

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

MATHEMATICAL CENTRE TRACTS 134

**PROCEEDINGS INTERNATIONAL
CONFERENCE ON ALGOL 68**

J.C. van VLIET (ed.)

H. WUPPER (ed.)

RECHENZENTRUM DER

RUHR-UNIVERSITÄT BOCHUM, BRD, MARCH 30–31, 1981

MATHEMATISCH CENTRUM

AMSTERDAM 1981

1980 Mathematics subject classification: 68-06,68B10,68B20

Computing Reviews: 4.22,1.52,4.34,4.12,5.23

INTRODUCTION

"What can we do with ALGOL 68?" is the question raised by the chairman of the Standing Subcommittee on ALGOL 68 Support in the first contribution to this conference. The three main topics of the conference suggest three possible answers to this question: one can teach the language, implement it, and apply it.

The papers collected in these proceedings deal with various aspects of teaching, implementation, and applications. They do not attempt to fully answer the question of what can be done, but they hopefully will provide some useful pieces of information, both to those who want to learn more about the language, and to those who are actually involved in problems in the areas addressed.

In several respects, ALGOL 68 has not been very successful. It is not heavily used except at a number of sites in Europe (most notably the UK), its new terminology has not been widely adopted, it is difficult to implement (or isn't it). But many people who know the language feel that its use to write programs reflects only one aspect, and that the more important advantages stem from its orthogonal design, carefully chosen notation, and clarity.

ALGOL 68 offers a unified view of both the user program and its environment. This combination, together with its carefully chosen notation, offers a good basis for teaching the language, and for understanding programming in general. ALGOL 68 is the only language which offers such a complete setting, which allows one to develop programs independent of the outside world.

It is our firm belief that many of the everyday problems that occur in programming are easily solved when the benefits of ALGOL 68 are taken into account. This conference may be regarded as an attempt to reveal some of these benefits.

The Program Committee for this conference consisted of

H. Ehlich (Ruhr-Universität Bochum, BRD)

C.H.A. Koster (University of Nijmegen, The Netherlands)

A.D. McGettrick (University of Strathclyde, Glasgow, UK)

S.G. van der Meulen (University of Utrecht, The Netherlands)

L. Trilling (Université de Rennes, France)

J.C. van Vliet (Mathematical Centre, Amsterdam, The Netherlands)

H. Wupper (Ruhr-Universität Bochum, BRD).

During the refereeing process, much help was obtained from P. Bakkus, A. Couvert, D. Grune, P. le Guernic, R.B. Hunter, H. Jonkers, K. Kleine, L.G.L.T. Meertens, F. Ployette, A. Quere and J. Voiron.

We thank the Director of the computer centre of the Ruhr-Universität Bochum, Prof.Dr. H. Ehlich, and his colleagues, who took care of the local organization.

We thank the Mathematical Centre for the opportunity to publish these proceedings in their series Mathematical Centre Tracts and all those at the Mathematical Centre who have contributed to its technical realization.

J.C. van Vliet

H. Wupper

CONTENTS

Session 1 (Monday, March 30, A.M.)

What can we do with ALGOL 68 (invited lecture)

S.G. van der Meulen 1

Syntactic errors made by beginners using an ALGOL 68 subset

J. André & J. Barré 17

A comparative evaluation of ALGOL 68 for programming instruction

P.R. Eggert & R.C. Uzgalis 33

Session 2 (Monday, March 30, P.M.)

Teaching with ALGOL 68 in Dresden (invited lecture)

G. Stiller 45

Semantic analysis and synthesis in the ALGOL 68 R 4000 compiler

H. Loeper, H.-J. Jäkel & H. Pietsch 59

Essay on copying

K. Wright 81

On the design of an abstract machine for a portable ALGOL 68 compiler

L.G.L.T. Meertens 97

Session 3 (Tuesday, March 31, A.M.)

An implementation of modular compilation in ALGOL 68 (invited lecture)

G.J. Finnie & M.C. Thomas 119

Programming languages for a course in data structures

V.J. Rayward-Smith 143

Context-free grammars and derivation trees in ALGOL 68

V. Linnemann 167

Session 4 (Tuesday, March 31, P.M.)

An ALGOL 68 prelude for the implementation of test generation algorithms

S.D. Butland 183

A programming system for interval arithmetic in ALGOL 68

G. Günther & G. Marquardt 201

Teaching with ALGOL 68, in Manchester (invited lecture)

C.H. Lindsey 217

List of addresses of authors 231

WHAT CAN WE DO WITH ALGOL 68

S.G. van der MEULEN

ABSTRACT

Despite its power, orthogonal design, and other nice features, ALGOL68 has not become a widely used language. Some possible reasons for this neglect by the computing community are commented upon. The paper also gives a short history of the way the language developed, and sketches a possible future. Finally, the place and tasks of the Subcommittee on ALGOL68 support are elaborated upon.

1. WHAT CAN WE DO WITH ALGOL68?

The most obvious answer to this question - and also the most natural one - of course is: We can write programs in it. And, beyond any doubt, this was precisely what we had in mind - when we designed the language, went to the limit in an attempt to define it with the utmost rigour, introduced it informally "for the uninitiated reader", implemented it with a keen eye for possible improvements, reconsidered and revised it with a keen and open eye for possible implementations, then went to an even further limit in redefining it with even more rigour and less consideration for the uninitiated, reintroduced it in an informal revision, implemented it again (and now complete and unabridged) and started to promulgate it, modelled its transport cleaning an (in spite of all efforts) still clumsy cluster of often overspecifying I/O- and file-manipulating routines, explored its possibilities through subsets and supersets and by extending it in its semantic prelude for particular application areas ----- and we wrote papers and books on it: primers, guides, treatises, conference proceedings and textbooks on various aspects such as two-level grammars, orthogonal design, recursive modes and mode-equivalencing, new implementation techniques, possible extensions --- and so on etcetera. But did we write programs in it?

An often heard objection is: There are no good compilers. But there are. And not only on CDC-CYBERS - also on IBM370s, on DEC20 and PDP11, on ICLs and other systems. Their availability may leave something to be desired (in no small measure a matter of demand), but they exist and some of them are pretty good - for teaching purposes or for production aims or for both. One of the benefits of an ambitious and well-designed language is, that the circle of its implementers is limited to competent and ambitious experts. As to demand and supply: it is a proven fact that manufacturers can be coerced into implementing the language (CDC) or supporting an existing implementation (DEC). It is - at least partly - up to those who want to write programs in it.

Another allegation is: There are no easy (and get-at-able) books on the language and its use. But again: there are. In the small library of our small computer science department we have as many titles on ALGOL68 as on any other programming language, including PASCAL which is a heavily supported language in our university. Yes, but they are difficult for the beginning programmer. Are they, all of them? It is true that The Report is unreadable for the non-expert and producing headaches with almost any one else. It is

also true that the other, more or less officially supported IFIP-WG2.1 "companion volume" - the Informal Introduction - is not a primer for the programming novice: it rather is another (and pretty precise) description of the whole language, and only informal as compared to that monstrous monument of formal rigour which is the Report. But it is equally true that most of the other primers, practical guides, introductions, tutorials and what else do we have, have been written for and are quite easy readable by any student and any intellectual who is able to think on a level not below FORTRAN.

And yet: we don't write programs in it. Or do we? Let us try to delimit more or less who are "we". Apparently, the "we" of the first paragraph is the small group of those who designed the language (IFIP-WG2.1 & affiliates), implemented it for the first time (the Malvern people of the Royal Radar Establishment), revised it (WG2.1 & affiliates & many responders), implemented it again (several groups and even individuals) and wrote about it (see bibliography) - not so many amongst them actually wrote programs in it. I now wake up to the fact that I wrote and tried my first ALGOL68 program in 1975 (shortly after the release of the CDC-CYBER implementation). The "we" of the ALGOL68-users is another "we": it is a very small subset of the total "we" that constitutes the computing community. I guess that many of you belong to that small subset, and I do (more or less), and my students and some of my scientific friends, and presumably your students and some of your friends, and - a very rough estimate - perhaps another ten times that total. A very small subset indeed. So, referring to the computing community - even if confined to the scientific computing community - we must admit: We don't write programs in ALGOL68.

And so the question arises: WHAT CAN WE DO WITH ALGOL68?

2. A PIECE OF HISTORY

Faced with the phenomenon of a programming language which is comparatively rarely used for real straightforward programming, but is nevertheless neither an obscure language, nor an unimportant language, and trying to understand this phenomenon, we must know something of its turbulent and confusing history. Whenever people start studying the history of computer science, one of the problems they cannot avoid is why programming language design always seems to generate so much heat and bitter disputes. And here is a rich source of material for at least one fascinating thesis on the

interference of scientific controversy and (only psychologically explicable) personal emotion.

When IFIP-WG2.1 in Munich, december 1968, after a sequence of exhausting meetings (more and more resembling heavy fought battles) finally accepted the "Final draft report on the algorithmic language ALGOL68" as the basis for a final "Report on the Algorithmic Language ALGOL68", there was a highly unfortunate side-output (so to say): a "minority report" of two pages. The six WG2.1 members undersigning it, certainly had their reasons - they left the group and found another base of operations in a new working group IFIP-WG2.3. (Programming Methodology). Their criticism was serious and, from their standpoint, in a sense inevitable. Notably, however, it hardly regarded the language as such - it rather was a sharp polemic against the defining mechanism. And above all: it had been dictated by anger (right or wrong: I am not writing that historical thesis).

It was a historical accident that it took quite some time before the 160-pages Report could be printed after a last finishing touch of the draft (Numerische Mathematik, Vol 14, 79-218, 1968), whereas the 2-pages minority report ran around the computing world in considerably less than eighty days - as could be expected. Consequently, the almost unreadable Report was received by a warned (if not biased) community and hardly taken for what it was: an extremely precise description of a very powerful language in need of "the crucial tests of implementation and subsequent use" (quotation from the preface to the Report). Amongst the few who actually read this remarkable document, and apparently understood it, were those who implemented it for the first time. Their ALGOL68-R compiler (Royal Radar Establishment, United Kingdom) - running as early as 1971 - without doubt, saved the language.

Reinforced by a fresh new quartet, the authors continued to reconsider and revise the language, working through a several meters high pile of corrections, improvements, suggestions, proposals etcetera - surprisingly, those who read the Report appeared to understand it quite well and sometimes showed an even deep understanding. The Malvern-experience, of course, had a great influence on this painstaking revision process. It was not before 1973 (WG2.1 meeting in Los Angeles) that the working group accepted a revised ALGOL68 and commissioned the (now eight) authors to rewrite the Report using an enriched form of the two-level grammar.

And then it took almost two years (of hard work with, and new discoveries in the expressive power of two-level grammar) for doing the job.

The outcome was a much more readable "Revised Report on the Algorithmic Language ALGOL68", though it remained a visitation for the application orientated (and thus in particular respects naive) potential reader. The first edition appeared late in 1975 - shortly after the release of its first full implementation (on the CDC-CYBER). Not many seem to know that what we still call ALGOL68, in fact is ALGOL73, which was not available before 1976. I think it was a psychological mistake not to call it ALGOL73, or even ALGOL75. Anyhow, in a rather precise sense, ALGOL68 is an at least five years younger language.

In the mean time, however, two very important events took place: PASCAL conquered the world and - not unlikely - ADA may take over on the longer run. Again we must ask ourselves: WHAT CAN WE DO WITH ALGOL68 ?

3. A POSSIBLE FUTURE

It is interesting to investigate in which sense ALGOL68 might be conceived as "a language somewhere between PASCAL and ADA". Clearly, historically, it is entirely wrong: ALGOL68 existed years before the brilliant didactician and compilerbuilder Niklaus Wirth created PASCAL. Moreover, the three languages had different (though in some sense also similar) design-objectives:

- The programming language PASCAL was originally developed for teaching programming, with an emphasis on the techniques known as structured programming.
- ALGOL68 is designed to communicate algorithms, to execute them efficiently on a variety of different computers, and to aid in teaching them to students.
- ADA is a programming language for numerical applications, systems programming applications, and applications with real time and concurrent execution requirements.

On the other hand, PASCAL can be closely approximated by a small subset of ALGOL68 (Wirth is a wise man, knowing his own and his students limitations), and ADA (borrowing its notation from PASCAL) borrowed quite a lot of basic ideas from ALGOL68 (but, alas, not the great binding principle of orthogonal design). It certainly is no act of usurpation when we classify both PASCAL and ADA under the ALGOL-like languages (though their spiritual fathers seem not to like, or perhaps even not to understand the very spirit of our ALGOL). A few remarks seem to be relevant in this context:

- PASCAL is not really the great language for programming education. Of course it is much much better than BASIC, it is a relief after FORTRAN, and it is an improvement upon ALGOL60 (though certainly not in every respect). However, if weighed against for example SIMULA67, I would prefer the latter: the SIMULA class-concept is much more important and interesting for scientific education than the PASCAL-"set".

- PASCAL's frequently advertised strength (i.e. its wise limitations), is also its weakness. It may be a nice, and easy, and even charming language for the beginner - but once he knows more or less what is programming about, he bumps his head against some missing feature (which he cannot create so easily) and he has to switch over to other languages for the real programming tasks, the real software design. I also have some difficulty with the term "easy language": that is not the real issue - what we need are languages that elucidate the process of program design in-general, and precisely here PASCAL is a too restricted language.

- PASCAL's actual strength is its implementability: it easily beats FORTRAN in this respect (even with runtime-efficiency). Indeed - recently, as programmers in business and industry at last have begun to discover the severe limitations of traditional programming languages, interest in putting PASCAL to work outside the classroom has increased. But THE important boost for PASCAL has come from its widespread implementation on microcomputers. PASCAL is becoming one of the standard languages that every programmer should know, for THAT reason!

- ADA is too young (and also too undefined at present) to be evaluated with respect to its design objectives. In view of the military power behind it (quo vadis) and the inevitable pressure on computer industries as a consequence, ADA will be "doomed to succeed" (a down-to-earth forecast of Gerhard Goos).

Now the situation becomes clearer: whether we like it or not, the future computer scientist must be familiar with at least FORTRAN (of course), PASCAL and ADA (and very likely also some assembler language, and PL/I, and LISP, and SIMULA, and name it yourself). It then is my conjecture, even my conviction, that a well-chosen subset of ALGOL68 is far and away the best language to begin with (in scientific education, and in particular at universities). Here are my arguments:

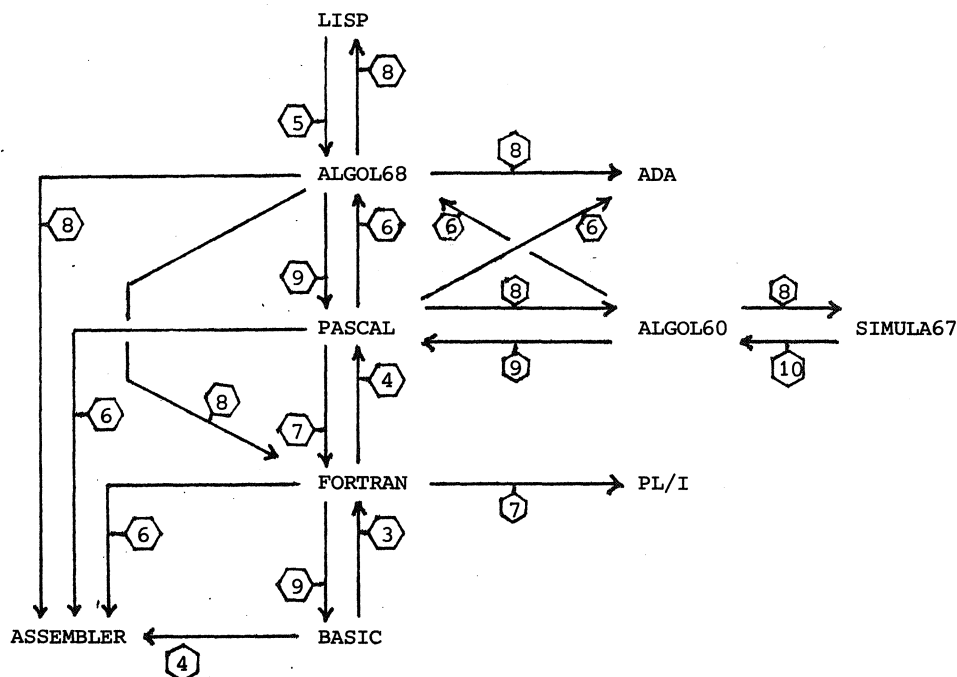
- A Pascal-size subset of ALGOL68 is easier to learn and to use than PASCAL itself (orthogonality = inherent logic).

