,  YCOORD,  ZCOORD); MODE DECLARATION.


         'PROC'   ON INDETERMINATE VECTOR := ('VEC' V)                    EVENT ROUTINE FOR RUN-TIME CREATION OF AN
                                                                          INDETERMINATE VECTOR SUCH AS THE UNIT VECTOR
                  'VOID': ( WRITE ((NEWLINE,  "ATTEMPTED CREATION OF AN",  OF THE ZERO VECTOR; USER ALTERABLE.
                           " INDETERMINATE VECTOR.",  NEWLINE,  "INPUT ",
                            "VECTOR IS PRINTED BELOW AND PROGRAMME",
                           " TERMINATED.", NEWLINE, NEWLINE, V));  STOP);


         'PR'PROG'PR'                                                     END OF VECLIB PRELUDE.
         'SKIP')                                                          END OF VECLIB POSTLUDE.


FTANPI   'PROC'  PIARCTAN  =  ('REAL'  SIN, COS) 'REAL':                  THE ARCTANGENT OF SIN/COS ON (-PI, PI).

         ('REAL'        S := SIN,     C := COS;
          'PROC'    ATAN2  = ('REF''REAL'  S2,  C2) 'REAL':
                             'PR' XREF A68FTN, ATAN2 'PR' 'SKIP';
           ATAN2 (S,  C) );


UNTVEC   'OP'   'E'       =  ('VEC'  U) 'VEC':                           UNIT VECTOR IN THE DIRECTION OF U.

         'INCL'   ABSVEC,  VECORS  'LCNI':

         ('REAL'  ABSU = 'ABS' U;
          (ABSU > SMALL NUMBER CLOSE TO MACHINE LIMIT | U / ABSU
                | ON INDETERMINATE VECTOR (U);  'SKIP') );


ROTVEC   'PROC'  ROTATION OF       = ('VEC'  V,  AXIS,  'REAL'  ANGLE) 'VEC': ROTATES V ABOUT AXIS THROUGH ANGLE
                                                                          ACCORDING TO RIGHT HAND RULE.
         'INCL'   VECTRS,  SUMVEC,  VECPRO,  UNTVEC  'LCNI';

         ('VEC' AX = 'E'AXIS;  'VEC' AXV = AX >< V;  'VEC' VP = AXV >< AX;
           V  +  VP * (COS (ANGLE)  -  1)  +  AXV * SIN (ANGLE));
```