- Such a subset is an excellent starting point for learning the whole language.
- ALGOL68 is, in a restricted but clear and useful sense, an extensible language (cf. TORRIX, GRAPHEX68 etc.).
- ALGOL68 has machine-oriented primitives (BITS, BYTES and the usual operations on these): low-level machine-programming can, in no small measure, be taught entirely within ALGOL68 (try it!). From there to any assembler language is hardly more than a few small steps.

And here is my main argument:

- If you have learned one language (A), and have to switch over to another language (B), the easiest transitions are always those where A=ALGOL68. There is only one exception, namely the trivial situation where B is (in some sufficient sense) a sublanguage of A.

We did a bit of home-cooked sociological research in an attempt to find support for this statement, asking (over quite some period of time; the inquiry continues) every student we met learning a new programming language, to give us an estimate of how difficult/time-consuming it was. The following transition scheme reflect our findings:



The digits have more or less the following meaning:

- 10 = without any difficulty (transition to a sublanguage)
- 9 = very easy
- 8 = with a little bit of effort
- 7 = it took me some time
- 6 = I had to work on it
- 5 = I found it difficult
- 4 = the language I knew did not help me at all
- 3 = the language I knew was a real source of errors
- 2 = the language I knew was my main difficulty
- 1 = I had to unlearn before I could start to learn

For a good understanding: this transition scheme cannot be considered, nor used, as the outcome of a carefully designed scientific inquiry. It should also be pointed out, that all transitions are isolated (as soon as a guy knows more than one language the inquiry may lose its meaning!). Yet, the figures seem plausible.

If they are true (and I think there may be quite some truth in them), then they are important for everyone who has to take decisions in planning computer science curriculae. Please, do your own inquiries, if you are in a position where you can find students who started with FORTRAN, or PASCAL, or ALGOL68, or even with LISP, and have the opportunity to learn one of the other languages. And tell me about your findings.

4. ALGOL68-EDU

This is the PASCAL-size subset mentioned above: EDU = Educational Decision (university of) Utrecht - or simply an abbreviation of EDUcational. ALGOL68-EDU is the orthogonal span of ALGOL68 without:

```

label-definition , GOTO , GO , EXIT ,
PAR , SEMA ,
OP , PRIO ,
LONG , SHORT , FLEX , UNION ,
COMPL , BITS , BYTES ,
FORMAT , FILE , CHANNEL ,

```

and every syntactic construct or operator, that comes in the wake of it. That is, the following bold-symbols do not occur in the subset:


```

ARG BIN BITS BYTES CHANNEL COMPL CONJ DIVAB DOWN
ELEM EMPTY EQ EXIT FILE FORMAT GE GO GOTO GT I
IM LE LENG LEVEL LONG LT MINUSAB NE OP OVERAB
PAR PLUSAB PLUSTO PR PRAGMAT PRIO RE SEMA SHL SHORT
SHORTEN SHR TIMESAB UP UNION

```

we use however:

```

+:= -:= *:= /:= +=:
** <= >= ""

```

For a detailed description, a 1½-level grammar in syntactic diagrams and a simple semantic explanation, we refer to the "ALGOL68-EDU REFERENCE MANUAL", presumably available during this conference or soon thereafter.

We now have an almost six-year experience with this subset (known as SPEEDY68 before it stabilized): about 250 students learned programming with it. The results seem to be quite satisfactory. Our most difficult group is the FORTRAN-programmers who want to take computer science courses and have to unlearn their clumsy FORTRAN-statement style of writing programs.

We do not have an ALGOL68-EDU compiler - we use the full-ALGOL68 implementation on the CDC-CYBER. We would like to have an ALGOL68-EDU preprocessor (for more didactic error-messages and -warnings), but do not really need one - many students start to use non-ALGOL68-EDU features from the full language, anyhow (and right they are). EDU is rather a pedagogical advice than a sublanguage in its own right. We believe that an EDU-compiler will not be an enormous piece of work, and if it is a portable compiler (I mean really portable) it may become a milestone in computer science education at universities and a blessing for the language.

5. WHAT CAN WE DO?

Preliminary question: What did we do?

Immediately after the Los Angeles decision on the revision of the language, a standing subcommittee on ALGOL68 MAINTENANCE AND SUPPORT was set up; it still exists. Its first task became, in the natural course of things, to assist the 4+2+1+1 group of revising authors in their painstaking labour. When this at last came to an end (there are ten years between the very first draft and the final revision), the subcommittee more and more found its proper working domain:

- Clarification of the Report where necessary.

- Correction of printing errors and other mistakes.
- Discussion and final judgement of sublanguages and "super-language"-features.
- Promulgation of the language.

Under "Clarification -----" we find important pieces of work such as Hans van Vliet's ALGOL68 TRANSPUT model, and Wilfred Hansen & Hendrik Boom's THE REPORT ON THE STANDARD HARDWARE REPRESENTATION FOR ALGOL68. Under "Correction -----" fall all more or less bureaucratic activities - not particularly amusing, but certainly necessary. Under "Discussion -----" fall the more creative activities, such as Peter Hibbard's definition of ALGOL68-S, and Charles Lindsey & Hendrik Boom's proposal for library modules. Finally, under "Promulgation -----" we find all other things to be done.

I feel uneasy with regard to this promulgation business. Clearly, there is something to be done:

- the compilation and updating of a complete ALGOL68 bibliography,
- the registration and updating of all available implementations, and where they come from, and who maintains them,
- the issue of a NEWSLETTER.

Our host in Bochum - the Ruhr Universität - made us an offer, already more than a year ago: to print and distribute an ALGOL68 NEWSLETTER. Until now we reacted hesitating and even reluctant - why? You see, we have our problems, and one of these is the ALGOL BULLETIN.

Should the ALGOL BULLETIN become a newsletter, or rather contain a newsletter, or should a NEWSLETTER become a new activity? If so, is it then wise to accept the generous Bochum offer, or should we try to use SIGPLAN NOTICES (provided they let us use them), or some other well established channel? The real important question, however, is (and that is the very root of our ambivalence): who is taking responsibility for the contents of this newsletter, and who guarantees regularity in appearance? The subcommittee? Or a subsubcommittee? Or no committee at all - a volunteer?

Being here together for two days, I invite you all to discuss with us the pro's and con's and why's and how's of a NEWSLETTER and - more generally - how to promulgate a language if you are a nice crowd of individuals, rather than a big manufacturer or a mighty pentagon.

The question is: WHAT CAN WE DO WITH ALGOL68 ?

TENTATIVE BIBLIOGRAPHY

This is a very incomplete, and also inaccurate list of titles on ALGOL68. Its sole purpose is, to encourage the reader to supplement and correct it, where possible. Please send your contribution(s) to:

S.G. van der MEULEN
 Department of computer science
 the University of Utrecht
 Princetonplein 5, P.O.Box 80.002
 3508 TA UTRECHT, The NETHERLANDS

- Ammeraal, J., *Mini ALGOL 68 User's Guide*, IW 32/75, Mathematical Centre, Amsterdam.
- Bacchus, P., J. André and C.H. Pair, *Manual of the Algorithmic language ALGOL68*, Paris, France.
- Bachmann, K.H., *Die Programmiersprachen PASCAL und ALGOL68*, Akademie-Verlag, Berlin (DDR), 1976.
- Bauer, F.L. and G. Goos, *Informatik Teil I und II*, Heidelbergse Taschenbücher BD. 80, 91, Springer Verlag, Berlin - Heidelberg - New York, 1971.
- Birrell, S.R., *ALGOL68C Implementers' Guide*, ALGOL68C technical report, University of Cambridge Computer Laboratory.
- Bourne, S.R., *ZCODE A simple Machine*, ALGOL68C technical report, University of Cambridge Computer Laboratory.
- Bourne, S.R., A.D. Birrell and I. Walker, *ALGOL68C Reference Manual*, University of Cambridge Computer Laboratory.
- Brailsford, D.F. and A.N. Walker, *Introductory ALGOL 68 Programming*, Ellis Horwood Publishers, Chichester, 1979.
- Branquart, P. et al, *An Optimized Translation Process and its Applications to ALGOL68*, Lecture Notes in Computer Science 38, Springer-Verlag, Berlin, 1976.
- Branquart, P., J. Lewi, M. Sintzoff and P.L. Wodon, *The Composition of Semantics in ALGOL68*, Communications of the ACM, Nov. 1971, vol. 14, no. 11.

- Cleaveland, J.C. and R.C. Uzgalis, *Grammars for Programming Languages*, Elsevier North Holland, Inc., 1977, Amsterdam.
- Colin, A.J.T., *Programming and Problem-solving in Algol 68*, The MacMillan Press Ltd., 1978, London.
- Denert, E., G. Ernst and H. Wetzel, *GRAPHEX68 - Graphical Language Features in ALGOL68*, Computer and Graphics I (1975) 195-202.
- Denert, E. and R. Franck, *Datenstrukturen*, Bibliographisches Institut AG, 1977 1977, Zürich.
- Denert, E., R. Franck, F. Müller and W. Streng, *Graphische Sprachenelemente in ALGOL68*, Technische Universität Berlin, Fachbereich Informatik, Bericht nr. 73-01, Jan. 1973.
- Differences between Algol 68-R and 68*, RRE Internal Document (2nd. Edn. June, 1973).
- Feldman, H., *Einführung in ALGOL68*, Skriptum für Hörer aller Fachrichtungen ab. 1 Semester, Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 1978, Braunschweig.
- Gerbier, A., *Mes premières constructions de programmes*, Lecture Notes in Computer Science, Vol. 55, Springer-Verlag, New York, Heidelberg, Berlin, 1977, (in French).
- Groupe Algol de l'AFCEP, *Manuel du langage algorithmique Algol 68*, Hermann, Paris, 1975.
- Groupe Algol de l'AFCEP, *Définition du langage algorithmique Algol 68*, Hermann, Paris, 1972.
- Grune, D., *The MC Algol 68 Test Set*, Mathematisch Centrum, IW-53, 1975, Amsterdam.
- Grune, D., L.G.L.Th. Meertens and J.C. van Vliet, *Grammar-handling Tools Applied to ALGOL68*, Mathematisch Centrum, 1973, Amsterdam.
- Hedrick, G.E. (ed.), *Proceedings of the 1975 International Conference on ALGOL68*, Oklahoma State University, Stillwater, Oklahoma, June 10-12, 1975.

- Hill, U., H. Scheidig and H. Woessner, *An Algol 68 Compiler*, Technische Universität München and University of British Columbia, 1972.
- Housden, R.J.W. and V.J. Rayward-Smith, *Conference Proceedings "Experience with Algol 68"*, 1975, Liverpool.
- Kelk, B.C., *A plotter package for ALGOL68C*, CAD Group Document 90, Computer Laboratory, University of Cambridge, 1976.
- Koch, W., and Ch. Oeters, *An Abstract ALGOL 68 Machine and its Application in a Machine-independent Compiler*, J. Mühlbacher (ed.): GI-5. Jahrestagung (LNCS 34), Springer 1975.
- Koster, C.H.A. and Th.A. Zoethout, *Systematisch programmeren in algol68*, Kluwer - Deventer 1978.
- Learner, A. and A.J. Powell, *An introduction to ALGOL68 through problems*, Macmillan Computer Science series, MacMillan Press, London, 1974.
- Lindsey, C.H., *ALGOL68 with fewer tears*, The Computer Journal, Vol. 15, no.2, 1972.
- Lindsey, C.H. and S.G. van der Meulen, *Informal Introduction to ALGOL68*, revised edition, North Holland Publ. Company, 1977, Amsterdam.
- MacGettrick, A.D., *ALGOL68 - A first and second course*, Cambridge University Press, 1977.
- Maybrey, A., *Proceedings of Liverpool Conference on Uses of Algol 68*, March, 1975.
- Meertens, L.G.L.T., *A Space-saving Technique for Assigning ALGOL 68 Multiple Values*, Mathematical Centre, 1976, Amsterdam.
- Meulen, S.G. van der, and P. Kühling, *Programmieren in ALGOL68*, Vol. I & II, Walter de Gruyter, Berlin, 1974, 1976 (in German).
- Meulen, S.G. van der, and M. Veldhorst, *Torrax; A programming system for operations on vectors and matrices over arbitrary fields and of variable size, vol. I.*, M.C. Tracts no. 86, Mathematical Centre, 1978, Amsterdam.
- Meulen, S.G. van der (et al), *ALGOL68-EDU: A REFERENCE MANUAL FOR STUDENTS*, RUU-CS-81-**, University of Utrecht.
- Pagan, F.G., *A Practical Guide to ALGOL68*, John Wiley & Sons, London.

- Paillard, J.P. and M. Simonet, *Attribute-like W-grammars*, The fifth annual international conference on the implementation and design of Algorithmic Languages, Rennes, May 1977.
- Peck, J.E.L. (ed.), *ALGOL68-Implementation*, North-Holland Publishing Company, Amsterdam, 1971.
- Peck, J.E.L., *An ALGOL68 Companion*, Department of Computer Science, University of British Columbia, Vancouver 8, B.C., Canada, 1972.
- Peck, J.E.L. (ed.), *Proceedings of an Informal Conference on ALGOL68-Implementation*, University of British Columbia, Vancouver 8, B.C. Canada.
- Proceedings of International Conference on ALGOL68 Implementation*, Utilitas Mathematica Publishing Inc., University of Manitoba, Winnipeg, 1974.
- Rayward-Smith, V.J. (ed.), *Proc. Conf. on Applications of ALGOL68*, 80-90.
- Robertson, A. and G.E. Hedrick, *A Portable Compiler for an ALGOL68 Subset*, Proceedings of the 1975 International Conference on ALGOL68, Oklahoma State University, Stillwater, Oklahoma, June 10-12, 1975.
- Sigplan Notices, ACM, Vol. 12, Numb. 5, May 1977
1. Wijngaarden, A. van et al., *Revised Report on the algorithmic language ALGOL68*.
 2. Hibbard, P.G., *A sublanguage of AALGOL68*.
 3. Hansen, W.J. and H. Boom, *The report on the standard hardware representation for ALGOL68*.
- Sigplan Notices, *Proceedings of the Strathclyde Algol 68 Conference*, ACM, vol. 12, numb. 6, June 1977.
- Sintzoff, M., *A brief review of Algol 68*, Algol Bulletin, no. 37, July 1974.
- Stiller, G., *ALGOL68-Begriffe und Ausdrucksmittel*, B.G. Teubner, Verlagsgesellschaft, Leipzig, 1974, Lizenzausgabe im R. Oldenbourg Verlag, München, Wien, 1974.
- Stiller, G., *ALGOL68-Datenorganisation*, R. Oldenbourg Verlag, München, Wien, 1976.

Tanenbaum, A.S., *A tutorial on ALGOL68*, ACM Computing Surveys 8 (june 1976) 155-180.

Vansina, C., *Manipulation dynamique de valeurs en algol68*, June 76, Mémoire de maîtrise en informatique, FNDP Namur, Belgium.

Vliet, J.C. Van, *Algol 68 transput*, Mathematical Centre Tracts 110&111, A'dam

Woodward, P.M. and S.G. Bond, *ALGOL68-R Users Guide*, Division of Computing and Software Research Royal Radar Establishment (RRE), Malvern, England, 1972.

Wijngaarden, A. van, et al., *Revised Report on the Algorithmic Language A ALGOL68*, M.C. tract 50 Mathematical Centre, 1976, Amsterdam; Springer Verlag, New York, Heidelberg, Berlin, 1976; Sigplan Notices, ACM, Vol. 12, Numb. 5, May 1977; Acta Informatica, Vol. 5, pts. 1, 2, 3, 1975.

Zosel, M.E., *A Formal Grammar for the Representation of Modes and its Application to ALGOL 68*, Ph.D. Thesis, University of Washington, 1971.

SYNTACTIC ERRORS MADE BY BEGINNERS
USING AN ALGOL 68 SUBSET

J. ANDRÉ & J. BARRÉ

ABSTRACT

First year students at the University of Rennes were introduced to Computer Science by means of an ALGOL 68 subset. This experiment was followed up on a pedagogical level by an analysis of the syntax errors made by the students. The analysis enabled us to improve teaching and to modify certain details in the subset's design. In particular, the results show that the students do not make more errors in ALGOL 68 than they do in other languages, with the exception perhaps of errors on representations.

1. INTRODUCTION

ALGOL 68 is fascinating because of its orthogonality which, starting from a limited number of basic concepts, gives a great freedom of expression to the programmer. But many teachers have remained sceptical as far as its effective use as an introductory programming language [20]. It's true that for a long time the lack of pedagogical documents and the presentation of the design grammar did not help things.

Nevertheless, a team at the University of Rennes, convinced of the important contribution of the language, defined, implemented and then used for several years an ALGOL 68 subset to introduce first-year students to programming.

It seemed interesting to us to follow the difficulties encountered by the students very closely. This led us to analyse the errors made in their programs. For material reasons, only syntactic errors were the object of this study. The most striking result is that the beginner students don't make more mistakes in ALGOL 68 than they do in other languages (except perhaps on representations).

After discussing the pedagogical context and the procedures of this study, results will be analysed and remarks about the definition of languages and the psychology of their learning will be presented.

2. THE CONTEXT OF THE STUDY

Our study took place over two scholastic years (1976-77 and 1977-78). It was done at the University of Rennes (France) and concerned the DEUG A (General University Studies Diploma, Mathematics and Physical Science majors) which correspond to the first year at the University after the "baccalauréat" (high school diploma). The teaching program was mostly Mathematics (260 hours) and Physics (160 hours). Computer Science only represents 9% of the test grade for 37 hours of classes (this situation does not motivate the students!). So we imposed the three following goals on ourselves:

- to establish as many liaisons as possible with the math program,
- to give to the future users of Computer Science (for applications in physics, chemistry, ...) the algorithmic bases which are hidden when confronted by a language like Fortran,
- to present to these students, who have never done Computer Science (and who maybe never will again) the limits of a computer.

The 37 yearly hours of Computer Science were divided as follows:

- 12 hours of lecture classes for the 450 students. This course presented generalities about Computer Science and computers (architecture, language, machine ...)
- 25 hours of "directed work", by small groups of 20 students during which algorithms and an ALGOL 68 subset were taught. The results of these exercises formed the subject of this study.

3. THE LANGUAGE USED

For an introduction to programming we thought it judicious not to use the ALGOL 68 ideas in their entirety. An ALGOL 68 subset, called SERA, was therefore defined and implemented [18].

SERA is a language whose power of expression is similar to that of ALGOL 60, but with the orthogonality and the syntax of ALGOL 68. Leaving aside the pedagogical aspects, it should be noted that the advantage of this language lies in the savings, of time and space, of the SERA compiler compared to the ALGOL 68 compiler.

4. TEACHING METHODOLOGY

The language concepts, as well as the work of a working group on the process of learning programming [7], enabled us to put forward the following points:

- 1) the emphasis on the distinction between identifiers, values and references
- 2) the association of data structures and control structures:
 - i) use of variables (and of assignment) only in loops. This point is especially well adapted to ALGOL 68 thanks to the identity declaration (but would cause problems in Pascal and Fortran)
 - ii) use of the for loop when using arrays.

Each student had to write four programs related to important points in the teaching program:

- the first having to do with sequentiality, unformatted input and output,
- the second on conditional clauses and case clauses,
- the third on arrays, loops and procedures,

- the fourth on arrays and procedure parameters.

Running the programs on the machine was done with punched cards. The students coded a form and gave it to the punching department. The corrections were their (the students) responsibility.

5. ERROR COLLECTION METHOD

A completely automatic system which would have discovered and analysed mistakes detected at compile time wasn't possible. In fact, if you don't happen to process an especially intelligent compiler, mistakes are not always brought to light. Let's consider the following lines:

```

ref int A, B, C = loc int;
ref int LETTERCOUNT = loc int;
...
LETTERCOUNT := 0;
A:= B C:= C+1;

```

A regular compiler would report, for the last two lines: UNDECLARED IDENTIFIER whereas in the first case it's a typing mistake (an I instead of a T) and in the second there is a semi-colon missing which should separate the two clauses.

We therefore had to analyse the programs de visu one by one. The erroneous listings were obtained by simply copying the compiler's output statements for error-messages on two files, one in normal output, the other on a special file, globally listed at the end of each batch session.

So only the programs having syntactic errors were analysed, which puts certain limits on our study:

- we don't have reports of certain expressions which, even if they are not mistakes, show an incomplete knowledge of the language on the student's part.

For example:

```

if A = 0 then B:= B+1 else B:= B fi
WRITE((LINE))

```

In the same way,

```
int perimeter = length + Width * 2
```

comes, in our opinion, more from a syntactic error (bad knowledge of operator priority rules) than semantic. But such errors could only be discovered accidentally, on the occasion of a real syntactic error. Their number is therefore doubtful!

The duplication of output listings was not done at runtime. So we have neither the accounts for mistakes reported by the runtime system, nor semantic errors. This also limits the scope of our study because it is known that syntactic mistakes represent only 20% of programming errors [2,24]. Furthermore, the number of runs per student and per exercise has not been studied.

6. RESULTS

Our study took place over two years and deals with the analysis of listings corresponding to four different problems. We will first present the results relative to the first year:

Number of erroneous programs studied	:	327
Number of mistakes discovered	:	708
Average number of lines per erroneous program	:	45
Average number of mistakes per erroneous program:		2.16

6.1. Breakdown of errors

Table 1 shows the errors found. This classification does not follow the ALGOL 68 Report [23] but more of a pedagogical order. The details of the mistakes are given in [1].

6.2. Frequency of errors

Figure 2 presents the breakdown of errors according to their decreasing frequency. More than half the mistakes are punching, representation or punctuation errors.

TYPE OF ERRORS	OCCURRENCE NUMBER	TOTAL NUMBER	PERCENTAGE
ENVIRONMENT Language limitations Standard identifier Others	32 9 9	50	7.0
MIXED CARDS		14	1.9
PUNCHING Punching errors Bugs in forms filled out	49 19	68	9.6
REPRESENTATIONS Symbols Strings Comments Operators Numbers Others	44 71 23 14 1 17	171	24.1
SIMPLE DECLARATIONS Identifiers Syntax Others	17 7 4	28	3.9
CALLS TO WRITE/READ Balancing Mode Others	10 26 17	53	7.5
EXPRESSIONS Arithmetic String Boolean Parenthesis	2 1 15 9	27	3.8
GENERAL CONTROL STATEMENTS Clauses Conditional clause Loops Case clause	12 15 9 6	42	5.9
PUNCTUATION Missing semicolon Extra semicolons Others	108 36 5	149	21.0
ARRAYS		16	2.3
PROCEDURES		11	1.6
MODES AND SCOPE Declaration simple General constructions Arrays Procedures	6 17 9 17	49	6.3

Table 1 - Distribution of errors found in 1976-77 study

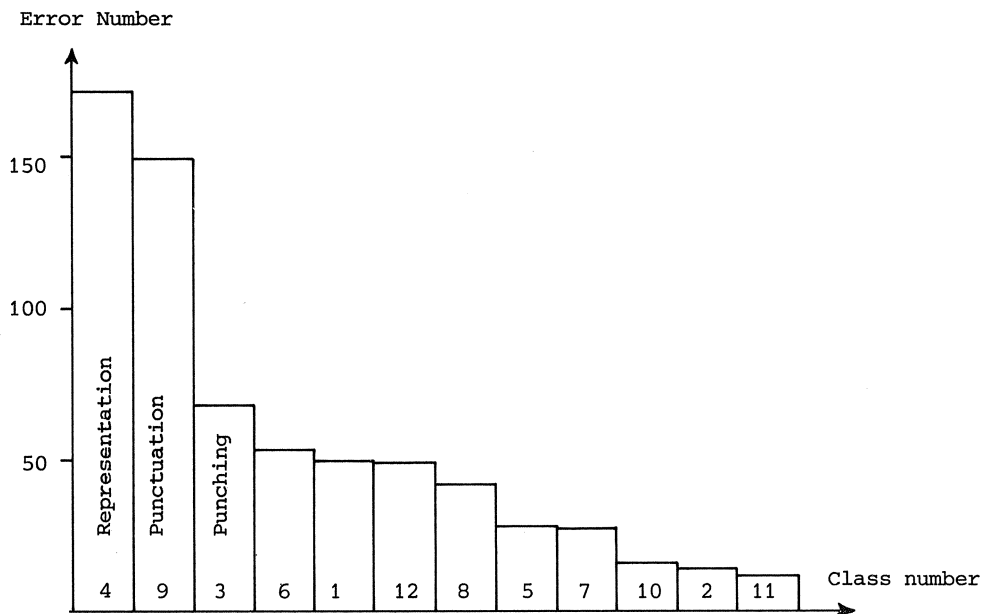


Fig. 2 - Breakdown of errors according to frequency

Classification number

- | | |
|---------------------------|--------------------------|
| 1. Environment | 7. Expressions |
| 2. Card problems | 8. General constructions |
| 3. Punching | 9. Punctuation |
| 4. Representation | 10. Arrays |
| 5. Simple declarations | 11. Procedures |
| 6. Calls to WRITE or READ | 12. Modes and scope |

6.3. Analysis of errors

Even though there are accounts of mistakes made in various other languages (e.g. BOIES & GOULD [2], GANNON & HORNING [6], LITECKY & DAVIS [15], YOUNGS [24] ...) it didn't appear worthwhile to make any comparisons with these studies because they seem to have been done under different

conditions than ours.

In this section underlining is used for bolds, except when useful to exhibit the actual hardware representation.

The first fundamental remark is that it doesn't seem as if ALGOL 68's syntax is the cause of special errors. It is true that the grammar taught was not that of the design [23] but was presented in the form of a classic flowchart. However we will see that even balanced constructions (if ... fi, case ... esac, etc) were the source of very few errors.

Even though the problems proposed to the students were relatively simple, it can be said that certain specific ALGOL 68 concepts, for instance the passage of parameters to procedures, resulted in very few errors (less than 2%). Notable, too, the explanation of the mechanism takes much less time with ALGOL 68 than with Pascal or ALGOL 60.

The choice of the strict form for variable declarations (which set off these objects very well) was not responsible for a significant amount of errors (1%).

6.4. Balanced constructions

Balancing in control structures doesn't cause any problem: the representations used were in French (inspired by [3]). In particular, the choice of closing keywords didn't mirror those of ALGOL 68 (if ... fi, etc) but were all constructed with the prefix f, abbreviation for fin (end), as KOVATS [13] now suggests and as it is in ADA [11].

<u>si</u> ... <u>fsi</u>		<u>if</u> ... <u>fi</u>
<u>cas</u> ... <u>fcas</u>	for	<u>case</u> ... <u>esac</u>
<u>faire</u> ... <u>fait</u>		<u>do</u> ... <u>od</u>

This orthogonality, and, without doubt, also the fact that the students were beginners in Computer Science, are probably the reasons why we only found two examples of bad balancing (if without fi).

Nevertheless, we have to point out that some errors were related to elif (whose French representation, sinsi, is not any clearer): this symbol is not very structuring when there are several embedded conditional clauses.

6.5. Mistakes related to WRITE calls

The united modes were not explicitly part of the SERA language. Some of ALGOL 68's concepts, like the WRITE and READ parameters, were nevertheless connected with them, but could not be clearly explained to the students.

The parameter mode of the WRITE procedure, and that of READ (less often used), was thus the cause of some parenthesis errors:

```
WRITE (A,B) instead of WRITE ((A,B)).
```

We also found several examples of a mistake that is difficult to explain to debutants (collateral in a weak context not allowing union):

```
WRITE (if DELTA = 0 then ("1 root :", X)
      else ("2 roots :", X1, X2) fi)
```

In other respects, several times we found something equivalent to:

```
proc P = (...) string : WRITE (...)
```

This confusion between the WRITE procedure (neutral mode) and the "text" which is finally printed comes without doubt from the difficulty beginners have in distinguishing between a procedure which prints a value and a procedure which returns a value, a difficulty already mentioned by LEITNER & LEWIS [14].

6.6. The most numerous mistakes

It is therefore not at the level of syntactic constructions, but, foolishly, at the lexicographic level that the students made the most mistakes: as seen in table 2, 50% of the mistakes found have to do with punctuation or representation problems:

The representation mistakes are the most numerous (around 25%). Basically, they concern forgetting one or (more often) two quotes for the symbols (e.g. 'BEGIN, END', AND instead of, respectively, 'BEGIN', 'END', and 'AND'). They are always occasional mistakes (not one program, for instance, was written completely without quotes). It would be interesting to

see how much this rate of mistakes would decrease with the new official rules which allow the writing of begin as .BEGIN or even BEGIN [8].

The string notations are also the object of many mistakes in particular forgetting (often systematically in the same program) quotation marks. This happens, moreover, most frequently in the WRITE parameter, e.g. WRITE (NO ROOTS), or in an identity declaration, e.g. 'CHAR' ENDOFTEXTE = # instead of 'CHAR' END OF TEXTE = "#".

Comment notations are the cause of mistakes either in the representation of co ('CO or CO', etc. instead of 'CO'), or for questions of program presentation, like

```
'REF' 'INT' LETTERC = 'LOC' 'INT' ; 'CO' LETTER COUNT
'REF' 'INT' WORDC  = 'LOC' 'INT' ;      WORD COUNT 'CO'
```

The absence of balancing symbols for certain constructions such as comments (co ... co and not co ... oc) or certain notations (" ... " strings when in ALGOL 60 there were two different symbols) is the reason why some trivial errors (co forgotten or incorrectly punched) have enormous repercussions on the rest of the program because a permutation occurs between what is a comment and what is not, and this happens in almost all languages (SCOWEN & WHICHMAN [22]), except in ADA [11].

Punctuation mistakes are linked to the use of semi-colons. They are numerous (21%), but far from surprising! They occur with about the same frequency in COBOL programs written by debutants (LITECKY & DAVIS [15]), but also in programs written by high-level students in TOPPS and TOPPS II (GANNON & HORNING [6]).

On this subject, several remarks should be made:

- the logical sequentiality of instructions is not implicated: we only discovered one absence of a semi-colon to separate two serial clauses in a single physical line;
- but the students confuse logical and physical sequentiality (explicit for them because of the physical ends of lines). As a matter of fact, two-thirds of errors have to do with the absence of semi-colons at the ends of lines. Several authors (HOLT [10], GANNON & HORNING [6], GANNON [5], PAGAN [19]), have already reported the importance of semi-colons. Some compilers (like those of BCPL or ALGOL 68 at Oxford - cited by MIDDLETON [16]) consider that in certain contexts the physical end of a line corresponds to a semi-colon.

- finally the absence of the empty instruction, like in ALGOL 60, is the cause of around 40 errors of excessive semi-colons (before else, fi, end, etc.) i.e. of missing void.

Punching mistakes (and especially those in the identifiers) are less numerous here than those discovered in COBOL by LITECKY & DAVIS [15]. The probable reason is that the identifiers are often (as much by the nature of the problems as by habit) longer in COBOL (where a frequent identifier will be of the form NEW_PAYROLL_FILE) than those of scientific disciplines (where the identifiers are in this form: T(I) or X). Forty percent of the mistakes come from misuse of the shift key on the IBM 29 card punch machine (e.g. HDEBUT' and X: = YF instead of 'DEBUT' and X: = Y;) and 25% of them seem to be related to bad handwriting: |FI' instead of 'FI' or P(X'Y) instead of P(X,Y).

7. FOLLOW-UP OF THE FIRST ANALYSIS

The mistakes cited above correspond to those found in 1976-77. Their analysis as of the end of the scholastic year allowed us to improve certain things in the SERA language and in teaching.

We quickly found that two kinds of errors had the SERA language itself as a cause:

- 1) the prologue contained procedures defined according to the schema:

```
proc GET m = m : (m X ; READ(X) ; X)
```

but in a non-orthogonal way: GETREAL had GETREL as an identifier; and GETCHAR didn't exist.

- 2) The comment notation co was 'CO', that is, 4 characters to write or punch and, therefore, a great amount of possible errors. This symbol was replaced by # (which is, by the way, legal in ALGOL 68).

In other respects, certain figures in table 2 seemed abnormal and we make a pedagogical effort in 1977-78 on some special points (in particular on boolean mode expressions, string notations and WRITE calls).

The 1977-78 listings were, in turn, studied. The same magnitude of errors were found, with, however, appreciable changes for the columns of table 3:

	1976/77	1977/78
Modifications of SERA and its compiler		
Comment	23	5
Prologue procedures	9	0
Modifications in teaching		
Strings	71	39
WRITE call	26	5
Boolean expression	15	5

Table 3: Evolution of errors in terms of improvements brought about in 77-78 to SERA and teaching.

(The 77-78 numbers are adjusted to the total number of 76-77 errors. Percentages would be very low).

8. IDIOSYNCRACIES IN PROGRAMMING LANGUAGES

Programming languages have a certain style that is neither completely natural, nor completely mathematical. Since programming languages are, all the same, close to natural languages, debutants have a tendency to not follow the proposed syntax. This is probably the origin of a certain number of errors like the following:

```

if DELTA < 0 WRITE ("NO ROOTS") fi      co missing then co
until char = "." do ... (instead of while char ≠ "." do ...)

```

This latter form has, by the way, been proposed by KNUTH [12].

In the same way mistakes are found that wouldn't be mistakes in other languages, like:

```

WRITE X
X:= a if new char = "-" then - else + fi b;

```

Two classes of errors are also attributable to arbitrary choices made in algorithmic languages which deceive mathematics students. The first is to limit the sense of and to only logical inclusion. The students then "naturally" write things like:

```

if c = " " then count:= 0 and sum += 1 fi
WRITE (X and Y)

```

The second class is characterized by the absence of certain notations, in particular those of set theory from which, excusably, the following:

```

for 1 ≤ i ≤ n do ... (i.e.  $\forall i \in \{1 : n\}$  do ...)
if day = 28 or 30 or 31 then ... (i.e. if day ∈ {28,30,31} ...)
if 2 ≤ c ≤ 80 then ...

```

9. CONCLUSION

Pedagogical models or psychological studies of the process of learning Computer Science have made much progress in the last few years (see, for example, FURUTA [4], HOC [9], LEITNER [14], YOUNGS [24], SCHOOMAN & BOLSKY [21]). Nevertheless, it's regrettable that few such studies have made a distinction between the learning of algorithms and of a programming language, and that none tackle the languages at the level of ALGOL 68. It is interesting to bring up again the fact that our pedagogics were found to be at fault on the one hand for a lack of orthogonality (in the prologue) and on the other hand because of a hidden concept (the union mode of the WRITE procedure). In both cases it seemed clear, at least according to the account of errors, that the students were confused.

Such statistics should be collected for other languages also and should be extended in such a way as to answer questions like: what is the total number of runs per student per exercise? How do errors differ if programmers are not beginners? How many of the errors can be attributed to factors other than the design of the programming language? And of course they should be concerned with semantic errors as well.

In conclusion, we should bring up once again the great number of errors related to representations and say that we think this is due to the fact that this subject was always scorned by theoreticians. There was an 8-year waiting period before ALGOL 68 received recommendations (HANSEN & BOOM [8]). We hope that no other programming language will ever be published without certain standards because it is at this level that debutants are the most sensible.

Our experience leads us to believe that ALGOL 68 is a good choice of an introductory programming language. We can even say that the discovered

errors are classic. Also, our experience as teachers has made us sure that the readability and good construction of the programs obtained is due to some of ALGOL 68's qualities, such as orthogonality, balanced constructions, identity declarations which allow for the clarification of the idea of variable, and a unique mechanism for the passage of parameters.

ACKNOWLEDGEMENTS

We would like to thank B. Boussais, J.P. Routeau, F. Ployette and L. Trilling for their help and our students for their involuntary contribution. Jean-Pierre Banâtre made a number of very helpful suggestions on earlier drafts of this paper.

REFERENCES

- [1] ANDRÉ, J. & J. BARRE, *Analyse des fautes commises par des étudiants débutant l'informatique par Algol 68*, P.I. IRISA no. 104, Rennes, 1978.
- [2] BOIES, S.J. & J.D. GOULD, *Syntactic errors in computer programming*, *Human Factors* 16(3), p. 253-257.
- [3] BUFFET, J., P. ARNAL, & A. QUÉRÉ, (ed.), *Définition du langage algorithmique Algol 68*, Act. sc. et Ind. no. 1338, Herman, Paris 1972.
- [4] FURATA, R. & P.R. KEMP, *Experimental evaluation of programming languages features: implications for introductory programming languages*, *SIGCSE Bulletin* vol. 11, no. 15, Feb. 79, p. 18-21.
- [5] GANNON, J.D., *An experiment for the evaluation of language features*, *Int. J. of Man Machines Studies* 8 (1976), p. 61-73.
- [6] GANNON, J.D. & J.J. HORNING, *The impact of language design on the production of reliable software*, *IEEE Trans. on Soft. Eng.* vol. SE-1, no. 2 (June 75), p. 179-191.
- [7] GERBIER, A., *Mes premières constructions de programmes*, Springer Verlag Lectures Notes no. 55, 1977.
- [8] HANSEN, W. & H. BOOM, *The report on the standard hardware representation for Algol 68*, *ALGOL Bulletin* no. 40 (August 76), p. 24-43.

- [9] HOC, J.M., *Role of mental representation in learning a programming language*, Int. J. Man Machines Studies 9, (1977) p. 87-105.
- [10] HOLT, R.C., *Teaching the fatal disease (or) Introducing computer programming using PL/I*, SIGPLAN Notices vol. 8 no. 5 (May 1973), p. 8-23.
- [11] ICHBIAH, J.D., *Reference manual for the Ada programming language*, DOD, July 1980.
- [12] KNUTH, D.E., *Structured programming with goto statements*, ACM Computing Surveys 6 (1974), p. 261-301.
- [13] KOVATS, T.A., *Program readability, closing Keywords and Prefix-style Intermediate Keywords*, SIGPLAN Notices vol. 13, no. 11 (Nov. 78), p. 30-42.
- [14] LEITNER, H.H. & H.R. LEWIS, *Why Johnny can't program a progress report?*, SIGCSE Bulletin vol. 10, no. 1 (Feb. 78), p. 266-276.
- [15] LITECKY, G.R. & G.B. DAVIS, *A study of errors, error-proneness and error-diagnosis in COBOL*, CACM vol. 10, no. 1 (Jan. 76), p. 33-37.
- [16] MIDDLETON, M., *The importance of syntactic trivia, Letter to the editor*, SIGPLAN Notices vol. 13, no. 3 (March 78), p. 17-19.
- [17] MUSA, J.D., *An exploratory experiment with "foreign" debugging of programs*, Proc. of symposium on computer soft. eng., (New York April 20-27, 1976), p. 499-511.
- [18] NICOLAS, M., M.J. PÉDRONO, J. BARRÉ & L. TRILLING, *A first programming course based on Algol 68*, Proc. of the conf. on applications of Algol 68, Univ. of East Anglia, Norwich 1976.
- [19] PAGAN, F.G., *Letter to the editor*, SIGPLAN Notices vol. 12, no. 4 (April 77), p. 3-4.
- [20] PAGAN, F.G., *Nested sublanguages of Algol 68 for teaching purposes*, SIGPLAN Notices vol. 15, no. 7-8 (July-Aug. 80), p. 72-81.
- [21] SCHOOMAN, M.L. & M.I. BOLSKY, *Types, distribution and tests and correction times for programming errors*, Proc. of 1975 Int. Conf. on reliable software, Los Angeles (April 21-23, 1975), p. 351.

- [22] SCOWEN, R.S. & B.A. WHICHMANN, *The definition of comments in programming languages*, Software Practice and Exp. vol. 4 (1974), p. 181-188.
- [23] VAN WIJNGAARDEN, A., B. MAILLOUX, J. PECK, C. KOSTER, M. SINTZOFF, C. LINDSEY, L. MEERTENS & R. FISHER (eds.), *Revised report on the algorithmic language Algol 68*, SIGPLAN Notices vol. 12, no. 5 (May 77), p. 1-70.
- [24] YOUNGS, E.A., *Human errors in programming*, Int. J. of Man Machines Studies 6 (1974), p. 361-376.

A COMPARATIVE EVALUATION OF ALGOL 68 FOR PROGRAMMING INSTRUCTION

P.R. EGGERT & R.C. UZGALIS

ABSTRACT

An experiment was performed on a large, intensive programming course to decide whether Algol 68 or PL/I should be used as a primary programming language. The main criterion used in the selection was the number of compiletime and runtime errors encountered during program development. Excluding the first two programs, in which language learning took place, Algol 68 programs had significantly more trivial syntax errors, although the total number of compiletime errors was roughly the same. PL/I programs had significantly more runtime errors, particularly subscript errors.

1. INTRODUCTION

One of the most commonly claimed benefits of strongly typed languages is that more errors are caught at compiletime; confirming experimental results with artificial languages and small sample sizes have been reported in GANNON [7]. In addition we observed that several errors are common when students move from a traditional syntax (in our case, PL/I) to the more unorthodox syntax of Algol 68; similar results were reported in GANNON [6]. We conducted an experiment testing the first claim on a larger sample with general-purpose programming languages; in the process we gathered data that sheds light on difficulties in learning the syntax of Algol 68.

2. BACKGROUND

At UCLA, the second course in computing, Computer Science 20 (or CS20 for short), has traditionally served as an intensive introduction to programming and problem-solving. Incoming students typically have had ten weeks of simple programming experience, and have written programs of at most 100 lines; outgoing students are expected to be mature programmers and often obtain jobs immediately in local industry. Approximately 40% of CS20's incoming students do not finish the course. Survivors are highly motivated.

One quarter, incoming students were partitioned into two sections, with no ability to transfer between sections. One section was taught using PL/I, the other using Algol 68. Incoming subjects arrived with no knowledge of Algol 68, and a limited knowledge of PL/I derived from exercises on the PL/C compiler [2], which gives excellent diagnostics. The first two weeks of the ten-week course covered language related material: advanced PL/I topics in the PL/I section, and most of Algol 68 in the Algol 68 section. The IBM PL/I optimizing compiler [10] was used for PL/I instruction; PL/C was considered unsuitable because it lacks pointers, and IBM's PL/I checkout compiler was considered too expensive. The Cambridge Algol68C compiler [1] was modified to improve its runtime diagnostics to quality approximately that of the PL/I optimizer's; the resulting compiler is called Calgol 68 [3]. At the time of the experiment, a major failing of the PL/I optimizing compiler was that exceeding resource limits caused some or all of the printout to be lost; a major failing of Calgol 68 was its many unimplemented features including multidimensional arrays, unions and formats.

Programming problems were run using a locally developed fast-turnaround batch system using upper-case-only keypunches. The set of attempts by a subject to solve a given problem will be called a "program" (although "program development process" might be a more accurate phrase). Each attempt to compile and execute was called a "run"; subjects were given a limited number of runs, 60, in order to finish the programming problems. Subjects manually kept summary records of the reason why each run failed; because it was hinted that the grade depended partly on the accuracy of these records, responses were of high quality. Part of each response was the system completion code yielded by the IBM OS/360 MVT operating system; this code served to clarify ambiguous entries.

Ten problems were assigned with varying point values. Subjects chose the problems, and could accumulate points up to a maximum of 700. Point values were calculated on the basis of difficulty observed when the problems were assigned previously, approximately one point per line of code for a good solution. Problems were as follows:

aa (150): Use a finite-state machine to follow specified rules for word abbreviation.

bn (100): Calculate π by simulation of dropping a needle on a floor with uniform parallel lines, using the method of Buffon.

er (100): Generate prime numbers using the sieve of Eratosthenes.

f (250): Convert arithmetic expressions involving parentheses and precedence into machine code.

j (200): Solve the general Instant InsanityTM puzzle.

kt (100): Calculate, by simulation, probability of a randomly touring knight falling on a given square.

lp (300): Three-dimensional graphics on a line printer.

pm (150): String pattern-matching using a tree algorithm.

s (150): Text formatting.

wm (150): Print a map of the world using several projections.

Subjects chose problems independently of the programming language they used. This can be seen from Table 0.

Table 0. Problem types and counts											
	aa	bn	er	f	j	kt	lp	pm	s	wm	total
PL/I	27	27	4	7	11	5	23	3	6	17	130
A 68	23	27	3	7	9	5	15	2	2	15	108

3. MEASUREMENTS

The summary error records produced by the subjects were scanned and reported errors were classified. Subjects typically reported only the errors that caused the run to fail; the two implementations are both unforgiving of errors, and so most runs caused only one reported error. If more than one error was reported in a run, all errors were counted equally. The programming problems, hereafter called "programs", required a number of runs for completion. Overall statistics are given in Table 1.

Table 1. Overall statistics			
	PL/I	Algol 68	Pooled s.d.
number of subjects	38	37	
number of programs	130	108	
mean programs/subject	3.4	2.9	1.3
mean runs/program	10.7	11.0	4.5
number of subjects receiving grades	35	37	
mean grade point average (0-4 scale)	2.7	2.5	1.4

None of the differences are significant at the 0.05 level of significance using a t-test. The two sections thus were fairly matched in work done and quality of performance measured by grades.

Table 2. Categories of runtime errors

Voluntary	Compiletime
no error	some syntax
logic error	Trivial syntax
changed index variable	, not OR
bad input data	a missing *
	op used as proc
	illegal identifier
	used EXIT
	Semicolon
External	some semicolon
compiler error	missing semicolon
unimplemented feature	extra semicolon
system crash	Comma
job control language	some comma
	missing comma
	extra comma
	unclosed comment
	String syntax
	missing quotes
	string across lines
	misused ' in string
Runtime	stropping
some runtime error	Syntax structure
Resource limit	some bracketing
time limit	if-syntax
storage limit	missing brackets
too much output	do-syntax
Language runtime error	format lists
Runtime pointer misuse	Declaration syntax
runtime REF	misspelled word
dereferencing NIL	array declaration
scope violation	missing declaration
zerodivide	redeclared tag
write on standin	forward declaration
Subscript and stringsize	bad INIT syntax
subscript error	for id used outside
stringsize	Types
Overflow	some type
overflow (hardware)	used =, not :=
size (software)	used =, not :=:
runtime conversion	"referencing"
uninitialized variables	Compiletime conversion
missing case	some compiletime conv.
read past EOF	int := real
	/, not %
	int MOD real
	print(VOID)
	compiletime REF

4. ERROR CATEGORIES

Each error described by a subject was classified as shown in Table 2. This classification scheme is derived from EGGERT & UZGALIS [4]. A short phrase at the start of each classification will be used in later tables. Some of the classes are composed of others: the classes thus have a tree structure, indicated by indenting in Table 2. Errors incompletely described by the subjects were put in category names beginning with "some".

The four major categories are Voluntary errors, discovered by the programmer and not by the system; External errors, which arise from circumstances outside the programmer's control; Runtime errors, which include both Resource errors (violating system limitations) and Language runtime errors, (violating language rules); and Compiletime errors.

The major null hypothesis tested by this data is that there is no difference in the two languages between the average number of errors experienced by a subject during the development of a program. To test this, the mean observed errors of each type per program were calculated. In Table 3 and subsequent tables, "err" refers to the number of observed errors, "mean" and "s.d." to the mean and standard deviation of the observed errors per program for PL/I and Algol 68. The column headed " α " gives the significance level for a t-test of the difference between the means. Errors that are not listed have $\alpha > 0.10$. Errors are listed starting with categories in which Algol 68 programs contained more errors, and ending with categories in which PL/I programs contained more errors; a horizontal line separates the two kinds of categories.

As can be seen, the major null hypothesis is decisively rejected in six major categories. Trivial syntax errors, Compiletime conversion, and Compiletime errors were far more common in Algol 68 programs; Subscript and Stringsized errors, Resource limit errors, and Runtime errors were far more common in PL/I programs. Part of the reason for the extra PL/I Resource limit errors and, by extension, the extra PL/I Runtime errors, may have been the deficient PL/I runtime support discussed earlier. The extra Subscript and Stringsized errors in PL/I may have arisen because character strings were not fully implemented in Algol 68; however, the Algol 68 programmers were forced to implement strings in terms of arrays, so similar errors should have resurfaced.

Table 3. Error category tabulation, all programs							
Categories with no difference at the .10 significance level are omitted.							
PL/I (N=130)			Algol 68 (N=102)			α	error category
err	mean	s.d.	err	mean	s.d.		
33	.25	.60	188	1.84	2.08	<.01	Trivial syntax
15	.12	.42	121	1.19	1.73	<.01	Semicolon
6	.05	.24	91	.89	1.54	<.01	some semicolon
445	3.42	2.51	547	5.36	3.10	<.01	Compiletime
0	.00	.00	16	.16	.42	<.01	stopping
0	.00	.00	10	.10	.30	<.01	used =, not :=
2	.02	.12	14	.14	.40	<.01	Compiletime conversion
0	.00	.00	9	.09	.32	<.01	int := real
0	.00	.00	9	.09	.32	<.01	unimplemented feature
10	.08	.34	31	.30	.79	<.01	unclosed comment
6	.05	.24	20	.20	.58	<.01	missing semicolon
12	.09	.34	23	.23	.58	<.05	some type
1	.00	.09	11	.11	.56	<.05	missing comma
2	.02	.12	12	.12	.57	<.05	Comma
0	.00	.00	3	.03	.17	<.05	, not OR
0	.00	.00	3	.03	.17	<.05	string across lines
0	.00	.00	3	.03	.17	<.05	"referencing"
9	.07	.28	17	.17	.47	<.10	if-syntax
8	.06	.35	22	.22	.93	<.10	runtime REF
3	.02	.15	10	.10	.48	<.10	extra semicolon
167	1.28	1.57	91	.89	1.60	<.10	Language runtime error
11	.08	.45	0	.00	.00	<.10	uninitialized variables
129	.99	1.54	57	.56	1.01	<.05	time limit
91	.70	1.16	35	.34	.85	<.01	Subscript & stringsize
24	.18	.67	0	.00	.00	<.01	stringsize
19	.15	.52	0	.00	.00	<.01	bad INIT syntax
21	.16	.53	0	.00	.00	<.01	runtime conversion
25	.19	.59	0	.00	.00	<.01	some compiletime conv.
44	.34	.98	1	.00	.10	<.01	too much output
186	1.43	1.94	68	.67	1.08	<.01	Resource limit
379	2.92	2.76	176	1.73	1.89	<.01	Runtime
31	.24	.59	0	.00	.00	<.01	format lists

Finally, the Trivial syntax errors may have been due to the process of learning the language. This possibility is related to the secondary null hypothesis tested by the data, namely, that there are no differences between the two languages once the first two programs turned in by each subject are excluded. When Table 3 is modified to exclude such programs the result is Table 4.

Categories with no difference at the .10 significance level are omitted							
PL/I (N=60)			Algol 68 (N=38)			α	error category
err	mean	s.d.	err	mean	s.d.		
14	.23	.65	37	.97	1.30	<.01	Trivial syntax
0	.00	.00	6	.16	.44	<.01	int := real
0	.00	.00	4	.11	.31	<.05	used =, not :=
0	.00	.00	4	.11	.31	<.05	unimplemented feature
4	.07	.25	10	.26	.60	<.05	some type
0	.00	.00	4	.11	.39	<.05	stropping
15	.25	.77	22	.58	.86	<.10	Types
2	.03	.18	6	.16	.44	<.10	Compiletime conversion
3	.05	.29	8	.21	.53	<.10	unclosed comment
0	.00	.00	8	.21	.87	<.10	missing comma
4	.07	.31	8	.21	.47	<.10	some semicolon
0	.00	.00	2	.05	.23	<.10	scope violation
0	.00	.00	2	.05	.23	<.10	Runtime pointer misuse
8	.13	.50	14	.37	.79	<.10	Semicolon
2	.03	.18	7	.18	.61	<.10	read past EOF
214	3.57	2.84	173	4.55	2.39	<.10	Compiletime
1	.02	.13	8	.21	.87	<.10	Comma
6	.10	.35	0	.00	.00	<.10	format lists
60	1.00	1.16	20	.53	.95	<.05	subscript error
8	.13	.39	0	.00	.00	<.05	runtime conversion
56	.93	1.49	13	.34	.67	<.05	time limit
11	.18	.47	0	.00	.00	<.05	stringsize
33	.55	1.33	0	.00	.00	<.05	too much output
71	1.18	1.26	20	.53	.95	<.01	Subscript & stringsize
93	1.55	2.20	19	.50	.83	<.01	Resource limit
213	3.55	3.26	72	1.89	1.47	<.01	Runtime

The secondary null hypothesis is rejected in four major categories. Trivial syntax errors were more common in Algol 68; Subscript and stringsize errors, Resource limits and Runtime errors were more common in PL/I.

A puzzling observation in the later programs is that subscript errors were significantly more common in PL/I. This may have arisen from lack of multidimensional arrays in Calgol 68; to the familiarity with refs that Algol 68 programmers must learn in order to program, encouraging use of pointers rather than subscripts; or to the slightly better runtime diagnostics of Calgol 68 vs the PL/I optimizer in this area.

Finally, a direct comparison between early errors, in the first two programs for each Algol 68 subject, and late errors, in subsequent Algol 68 programs, yields Table 5.

Table 5. Error category tabulation, early vs late Algol 68 programs							
Categories with no difference at the .10 significance level are omitted.							
Early (N=64)			Late (N=38)			α	error category
err	mean	s.d.	err	mean	s.d.		
250	3.91	1.67	188	4.95	2.48	<.05	Voluntary
0	.00	.00	2	.05	.23	<.10	do-syntax
146	2.28	.97	105	2.76	1.70	<.10	no error
94	1.47	1.51	81	2.13	2.30	<.10	logic error
3	.05	.21	6	.16	.44	<.10	int := real
15	.23	.77	20	.53	.95	<.10	subscript error
15	.23	.77	20	.53	.95	<.10	Subscript & stringsize
7	.11	.40	0	.00	.00	<.10	String syntax
44	.69	1.15	13	.34	.67	<.10	time limit
15	.23	.56	2	.05	.23	<.10	if-syntax
12	.19	.50	1	.03	.16	<.10	misspelled word
374	5.84	3.39	173	4.55	2.39	<.05	Compiletime
8	.13	.33	0	.00	.00	<.05	job control language
151	2.36	2.28	37	.97	1.30	<.01	Trivial syntax
83	1.30	1.80	8	.21	.47	<.01	some semicolon
107	1.67	1.94	14	.37	.79	<.01	Semicolon

Voluntary errors went up, probably because subjects attempted harder problems as time went on. The more subscript errors and fewer time limit errors may be due in part to the many subjects who picked problem 'bn' early; it needed no arrays but required many iterations to converge. The do-syntax errors were primarily a failure to get the control pieces of an Algol 68 do in the right order. The most significant differences occurred in the Trivial syntax errors, particularly semicolon errors, as would be expected according to the hypothesis of learning.

5. CONCLUSIONS

In searching for the differences between the languages, it is easy to overlook the surprising similarities. Despite the disparity between the two languages, and despite the greater familiarity with PL/I, after two programs, such important categories as Voluntary errors, logic errors, and Syntax structure showed no significant difference in later programs. When teaching the course, our feeling was that the CS20 students struggled with the problems far more than with the languages; this seems to be borne out by the data.

From general observation of the class during consulting, there were some nagging compiletime errors in both languages that deserve attention when either language is taught or when new languages are designed. The PL/I format lists and INITialization syntax are too complicated, and the Algol 68 stopping, semicolon rules, and unclosed comments caused especially annoying reruns. Part of the stopping problem was due to the upper-case-only environment, but that environment is still the rule in many places. Semicolons and comments are both the source of unnecessary learning errors in Algol 68.

However, in increasingly common interactive environments, compiletime errors are unimportant; they represent solved problems. Runtime and Voluntary errors are the source of most debugging expense; here Algol 68 was markedly better than PL/I, despite the handicap of being a new language for the subjects. Recent research in programming language design has concentrated on reducing incidence of Runtime and Voluntary errors [5,8,9,11-17]; future studies should reveal the effect of these designs on software errors.

REFERENCES

- [1] BOURNE, S.R., A.D. BIRRELL & I. WALKER, *ALGOL68C Reference Manual*, Cambridge University (1974).
- [2] CONWAY, R. & T.R. WILCOX, *Design and implementation of a diagnostic compiler for PL/I*, CACM 16, 3 (March 1973), 169-179.
- [3] EGGERT, P.R., M.G. KEARNS, A.S. TANENBAUM & R.C. UZGALIS, *UCLA Calgol 68 Programmer's Guide*, UCLA Computer Science Department (September 1978).
- [4] EGGERT, P.R. & R.C. UZGALIS, *Taxonomies of software errors and error detection methods*, UCLA Computer Science Department Quarterly 7, 2 (April 1979), 116-133.
- [5] EGGERT, P.R., *Detecting software errors before execution*, PhD dissertation, UCLA Computer Science Department (September 1980).
- [6] GANNON, J.D. & J.J. HORNING, *Language design for programming reliability*, IEEE Trans Software Eng SE-1, 2 (June 1975), 179-191.
- [7] GANNON, J.D., *An experimental evaluation of data type conventions*, CACM 20, 8 (August 1977), 584-595.

- [8] GOGUEN, J., J. TARDO, N. WILLIAMSON & M. ZAMFIR, *A practical method for testing algebraic specifications*, UCLA Computer Science Department quarterly 7, 1 (January 1979), 57-80.
- [9] GUTTAG, J.V., *Notes on data type abstraction* (version 2). IEEE Trans Software Eng SE-6, 1 (January 1980), 13-23.
- [10] IBM Corp, *OS PL/I checkout and optimizing compilers: language reference manual*, IBM publication SC33-0009-1 (September 1971).
- [11] ICHBIAH, J.D., J.G.P. BARNES, J.-C. HELIARD, B. KRIEG-BRUECKNER, O. ROUBINE & B.A. WICHMANN, *Preliminary Ada reference manual*, SIGPLAN Notices 14, 6 (June 1979), Part A, entire issue.
- [12] ICHBIAH, J.D., J.G.P. BARNES, J.-C. HELIARD, B. KRIEG-BRUECKNER, O. ROUBINE & B.A. WICHMANN, *Rationale for the design of the Ada programming language*, SIGPLAN Notices 14, 6 (June 1979), part B, entire issue.
- [13] LAMPSON, B.W., J.J. HORNING, R.L. LONDON, J.G. MITCHELL & G.J. POPEK, *Report on the programming language Euclid*, SIGPLAN Notices 12, 2 (February 1977), 0, i-ii, 1-79.
- [14] LISKOV, B.H., A. SNYDER, R.R. ATKINSON & C. SCHAFFERT, *Abstraction mechanisms in CLU*, CACM 20, 8 (August 1977), 564-576.
- [15] POPEK, G.J., J.J. HORNING, B.W. LAMPSON, J.G. MITCHELL & R.L. LONDON, *Notes on the design of Euclid*, SIGPLAN Notices 12, 3 (March 1977), 11-18.
- [16] WIRTH, N., *Modula: a language for modular programming*, Software - Practice & Experience 7, 1 (January-February 1977), 3-35.
- [17] WULF, W., R.L. LONDON & M. SHAW, *Abstraction and verification in Alphard: introduction to language and methodology*, ISI Technical Report ISI/RR-76-46 (June 1976).

TEACHING WITH ALGOL 68 IN DRESDEN

G. STILLER

ABSTRACT

ALGOL 68 has been used in lectures on problem-oriented programming for approximately 10 years as a prototype of a high level language in the education of "engineers for information processing". After a brief description of the aims of the corresponding branch of study its realization is outlined followed by a discussion of instructional and methodological aspects regarded to be significant for this subject in general and for the application of ALGOL 68 in particular.

1. GENERAL REMARKS

After ten year practice in teaching problem-oriented programming, mainly on the basis of ALGOL 68, a brief summary of impressions and experiences seems to be justified. These are predominantly positive. The author and his team, however, are aware that the subject under discussion is, to a noticeable extent, subjective and even self-confirming. There are, of course, other opinions and educational strategies with a different view of some basic principles, with other advantages and disadvantages and, finally, an also optimistic estimation of the success. An exchange of thoughts is, consequently, sufficiently motivated.

The training of "engineers for information processing" (comparable with "software engineers") at the Technical University Dresden lasts 4.5 years (9 semesters) including practical work in industry: all the 7th semester (the so-called "Ingenieurpraktikum") is spent in industry, and likewise, there are 4 weeks prior to the beginning of the studies (the so-called "Berufspraktikum"). Programming disciplines as an important part of the entire training program amount to $\approx 22\%$ of the total time available [18]. The main part of this time is spent on the topics stated in table 1 (additionally, but not mentioned here, there are lectures on compiler construction, operating systems programming methodology etc.).

Table 1: Survey of Lectures in Programming

Subject	Semester	Σ	Amount of Time (hours ¹⁾)		
			Lec- tures	Exer- cises	Prac- tice
Fundamentals of Programming	1	64	32	32	-
Machine Oriented Progr.	2,3,4,5	272	96	96	80
Problem Oriented Prog.	3,4,5	240	88	88	64

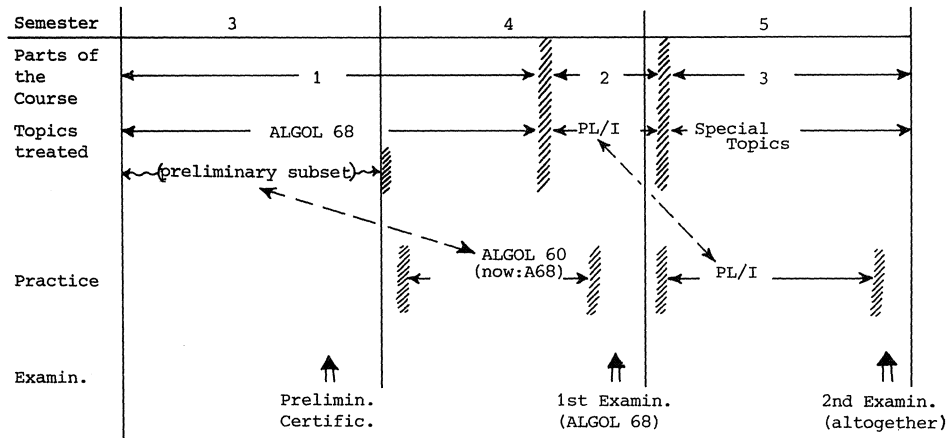
1) 2 hours = 85 min; exercises supplement lectures (repetition, explanation etc.), practice means practical programming (writing, punching, compiling, checking and elaborating programs)

Training in (problem oriented) programming is to be based upon a concrete language ("model language") that will fulfill well known conditions like clearness, understandability, typicalness of the state of the art etc. These properties are often not possessed by languages developed in the scope of industrial computer application. On the other hand, languages provided in education are frequently not widely used in industrial practice. Therefore, we feel obliged to accept a compromise: to take one (training) language in order to create or influence the student's view of programming and to use (more briefly) another one being applied in industry in order to also endow students with the appropriate knowledge. This may be considered unsatisfactory but, finally, it supports our pronounced intention of achieving a fair degree of versatility for our graduates. This leads to the following aims as a guide for education [17,8]:

- (a) students must be able to write programs (self-evident)
- (b) students must obtain a sufficiently universal knowledge of typical concepts in problem oriented programming (reflected by the model language)
- (c) students should be able to grasp yet unknown (to them) or new programming languages (or principles) based upon the view and knowledge gained.

From these aims criteria for the choice of a suitable model language have been derived. (a) is obviously supported by language features like clearness, understandability, simplicity (combined with a certain degree of orthogonality). (b) is favoured by generality, universality, whereas (c) requires properties somewhat like a "metalanguage behaviour" in the sense that the effect (semantics) of constructs in other languages may fairly well be expressed in terms of the model language. Our choice was in favour of ALGOL 68 because in our opinion this language satisfies (b) and (c) and (with certain restrictions) also (a). This resolution was taken comparatively early (1971). So we have been able to train - using ALGOL 68 - each annual course of students from the establishment of our department. It should be mentioned here that there was a close co-operation with I.O. Kerner who also performed the lectures of the first course (1971/72) as a host professor. The use of ALGOL 68 was, however, restricted to lectures and exercises only. Not until last year did we have an ALGOL 68 compiler available (our own compiler [12] is just now used for the first time for practical programming i.e. for "practice", viz. table 1). This was not too disadvantageous since ALGOL 60 may be regarded as a (somewhat modified) subset of ALGOL 68.

Table 2: Schedule of the Problem Oriented Programming Course



In this way practical programming has been performed on the basis of ALGOL 60. Tab.2 outlines a schedule of lectures, exercises, practice and examinations for the whole course. Examinations consist of a written exercise (60 min.) and an immediate individual discussion of the results (30 min.) with the examiner. The scheme outlined in tab.2 determines only a certain structure filled with the denoted contents, permitting actualizations of the topics treated, if necessary. It indicates that PL/I serves as the language for industrial applications, treated after the model language. The students are supported by some lectures. They have, however, to learn PL/I mainly by private studies using books. In this way qualities contained in (b) and (c) are simultaneously acquired, applied and checked. The third part (≈ 14 hours lectures) deals with special topics, such as list processing (ALGOL 68 and PL/I), parallel processing (ALGOL 68 with a glance at CONCURRENT PASCAL) and certain methodological questions.

2. SOME CHARACTERISTICS OF ALGOL 68 AND THEIR INFLUENCE ON TEACHING

ALGOL 68 differs from other comparable languages by strict orthogonality, a high degree of generalization as well as the consideration of implementation oriented principles (machine independent and compatible with a problem-oriented view). We regard this combination of an abstract and

realistic view as constructive but it has, on the other hand, also been an obvious reason for objections to ALGOL 68 [9].

The specific expressive power of ALGOL 68 has even stimulated attempts to use the language as a metalanguage for denotational semantics [13]. One can comparatively well explain the meaning of constructs in other languages by a suitable ALGOL 68 text. This is not to be understood as an attempt to formalize the semantics (according to [13]) but concerns somehow this idea. It is (only) intended to illustrate essential properties of language constructs by means of a similar text in the model language, often performed for a special example (for instance: declarations, parameter transfer mechanisms, data structuring, component access). In this sense ALGOL 68 approaches a "unified model" for high level languages. For that reason we like to speak of "teaching with ALGOL 68", not only teaching the language itself.

Orthogonality strongly supports the correct use of a language since straightforward rules may be mastered comparatively well and securely. So it supports appropriate coding of problems and helps to avoid primitive errors due to inadequate use of the language. This is regarded to be a contribution to programming security that should not be underestimated. It provides, of course, complete familiarity with the orthogonal rules.

For noninitiated students, however, orthogonality combined with generalizations acts as an initial barrier to understanding due to pronounced interaction (nesting, pile up) of certain orthogonal constructs: serial clauses contain units and declarations; declarations contain declarers and units; units contain declarers and/or serial clauses; declarers may contain units; units may be statements or expressions ... etc.

It is, consequently, necessary to separate such interactions in the beginning by preliminary simplifications of some concepts, starting with an informal prelude, in order to achieve a gradual, incremental understanding. This has very well been demonstrated in [11]. There are, indeed, several ways of realizing this. After having overcome the initial barrier all the further understanding proceeds progressively: behind the first mountains a nearly plain area of active programming is accessed. Such a preparation for easy understanding is closely related to the question how a suitable (set of) sublanguage(s) can be derived which fit the given requirements. This is obviously easy with regard to the orthogonal principles (omission of unions and flexibility, for instance). Other subsets of various power can also be derived, possibly with a certain loss of

orthogonality. Even the notion "reference", often regarded as the most important element of ALGOL 68, can be eliminated (after omission of all means of handling references as values "ref" remains only in formal parameter declarers and may here be understood as a special attribute indicating the in/out parameter passing mechanism "call by reference").

The power of ALGOL 68 has been satisfactory during all the work in the past. The extension mechanisms (mode and operation declaration, library prelude) have proved their worth as comparatively simple but powerful means to express specialized language features. Nevertheless, for each language being applied certain (possible or desirable) extensions or modifications will sooner or later be discussed. In the case of ALGOL 68 it was comparatively late that we felt some need to consider extensions, mainly from an experimental point of view. It seems to be possible to introduce, for instance, such PASCAL descendent [5] facilities as value ranges (even dynamic) and enumeration types into ALGOL 68 in a rather concise and clear manner [6]. Abstract data represent - from the ALGOL 68 point of view - a hybrid combination of properties of different constructs (structures and procedures, for instance) and may still have some particular features.

If an abstract data type (or "mode") is defined and variables of this type (mode) are declared and initialized later on this causes copies of the data type pattern to be written into the respective memory occupied by those variables. If the data type contains routines these are, of course, to be copied too. In this connection the "partial parametrization" proposed in [10] proves to be useful. Apart from a better understanding of the procedure call and the coercion "deproceduring", the yielding of a routine as the result of another routine (possibly by a partially parametrized call) gives a compact description of such copying. If such routines are produced for parallel elaboration this copying is essentially to be understood as a "physical copying" providing reentrant behaviour. Thus it is possible to describe semantic features of abstract data [3] and their respective manipulation to a certain extent and sufficiently elegant in terms of (extended) ALGOL 68 in a manner suitable for training.

3. HOW TEACHING OF ALGOL 68 HAS BEEN PREPARED

The author fully agrees with the opinion stated in [1], namely to achieve understanding for programming languages mainly from the semantical point of view. ALGOL 68 supports this too, because a noticeable part of

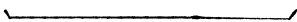
properties essential for the intended effect of programming belongs to so-called "internal objects" which is part of the semantics.

How do we provide students with useful literature? The strict formalization of the defining report [19] forbids its recommendation as a reference for learning ALGOL 68. Although there is a lot of additional informal literature one cannot really do without a certain framework of rules as a guide for active programming. This renders the application of ALGOL 68 undoubtedly more difficult or expensive but it is not a property of the language itself.

We have, therefore, prepared (and printed) a summary [16] of data (tables, rules etc.) as a supplement to lectures. It contains a set of syntactic rules written in an extended Backus-Naur representation supplied with some information about the context according to the 2VW grammar in a table-like representation. The following example (tab.3) denotes a simplified rule for the serial clause without labels and completers, where (D) indicates

Table 3: Simplified rule for the serial clause

$$\langle \text{serial clause} \rangle ::= [\langle \text{phrase} \rangle ;]^{0:\infty} \langle \text{unit} \rangle$$

(D)			
(M)	moid	(void)	moid
(P)	context	strong	context

a (possible) new layer for declarations, (M) and (P) denote the required mode and syntactic position, respectively. About 25 rules are already sufficient for practical programming. The rules are not intended to replace the defining report, they are not "exact", but they do their duty rather well. The rules are discussed within the lectures together with the corresponding semantics described in natural language and by drawing boxes as already proposed in [11] (a simple but very illustrative method).

Students trained at first in problem oriented programming are struggling against programming on machine oriented level (what's fully understandable but, finally, not permissible). Therefore nearly parallel performance of the machine and problem oriented courses has been provided. Nevertheless, students now obstinately try to write loops and alternatives by means of labels and jumps. Consequently, these means have been omitted (viz. the above mentioned rule) or, eventually, been postponed to the very

end.

The whole of ALGOL 68 is not treated in this way. Unions, longs and shorts are omitted (but can be included if desired). Flexibility appears only as a special property of string variables. Transput is restricted (read, print only). Parallelism is postponed till the third part, as mentioned before.

The informal introduction presents a still more restricted but nevertheless powerful sublanguage comparable with ALGOL 60, exceeding it, however, with respect to structures and procedures. The ALGOL 68 model of a variable including dereferencing as well as the model of a constant (denotation) is introduced very early. At the beginning only the variable declaration is used for the modes `int`, `real`, `bool`, `char`, `[..]`... and `struct(..)`. Slice and selection are defined as to permit read and write access via the subname to one component only. "Void" is introduced at the beginning as a metalinguistic notion only, in order to define a so-called "void block", "mode block", "void compound" and "mode compound" by means of simple rules as preliminary incarnations of the serial clause (to be explained later). Conditional and loop clauses contain only nested compounds at this stage, thus postponing the problems of block structure once more.

Procedures are instantly explained to be a representation of subprograms as data with well defined modes. They are correctly declared using the corresponding identity declaration, which is presented only for that purpose. Now, for the first time "void" appears as a possible (formal result) declarer which is intuitively well understood. Parameter transfer is explained anticipating the semantics of the (still unknown) identity declaration, and anticipating at this point also the idea of a nested block in the case of parametrized procedures. An intended, intuitive preunderstanding of the identity declaration is achieved in this way, leading to a somewhat surprising enlightenment when the declaration is later treated and recognized as an "already known" orthogonal element within the procedure mechanism. The introduction of the identity declaration is to be regarded as an essential step that should be performed carefully.

The informal part ends up with necessary generalizations and surveys (serial clause, block structure, unitary clause, coercions, references, generators, identity declaration, collaterality). These topics can already be treated more concisely, leading to a more formalized overall description with some repetitions.

4. DISCUSSION

Ten years ago ALGOL 60 was still taught within a total of 32 hours (lectures only). Now, with the same expense, the preliminary ALGOL 68 subset is treated. It exceeds ALGOL 60, and includes several provisions for generalizations. The complete ALGOL 68 part required ≈ 52 hours (lectures) within the last course. We intend to reduce this amount to ≈ 44 hours because more and improved printed material will be available this year for private studies. In our opinion the expressive power of ALGOL 68 exceeds that of ALGOL 60 significantly more than the corresponding time for training seems to indicate. We regard this to be a relative gain in time due to the orthogonality principle. Nevertheless, this amount is high: it is obviously the price to be paid today in favour of the appropriate horizon and according to the fact that the whole program development becomes more and more language-aided and even language-guided [7]. Restrictions, if necessary, may reduce this amount during education and postpone it to a later period of work.

What may be told about the results of education? There is, at first, a quite normal distribution of excellent, good, fair and bad results over the number of students. "Good" students are usually also successful in other disciplines, "bad" ones usually not. Successful students are not only using the language in a proper way but are made fit for original, creative programming (they have comprehended the whole philosophy).

The study of PL/I is also supported by some self-made (and printed) surveys including a skeleton of rules. A lot of program examples is presented and explained, usually in relation to corresponding ALGOL 68 solutions. In this context it is interesting to note that the students very soon perceive plain differences between both languages mentioned and sometimes they spontaneously utter critical comments. If the education is (partially) responsible for such an ability to properly estimate the characteristics of a language we are willing to regard this as a success.

The survey and versatility obtained seems to be fair, in any case better than ten years ago: to become familiar with a language of the FORTRAN, PASCAL or ALGOL 60 size is now one weekend's work. Graduates in industry have confirmed that they are able to follow continued professional training with less trouble compared to other collaborators with a more conventional level of education in this field.

A weaker point ought to be mentioned, however. Students don't get too

much support for programming on the level of operating systems that serve as PL/I environment. The situation in this field seems unsatisfactory in general. Much trouble is caused by the subject itself: compilers are often handled in the environment of operating systems which are completely machine oriented. Communication with the system then proceeds on a rather low language level. It is impossible to cast teaching into such a frame. On the other hand, the question is important. Users are more and more "programming within the operating system", i.e. they are activating system actions connected with file handling and transput, event handling, parallelism, scheduling etc.

Education has to take this into account but it can be done only on the basis of a problem oriented solution [2]. Transput and file handling are already solved in an adequate manner. There is, however, usually an overlap with lectures on data bases which are mainly leaning upon the COBOL and/or PL/I philosophy as a predominant orientation in industrial practice. For the next future we intend to leave this subject to the responsibility of the courses on data bases, with attention, however, to the further development. For this reason we shall retain transput facilities on a not too extended level, also when using our own compiler.

The teaching program outlined in this paper has been discussed seriously, and by general consent, among specialists, who also agreed with our decision in favour of ALGOL 68 as the training basis in problem-oriented programming. Nevertheless, in other institutes, universities or sections different "model languages" have been chosen, for instance PL/I, PASCAL, ALGOL 60, FORTRAN and even BASIC. The reasons for such decisions may be various: computational (availability of compilers); special directions in research closely related to the use of a specific language which is then, consequently, also applied in training; more restricted amount of time for lectures; recommendations of users etc. Our feeling is that these decisions are, surely, well motivated, as is our own decision: the use of high level languages is, obviously, in itself a matter of problem orientation. Only two arguments shall be briefly commented upon in this context.

The first argument states that attention should mainly be paid to the writing of well-structured programs and this is possible in any language. This opinion obviously reduces the contributions of high-level languages to the art of programming to certain convenient structuring facilities only (alternative, loop, subprogram, perhaps some formatting of the program text). These facilities exist, indeed, in nearly any language (even

assembler) but cover only a small part of what is really to be considered [20,9].

The other opinion prefers effective communication (dialogue) to the debit of the level of programming (BASIC for instance). We must differ from this opinion since we regard the dialogue to be another quality that cannot replace the essential characteristics of high-level languages.

As mentioned before, ALGOL 68 includes certain implementation principles condensed in language constructs, rules or notions, which are in this way made visible at the problem oriented level (in a rather generalized manner, however).

We know about objections to these specific characteristics. We have to agree that they somewhat complicate the initial understanding (the already mentioned "barrier") and that they, to some extent, are dispensable at a lower grade of education, as frequently required for non-specialists. Although general reasons for the choice of ALGOL 68 have already been discussed a few comments should be added to this question.

The principle of utmost simplicity (as an objection also to ALGOL 68) has not been realized in certain recently designed languages (for instance PEARL, ADA, CHILL [15,4,14]). Obviously the actually (also by non-specialists) required expressive power leads to voluminous languages exceeding the intuitive boundary of extreme simplicity. All the more it seems to be necessary to keep languages transparent by stricter orthogonality. ALGOL 68 is obviously still unique in this respect.

If a certain degree of simplicity or universality is required which is not met by the model language selected we prefer to take a subset of a more powerful language rather than to extend a too simple one (the possible derivation of ALGOL 68 sublanguages has already been discussed). This is in favour of homogeneity and orthogonality.

Computers have to be adapted to the human mind, not vice versa. This is a clear vote in favour of problem orientation as a long-term aim. To which extent this may actually be realized depends on the whole "context" of programming. This necessarily leads to some compromise.

When an inexperienced programmer has learnt $x[i]$ to be a (subscripted or component) variable that can be assigned (or can yield) a certain component value of an array, he has obtained the information for a correct use of this construct but this is not sufficient for skilled programming according to the requirements of practice. He has to learn additionally, either by his own experiences or by extra instructions, that this construct

behaves quite different from a "normal" variable with respect to its run-time behaviour and that it obviously is another thing. The ALGOL 68 notion of a slice is more realistic. A similar question is, whether parameter-passing mechanisms should be defined within the language or not (copying or not, etc.). Possibly this knowledge is required since it influences the portability of programs. The limited accuracy of numerical value representations turns out to even become an essential concept in language design.

It is no secret that information handled by present computers is contained in (referable) locations of a memory of limited size and that this handling must be programmable and executable economically, considering the peculiarities of the whole process. As long as certain peculiarities exist and significantly effect programming they have to be taken into account.

Obviously the question is to which extent some (more pragmatic) instructions (or definitions) are excluded from language descriptions in order to hide them from the user in favour of a purely problem oriented view. The hidden information is then, however, obtained from elsewhere, usually in a much more machine-oriented manner than is desirable, and this really is no advantage. The ALGOL 68 solution is problem oriented and realistic in the sense explained above, and it is machine independent. This obviously reflects an engineering point of view. We have used it with success.

REFERENCES

- [1] BAUER, F.L., G. GOOS, *Informatik I*, Berlin-W.-Heidelberg-New York, Springer-Verlag 1971, Heidelberger Taschenbücher 81.
- [2] ELZER, P., R. ROESSLER, *Real Time Languages and Operating Systems*, 5th IFAC/IFIP Int. Conf. on Digit. Computer Appl. to Process Control, van Nauta Lemke, A.-R. Verbruggen, H.B. (Editors), North Holland Publ. Co. (Preprints) 1977, Amsterdam.
- [3] HEINER, M., G. STILLER, *Synchronisation bei Parallelverarbeitung*, Technische Universität Dresden, Weiterbildungszentrum für Mathematische Kybernetik und Rechentechnik, Informationsverarbeitung, Heft 46/80 (1980).
- [4] ICHBIAH, J.D. et al., *Preliminary ADA Reference Manual*, ACM Sigplan Notices 14 (1979) No. 6.

- [5] JENSEN, K., N. WIRTH, *PASCAL - User Manual and Report*, Lecture Notes in Comp. Sci. 18 (1974).
- [6] KÖNIG, H., G. STILLER, *Einbeziehung von Wertebereichen und Aufzählwertarten in das mode-Konzept von ALGOL 68 und ihr Zusammenhang mit dem Begriff der Wertartäquivalenz*, Submitted to the conference "Algorithmische Sprachen ALGOL" Dresden, Pädagogische Hochschule "K.F.W. Wander II", June 1-5, 1981.
- [7] LEHMANN, N.J., *Sprachlich gestützte und geleitete EDV-Projektierung*, Technische Universität Dresden / Informationen Sektion Mathematik / WB MKR 07-28-79.
- [8] LEHMANN, N.J., G. STILLER, *Einige methodische Aspekte der Entwicklung und Nutzung höherer Programmiersprachen*, Rechentechnik und Datenverarbeitung 13, 2. Beiheft 1976, 7-10, Berlin, Verlag Die Wirtschaft.
- [9] LEHMANN, N.J., G. STILLER, *Unterstützung des Programmentwurfs und der Programmtestung - Folgerungen bezüglich geeigneter Spracharchitektur*, Wissenschaftliche Zeitschrift der Technischen Universität Dresden 27 (1978), 1135-1144 (Heft 6).
- [10] LINDSEY, C.H., *Specification of Partial Parametrization Proposal*, ALGOL Bulletin 39, pp. 6-9.
- [11] LINDSEY, C.H., S.G. VAN DER MEULEN, *Informal Introduction to ALGOL 68*, North Holland Publ. Co. 1971 and 1977, Amsterdam.
- [12] LOEPER, H., H.-J. JÄKEL, H. PIETSCH, *Semantic Analysis and Synthesis in the ALGOL 68 R 4000 Compiler*, (These proceedings).
- [13] PAGAN, F.G., *ALGOL 68 as a Metalanguage for Denotational Semantics*, The Computer Journal 22 (1979) 1, pp. 63-66.
- [14] Proposal for a Recommendation for a C.C.I.T.T. High Level Programming Language (2nd Edition), C.C.I.T.T. Study Group (1977).
- [15] Programmiersprache PEARL, Basic PEARL, Beuth-Verlag Berlin-Köln, DIN 66253, Teil 1, June 1978.
- [16] STILLER, G., *Skripte zur Lehrveranstaltung ALGOL 68*, Technische Universität Dresden, Sektion Informationsverarbeitung, als Manuskript gedruckt 1976, 1978, 1980.

- [17] STILLER, G. et al., *Lehrprogramm für das Lehrgebiet Problemorientierte Programmierungstechnik zur Ausbildung in der Grundstudienrichtung Informationsverarbeitung an Universitäten und Hochschulen der DDR*, Herausgeber: Ministerium für Hoch- und Fachschulwesen der DDR, Berlin 1978, als Manuskript gedruckt.
- [18] Studienplan für die Grundstudienrichtung Informationsverarbeitung zur Ausbildung an Universitäten und Hochschulen der DDR, Herausgeber: Ministerium für Hoch- und Fachschulwesen der DDR, Berlin 1976, Bestell-Nr. 338 341 9.
- [19] WIJNGAARDEN, A. VAN et al., *Revised Report on the Algorithmic Language ALGOL 68*, ACTA INFORMATICA Vol. 5 Fasc. 1-3 (1975), Berlin-W.-Heidelberg-New York, Springer-Verlag.
- [20] WIRTH, N., *On the Design of Programming Languages*, in: Rosenfeld, J.L. (ed.): Information Processing '74, 2 (1974) 386-393, Amsterdam, North-Holland Publ. Co. 1974, (Proceedings of the IFIP Conference 1974).

SEMANTIC ANALYSIS AND SYNTHESIS IN THE ALGOL 68 R 4000 COMPILER

H. LOEPER, H.-J. JÄKEL, H. PIETSCH

ABSTRACT

The paper gives a short survey of the implementation of an ALGOL 68 language version on the medium-size Robotron computer R 4000 at the Department of Information Processing of the Technical University Dresden. The aim of the implementation is to make an ALGOL 68 compiler available for teaching in the field of problem-oriented programming, in which ALGOL 68 is used as a prototype of a high-level language. Therefore, only few restrictions exist for declarations and units in comparison to the full ALGOL 68. The user may extend the set of defined standard objects (modes, procedures, operators) by using a special pragmat. The implemented ALGOL 68 version makes a modular program structure possible.

Furthermore, the global structure of the compiler is briefly described by a short explanation of the five compiler passes. Especially, the paper deals with the realization of semantic analysis and synthesis in the ALGOL 68 R 4000 compiler. In this compiler, semantic analysis and synthesis have been separated sharply from the other tasks of the compiler by well-defined interfaces:

- representation of the syntactic structure of the source program in an intermediate program,
- representation of the meaning of the program in a machine-independent target program,
- representation of semantic information in the symbol table.

The implemented semantic analysis and synthesis are based on the so-called O-attribute grammars derived from the general attribute grammars, which were developed by D. Knuth. Their application is described.

O-attribute grammars have only synthesized attributes and, in addition to these, O-attributes.

By using O-attribute grammars, the linear representation of the syntax tree of the source program may be translated sequentially into the target program. A simple and effective method for the realization of O-attribute grammars is presented, and explained by an example of an ALGOL 68 language construct.

1. A SHORT SURVEY OF THE IMPLEMENTED ALGOL 68 LANGUAGE VERSION

In order to ease an estimation of the effectiveness and applicability of the described semantic analysis and synthesis method, a short survey of the implemented ALGOL 68 language version is provided before the proper explanations of the semantic analysis and synthesis. In comparison to the full ALGOL 68 only few restrictions exist for declarations and units in ALGOL 68 R 4000. The ALGOL 68 version has generators for the local runtime stack and the global heap. The realized block concept allows declarations and statements in any sequence within the block (serial clause with declarations). Blocks and procedure calls may have results of any mode. Dynamic arrays (multiple values) and structures may be used in the declaration of new objects without restriction. But there are no flexible arrays. The data-related reference concept is completely realized.

In comparison to the full ALGOL 68, the following restrictions need be mentioned:

- There is no parallel processing and no semaphore technique.
- The union mode and the conformity case clause are not included in the language version.
- The completer (exit) is not allowed in the serial clause.
- Enclosed clauses are only permissible in strong positions, so that no balancing is necessary.
- The possibilities of transput, which is realized by special syntactic constructions, and the format-texts are restricted.

In ALGOL 68 R 4000, main programs and subroutines exist additionally as modules and may be translated separately. The realized module concept, which permits a higher clearness in programming and more effective validation by separate compilation and test of the modules, is very simple. A main program is always a labeled closed clause, e.g.

```
extern mp1: begin ... end .
```

Subroutines have the form of a labeled routinetext, e.g.

```
extern sp1: (real a, b)real: sqrt(a * a + b * b).
```

The subroutine is a module that can be activated in other modules by its external identifier (e.g., sp1). The so-called code declaration for procedures and operators, which is derived from the ALGOL 68 identity declaration, and describes the connections between the modules, has been

incorporated in ALGOL 68 R 4000. By means of the code declaration a subroutine is ascribed to the declared procedure identifier, or to the declared operator indicator, respectively. The above-given subroutine may be used in another module by the following code declarations:

```
proc (real, real) real diagonale = extern sp1;
op (real, real) real hypot = extern sp1.
```

Note: subroutines can also be programs written in other languages, and operators can be overloaded by several code declarations.

The ALGOL 68 R 4000 language and the compiler contain tools for extending the set of standard declarations (modes, procedures, operators) by the user. New standard modes, procedures, and overloaded operators of any priority can be inserted into the standard frame by means of mode declarations, code declarations and priority declarations.

Note: actual bounds in mode declarations for extending the standard frame must be denotations.

For that purpose, a special pragmat exists, which is a sequence of discussed declarations enclosed by pr-symbols. A pragmat can be separately compiled, but it can also appear before each ALGOL 68 R 4000 program:

```
<compilable unit> ::= <ALGOL 68 R 4000 program>|
                    <pragmat><ALGOL 68 R 4000 program>|
                    <pragmat>.
```

The compilation of the following pragmat makes the operator hypot with priority 8 available as a standard operator:

```
pr op (real, real) real hypot = extern sp1;
    prio hypot = 8
pr.
```

After the compilation of a pragmat, the extended symbol tables are organized in a file which can be read in the next translation process. Each module possesses one directly subordinated declaration level in relation to this, possibly extended, standard frame. By translating several pragmat, a repeated extension of the standard frame is possible. Using a file name of the extended symbol table part, special commands allow to work with one of the several special standard frames for classes of users.

2. THE STRUCTURE OF THE COMPILER

The pass division gives one of the most important pieces of information about the global structure of a compiler. The pass division depends on the source language, the level of the target language, the features of the computer (main store capacity, periphery) and special aims of the implementation (e.g., optimizing). Lexical and syntactic analysis, code generation, as well as symbol table organization, semantic analysis and synthesis are relatively separate tasks in a compiler. The clear delimitation of the tasks by an appropriate pass division, and by a modular structure, guarantees the reliability, portability, and adaptability of the compiler.

In the present ALGOL 68 compiler the lexical analysis (the recognition of the morphems and their conversion into internal code) is realized in a separate first pass. The first pass translates the source program into a program of the so-called syntax language, a sequence of coded morphems. Morphems are the smallest syntactic entities of the source program that carry meaning. The context-dependent conversion of some symbols happens in the second pass before the syntactic analysis, because the context-dependent conversion of certain basic symbols (e.g., indicators) into corresponding internal codes is only possible by means of unlimited right context information.

The syntactic analysis is based on a context-free grammar derived from the ALGOL 68 definition, and is realized in the second compiler pass by a precedence-controlled method with bounded context examination. The second pass delivers an intermediate program which is the right linear representation of the syntactic program structure (syntax tree).

The semantic analysis is divided in two passes. At first, in the third pass, the mode information is built up in a mode graph by investigating the syntactic structure of the declarers. After the equivalence investigation of the modes, the proper mode checkings (e.g., check of coercions, operator identification) in the program to be translated are executed in the fourth pass. The meaning of the program (dynamic semantics) is provided in a machine-independent target language improving the portability of the compiler. The fifth pass generates the machine-dependent assembly language.

A survey of the most important tasks of the five passes is given below.

- 1. pass: - morphem recognition and context-dependent lexical analysis and conversion
 - construction of the symbol table
 - check of the block structure
- 2. pass: - context-dependent conversion of some morphems
 - determination of the syntax tree
 - construction of the symbol table
- 3. pass: - construction of the mode graph, check of mode equivalence and well-formedness
 - completion of the syntax tree by mode and right context information
- 4. pass: - semantic analysis and synthesis
 - operator identification
 - check of coercions
 - generation of the machine-independent target language
- 5. pass: - generation of the assembly program (macro expansion)
 - machine-dependent compile-time checks

The source program with error messages and warnings detected in all passes is listed after the expansion of the macro-like target program. Subsequently the program is assembled into the object program. The size of each of the five passes is at most 20 kByte, so that more than 20 kByte can be used by the symbol table organization in connection with a virtual storage management system.

The discussed implementation tries to form a machine-independent compiler realization by the following premises:

- The use of the system-programming language CDL for all five compiler passes.
- An unambiguous identification of the interfaces to the computer and its operating system, and the collection of machine-dependent program sections in special CDL modules.
- The separation of the machine-independent compiler parts from the machine-dependent code generation by using a machine-independent target language. The so-called macro-processor, which translates the target

program into the assembly language, is also written in CDL.

Therefore, the first four passes can be transferred to any other computer without large modifications; a lot of modules of the fifth pass can be re-used. Considering the problems of portability, it must be kept in mind that the ALGOL 68 implementation has a large storage run-time system and an I/O-System, which are naturally machine-dependent. Extensive conceptual and programming toil lies in these components of the ALGOL 68 programming system.

3. SEMANTIC ANALYSIS AND SYNTHESIS

3.1. Preliminary remarks

Semantic analysis deals with the compile-time control of such determinations of the language definition that cannot be proved in lexical and syntactic analysis (static semantics). The semantic synthesis represents the meaning of the source program machine-independently and uses information of the semantic analysis (dynamic semantics). Therefore semantic analysis and semantic synthesis can be considered as connected tasks of the compiler, often called evaluation. For the evaluation, three interfaces are important: syntactic analysis, symbol table organization, and code generation:

- 1) The syntactic structure of the source program is the input information of the semantic analysis and synthesis. Semantic analysis and synthesis can be delimited to the syntactic analysis in two ways:
 - The evaluation is immediately executed for each syntactically analyzed source program construct, e.g., a sequence of calls of semantic routines mediates the syntactic structure.
 - The syntactic structure of the source program is completely represented in an intermediate program, which is processed in the semantic analysis and synthesis.

The second variant, which is used in the present ALGOL 68 implementation, offers the advantage that the evaluation can be formed independently of special lexical and syntactic analysis methods, especially those for error recovery and correction.

- 2) It is efficient to use the symbol table during the analysis of certain context-dependencies, e.g., during the identification process.

A suitable symbol table organization can influence the evaluation decisively. However, these problems will not be dealt with here in detail.

- 3) The result of the evaluation is a target program, which represents the meaning of the source program.

The semantic analysis and synthesis of the present ALGOL 68 compiler are realized on the basis of a formal description by the so-called O-attribute grammars. From now on we will deal with the representation of the syntactic structure of source programs, with the application of the O-attribute grammar in the translating process, and with the machine-independent target language used in the compiler of the implemented ALGOL 68 language version.

3.2. The representation of the syntactic structure of the source program

Since the syntactic analysis is based on a context-free grammar, the syntactic structure of a program is a syntax tree. The syntax tree is a finite directed graph with labelled nodes and arcs.

- D 1: The quintuple (K, Z, k_0, f, g) is a syntax tree in relation to the context-free grammar (V, A, R, s) , if
- (K, Z) is a tree with the root k_0 ;
 - K is the set of nodes which can be subdivided into the disjoint sets of the terminal nodes K_t and of the nonterminal nodes K_n :

$$K = K_n \cup K_t \text{ and } K_n \cap K_t = \emptyset;$$
 - f is the node labelling function, which labels each node $k \in K$ with a pair (v, r) , in which v is a vocabulary symbol or ϵ and r is a rule, $r \in R$ or $r = \epsilon$;
 - $f: K \rightarrow (V \cup \{\epsilon\}) \times (R \cup \{\epsilon\})$, especially
 $f: K_t \rightarrow (A \cup \{\epsilon\}) \times \{\epsilon\}$,
 $f: K_n \rightarrow (V - A) \times R$ and
 $f(k_0) = (s, r)$ with $r = (s, w) \in R$ and $w \in V^*$;
 - Z is the set of arcs with $Z \subseteq K_n \times K$, where

$$\forall k [k \in K_n \rightarrow \exists k' [k' \in K \wedge (k, k') \in Z \wedge f(k) = (z, r) \wedge r = (z, v_1 v_2 \dots v_n) \in R \wedge \text{card}(Z_k^{+1}) = n]].$$

- g is the labelling function that attaches a natural number to each arc in order to arrange the set of direct descendant nodes Z_k^{+1} of each node $k \in K_n$:

$$g: Z \rightarrow N$$

$$\forall k [k \in K_n \wedge f(k) = (z, r) \wedge r = (z, v_1 \dots v_i \dots v_n) \in R \rightarrow$$

$$\forall i [1 \leq i \leq n \rightarrow \exists ! k' [k' \in K \wedge (k, k') \in Z \wedge g((k, k')) = i \wedge$$

$$f(k') = (v_i, r') \wedge r' \in R \cup \{\epsilon\}]]].$$

The syntax tree can be represented by lists or linear bracket representations in the intermediate program. In the ALGOL 68 implementation discussed here, the right-linear representation of the syntax tree is used. This representation is well-suited both for tree construction by syntactic analysis as well as for processing the syntax tree by semantic analysis and synthesis, because only a sequential file organization is needed, and, what is more, the storage size is relatively small compared to other representation methods. The advantage of the right-linear representation is that a syntactic construct is only then identified when all its constituents are represented. This principle is profitably applied in the semantic synthesis, because a target language operation corresponding to a syntactic construct can not be generated before the constituents of the syntactic construct are determined.

- D 2: The right-linear representation of a syntax tree $b_0 = (K, Z, k_0, f, g)$ is the right-linear representation of the syntax subtree b_0 , at which the right-linear representation of a syntax subtree $b' = (K', Z', k'_0, f', g')$ is recursively defined by

$$rp(b') = \begin{cases} a & \text{if } f'(k'_0) = (a, \epsilon), a \in A \cup \{\epsilon\} \\ (rp(b_1), rp(b_2), \dots, rp(b_n))r & \text{if } f'(k'_0) = (v, r), \\ & r = (v, v_1 v_2 \dots v_n) \in R \end{cases}$$

and $\forall i [1 \leq i \leq n \rightarrow b_i = (K_i, Z_i, k_i, f_i, g_i)$ is a syntax subtree of b_0 with the root $k_i \in K' \wedge (k'_0, k_i) \in Z' \wedge g((k'_0, k_i)) = i]$.

- D 3: The quintuple (K', Z', k'_0, f', g') is a syntax subtree of the syntax tree (K, Z, k_0, f, g) in relation to the context-free grammar

Nevertheless, a linear representation can only be used if sequential processing of the syntactic structure is possible in the evaluation. This condition is not given a priori. But the implementation of the extended version of ALGOL 68 has shown that right-linear representations of syntax trees can be translated in a sequential process into macro-like machine-independent target programs by means of the so-called O-attribute grammars.

3.3. Aspects of the machine-independent target language

The target language generated by the evaluation is the interface between the machine-independent and machine-dependent part of the compiler. Such languages are often described in the literature. They have a different language level. The target language used in the ALGOL 68 R 4000 compiler is a simple macro-like language. The machine-independent target program is a sequence of macro-statements with the following general structure of macro-operators:

$$m_i = mop_i (opd_{i_1}, opd_{i_2}, \dots, opd_{i_n})$$

mop_i macro-identifier

opd_j macro-operand which parameterizes the macro-operator.

Macro-operands cannot be macro-statements, so that the target program has a simple structure. The design of the macro-operators takes into consideration the following criteria:

1) Generality of macro-operators

The operators are borrowed from the elementary constructs of high-level programming languages and are defined machine-independently.

2) Simplicity of macro-control-operators

The control structures (alternatives, loops, case clauses) of the source language are realized by elementary tools of the target language. There are macro-control-operators only for unconditional and conditional jumps, for labelling macro-statements, and for realizing subroutine calls.

3) Efficient level of decomposition

Syntactically interlocked constructs of the source language are decomposed by elementary macro-statements of the target language. All implicit actions of the source program are explicitly represented in the target

program by macro-statements. Note: block begins, block ends, and procedure calls are not decomposed.

4) Symbolic representation of the macro-operands

The operands of the macro-operators are symbolically represented and can be classified into the following types of operands derived from the source language objects: denotations, block-dependent objects, formal objects, actual objects, routine texts, format texts, logical accumulator, stack.

5) Using mode information

Only elementary source constructs which are defined for an infinite set of modes are directly represented by macro-operators (e.g., assignment statement, and subroutine calls). These macro-operators are parameterized by the modes of the operands of the elementary source construct.

6) Using the symbol table

Operand information which is stored in the symbol table and used in the target language expansion (e.g., mode representations, and denotations) is represented by pointers in the macro-statement. Thus the size of the target program is reduced and the evaluation is simplified.

By using such a macro-like language, the lexical and syntactic analysis, and the semantic analysis and synthesis are realized in the compiler machine-independently for a wide range. 55 macro-operators are defined in the target language for the implementation of the ALGOL 68 language version on the R 4000. The following example is to demonstrate the level and some features of the globally discussed target language.

Example:

Let *i* be an integral variable. The assignment $i := i + 1$ is represented by the following macro-operators:

```
CALL BEGIN      (BNR, proc(int, int) int, (EX, N+))
VALUE COPY     ((AO, BNR, 2), (KS, 1), int)
DEREFERENCING ((B0, BNri, BPOSi), ref int, int)
VALUE COPY     ((AO, BNR, 1), (AC), int)
CALL           (BNR, proc(int, int) int, (EX, N+))
PARAMETER     ((AC), int)
PARAMETER     ((KS, 1), int)
```

```
CALL END      (BNR, proc (int, int) int, (EX, N+))
MOVE         ((BO, BNRi, BPOSi), (AC), int)
```

It is important to know that:

- The modes are represented by pointers to the symbol table.
- BNR is an integral number attached unambiguously to the call.
- (EX, N+) is an external procedure realizing the int-Addition.
- (AO, BNR, 1) and (AO, BNR, 2) are the actual parameters and actual objects of the call, respectively.
- (BO, BRNi, BPOSi) is the representation of the operand i (block object), in which BRNi denotes the scope of i and BPOSi the position of i within this scope.
- The results of dereferencing and of the call are stored in the logical accumulator (AC).

The target language used in the ALGOL 68 R 4000 compiler has a relatively high level. For instance, the assignation of objects, which can have any mode, is represented by only one macro-operation of the target language. This, secures on the one hand a complete machine-independence of the target language, and on the other hand that the efficient implementation of the target language is not hindered by a too extensive elementarization. Naturally, the cost of translating this language into any assembly language is not small.

3.4. O-attribute grammars

In the implementation of evaluations by general attribute grammars as developed by Knuth, the attribute storing and the general representation of the syntactic structure of the source program as well as the algorithms for the calculation of all attributes are expensive. A central problem in the use of these grammars is the suitable restriction between characteristics of attribute grammars and the scanning method of the syntactic structure in order to reduce implementation costs. The O-attribute grammars derived from the general attribute grammars are the result of investigation in this direction. In the ALGOL 68 R 4000 compiler the translation of the syntactic structure of ALGOL 68 programs into the machine-independent target programs is completely described formally by an O-attribute grammar.

Compared with the general attribute grammar, the following problems can be easily solved by restricting to synthesized attributes only:

- 1) The proof of cyclic dependencies of the attributes is trivial because there are only synthesized and no inherited attributes.
- 2) For calculating the attribute values, the syntax tree is to be scanned from the leaves to the root only once, since the synthesized attributes of a node are only dependent on the attributes of its direct descendants.
- 3) From this it follows that the syntax tree can be processed sequentially. Therefore the use of the simple right-linear representation of the syntactic program structure is profitable.
- 4) The storage of the attributes can be managed in a stack-oriented manner, because the attributes of a node are only needed for determining the attributes of its direct ancestor.

By restricting to synthesized attributes only, a representation of context-dependencies becomes impossible. The evaluation of a language construct can be completed only after all context-dependencies are known, i.e., in a higher construct. Such solutions are difficult to attain and not even efficient. O-attribute grammars, however, allow to determine the attributes of a node not only by the attributes of its direct descendants, but also by the attributes of its direct-left context. A direct-left context is sufficiently illustrated by figure 2 that shows part of a syntax tree. In this figure all nodes k , which can be reached on the path from k_{i+1} to k labelled with one (k_{i+1} is included), possess the same direct-left context k_i .

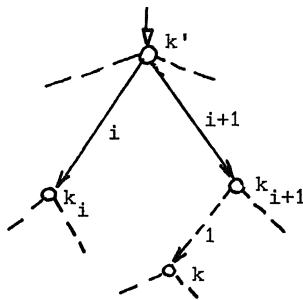


Figure 2: Part of a syntax tree for the representation of the direct-left context

Note: all attributes of the node k_i are determined before calculating the attributes of node k resp. k_{i+1} , if the syntax tree is scanned from bottom to top and from left to right. It is easy to prove that each node k of a syntax tree has at most one node constituting the direct-left context of the node k . Furthermore, the nodes k , which are on the path from the root to the node k labelled with one, have no direct-left context. Besides the attributes of the direct descendants of k , the attributes of the node k_i can also be used as so-called O-attributes in order to calculate the attributes of the node k . Attribute grammars modified in this manner are called O-attribute grammars.

D.4: The quintuple (AT, V_a, A_a, R_a, S) is an O-attribute grammar of the context-free grammar (V, A, R, S) , if the following is true:

1. AT is a finite set of attributes.
2. $V_a \subseteq V \times P(AT) \times P(AT)$ is the attributed vocabulary, in which exactly one triplet (v, S_v, A_v) exists in V_a for each $v \in V$. S_v and A_v^O denote the synthesized and O-attributes, respectively, of the symbol v . S_v and A_v^O are subsets of AT . The synthesized attributes enable information transmission in root direction. The O-attributes allow the information transmission from left to right, i.e., depending on the left context. The start symbol S of the grammar possesses no O-attributes.

Note: $P(AT) = \{X: X \subseteq AT\}$.

Let $w(a, i)$ with $0 \leq i \leq n$ describe the value of a synthesized attribute a of the symbol v_i appearing in the i -th position of the rule $(v_0, v_1 \dots v_i \dots v_n)$ of the grammar. $w(a, 0)$ denotes the value of a synthesized attribute a of a terminal and $w^O(a)$ represents the value of the O-attribute a .

3. A_a is the attributed alphabet. Each terminal $t \in A$ possesses a set F of semantic functions, that determine the values of the synthesized attributes of t . The values of other synthesized attributes and of O-attributes of t can be arguments of these functions:

$$F = \{f_s : s \in S_t \wedge w(s, 0) = f_s(w_1, \dots, w_j, \dots, w_m, w_1^O, \dots, w_i^O, \dots, w_\ell^O) \wedge \\ w_j = w(a, 0) \wedge a \in S_t \wedge 1 \leq j \leq m \wedge \\ w_i^O = w^O(a') \wedge a' \in A_t^O \wedge 1 \leq i \leq \ell\}.$$

4. R_a is the attributed set of rules. Each rule $(v_0, v_1 v_2 \dots v_n) \in R$ has a set F of semantic functions that determine the set of synthesized attributes of v_0 . The synthesized attributes of all symbols v_i with $0 \leq i \leq n$ of the rule and the O-attributes of v_0 can be arguments of these functions:

$$F = \{f_s : s \in S_{v_0} \wedge w(s, 0) = f_s(w_1, \dots, w_j, \dots, w_m, w_1^O, \dots, w_i^O, \dots, w_\ell^O) \wedge$$

$$w_j = w(a, k) \wedge a \in S_{v_k} \wedge 1 \leq j \leq m \wedge 0 \leq k \leq m \wedge$$

$$w_i^O = w^O(a') \wedge a' \in A_{v_0}^O \wedge 1 \leq i \leq \ell\}.$$

Contrary to Knuth's attribute grammars, the test of cycles in the O-attribute grammars is trivial. Cyclic dependencies of attributes on several nodes cannot arise, because the information transmission is possible only from bottom to top and from left to right in the syntax tree. Further conditions must be fulfilled in O-attribute grammars:

1. Each node with no direct-left context can only be labelled by a vocabulary symbol that does not possess O-attributes.
2. The O-attributes of vocabulary symbols must be assigned to the direct-left context as synthesized attributes.

For a given O-attribute grammar these conditions can always be proved in the following steps:

- The sets L_v and LK_v are determined for each vocabulary symbol $v \in V$.

$$L_v = \{v' : v' \in V \wedge v \xrightarrow[G]{*} v'w \wedge w \in V^*\}$$

$$LK_v = \{v' : v' \in V \wedge \exists r [r = (z, w_1 v' w_2) \in R \wedge w_1, w_2 \in V^* \wedge$$

$$z \in V - A \wedge v \in L_{v''}] \}$$

- Condition 1 is exactly fulfilled if the set L_S , where S denotes the start symbol, contains only symbols to which no O-attributes are attached

$$\forall v [v \in L_S \rightarrow A_v^O = \emptyset]$$

- Condition 2 is exactly fulfilled if for each symbol v , which possesses O-attributes, the set LK_v contains only symbols which have these attributes as synthesized attributes

$$\forall v [v \in V \wedge \forall v' [v' \in LK_v \rightarrow A_v^o \subseteq S_{v'}]].$$

3.5. Determination of the attribute values

The evaluation by O-attribute grammars consists of the determination of attribute values of all syntax tree nodes. The important advantage of the O-attribute grammars lies in the possibility to determine the values of all attributes by sequential reading of the right-linear representation of the syntax tree. The evaluation by O-attribute grammars uses a stack for attribute storing which is an efficient storage management scheme, because attribute values no longer needed are implicitly released. The method of calculating the attributes can be described as follows:

The right-linear representation of the syntax tree is sequentially processed from left to right. If a syntax subtree with root k is completely worked off, then

1. the attribute values of the direct-descendant of node k are popped off the stack;
2. the attribute values of the direct-left context of node k are read from the stack top;
3. the attribute values of node k are calculated by semantic functions of the rule $r = (v_0, v_1 v_2 \dots v_n) \in R$ for $f(k) = (v_0, r)$ and by the semantic functions of the terminal $t \in A$ for $f(k) = (t, \epsilon)$;
4. the attribute values of node k are pushed onto the stack.

Because O-attribute grammars permit a direct dependence of attributes of a node from the attributes of its direct descendant and its direct-left context, the use of information of the complete left context is indirectly possible. The complete left context and all descendants of a node k are exactly all the nodes which have already been processed. Therefore, the evaluation by O-attribute grammars is a suitable adaptation to the scanning method of the right-linear representation of the syntax tree.

3.6. The O-attribute grammar of the assignment statement - an example

The following example of the assignment statement vividly demonstrates the use of O-attribute grammars. The evaluation is based on the syntac rules

```
< UNIT > ::= = < DESTINATION > < UNIT >
< DESTINATION > ::= < TERTIARY > becomes token.
```

The symbols possess the following attributes:

symbol	synthesized attributes	O-attributes
UNIT	value ac-saved	mode-of-ac-outside scope destination-mode
DESTINATION	mode-of-ac-outside scope destination-mode mode value ac-saved	mode-of-ac-outside scope
TERTIARY	value ac-saved form mode	mode-of-ac-outside scope

The following semantic functions are attached to the syntactic rules. Attributes with an integral number $i > 0$ denote the attributes of the symbol v_i of the rule $(v_0, v_1 v_2 \dots v_n)$; attributes with the number 0 denote the O-attributes of the symbol v_0 and attributes without any number denote the synthesized attributes of the symbol v_0 .

< DESTINATION > ::= < TERTIARY > becomes token

```

scope := scope 0
mode := meek(mode 1, reference)
value := if mode = mode 1 then value 1 else ac fi
destination-mode := deref(mode)
ac-saved := if mode = mode 1 then ac-saved 1
                else mode-of-ac-outside 0 ≠ void
                fi
mode-of-ac-outside := if value = ac then mode
                else if ac-saved 1
                    then void
                    else mode-of-ac-outside 0
                fi
                fi
if mode ≠ mode 1
    then if ¬ ac-saved 1
        then if mode-of-ac-outside 0 ≠ void
            then GENERATE(SAVE AC(mode-of-ac-outside 0))
            fi
        fi;
        generate meek coercion(mode, mode 1, value 1)
    fi
if mode = error-mode
    then if mode 1 ≠ error-mode then error("tertiary is no reference") fi
fi
< UNIT > ::= < DESTINATION > < UNIT >
value := if destination-mode 0 = mode 1 then value 1 else ac fi
ac-saved := if mode-of-ac-outside 0 = void
                then false
                else ac-saved 1 ∨ ac-saved 2 ∨ destination-mode 0 ≠ mode 1
                fi
if value 1 = ac
    then if ac-saved 2
        then GENERATE(MOVE(stacktop, value 2, destination-mode 1));
            GENERATE(RESTORE AC)
        else GENERATE(MOVE(value 1, value 2, destination-mode 1))
    fi

```

```

        fi
      else GENERATE(MOVE(value 1, value 2, destination-mode 1))
fi
if destination-mode 0 ≠ mode 1
  then if mode-of-ac-outside 0 ≠ void
    then if ¬ ac-saved 1 ^ ¬ ac-saved 2
      then GENERATE(SAVE AC(mode-of-ac-outside 0))
      fi
    fi;
    generate strong coercion(destination-mode 0, mode 1, value 1,
                             comorf)
  fi
if ¬ strong coercion possible(destination-mode 0, mode 1)
  then if mode 1 ≠ error-mode
    then error("no coercion to destination mode")
  fi
fi

```

A detailed explanation of the example would exceed the scope of this paper. In this context only the note should be made that calls of generating routines and error routines are applied instead of attributes for the generation of the target program and for the message of semantic errors.

4. FINAL REMARKS

In [7], KASTENS only uses the direct ancestors and descendants of a node in the syntax tree for the attribute evaluation. O-attribute grammars are derived from S-attribute grammars [7]. S-attribute grammars have only synthesized attributes. For a suitable representation of context dependencies the so-called O-attributes are inserted additionally. With these O-attributes it is possible to represent dependencies of attributes of a node in relation to the attributes of the direct-left context. With this the restriction of attribute dependencies to the scope of one syntactic rule is given up. Therefore, the O-attribute grammars can not be arranged in the hierarchical classification of attribute grammars as given by KASTENS. O-attribute grammars have an a priori predefined attribute evaluation strategy. The synthesized- and O-attributes may be

evaluated in a bottom-up pass if the walk through the syntax tree is considered. It is not possible to cover right-context dependencies with O-attribute grammars. Such instances are rare in practice, so that their handling can for instance be ensured by a suitable symbol-table organization. It must be emphasized, however, that the description of the semantic analysis and synthesis by O-attribute grammars enables an effective, clear and low-error programming of the corresponding compiler part.

REFERENCES

- [1] BOCHMANN, G., *Semantic Evaluation from Left to Right*, CACM 19, 2 (1976).
- [2] BOCHMANN, G & P. WARD, *Compiler Writing Systems for Attribute Grammars*, The Computer Journal 21, 2 (1977).
- [3] JÄKEL, H.-J., *Erzeugung einer rechnerunabhängigen Zielsprache aus der rechtslinearen Darstellung syntaktischer Bäume*, Dissertation, TU Dresden, 1980.
- [4] JÄKEL, H.-J. & H. LOEPER, *Struktur und Darstellung der Zwischenprogramme des ALGOL 68-R 4000-Compilers*, TU Dresden, WBZ MKR/IV 1980, H. 46.
- [5] JÄKEL, H.-J., H. LOEPER & W. OTTER, *Ein Algorithmus zur Behandlung der Modusäquivalenz in einer Untersprache von ALGOL 68*, EIK 14 (1978), H.4.
- [6] KASTENS, U., *Ein Übersetzer-erzeugendes System auf der Basis attributierter Grammatiken*, Dissertation, Universität Karlsruhe, 1976.
- [7] KASTENS, U., *Ordered Attribute Grammars*, Acta Informatica 10, 3 (1980).
- [8] KNUTH, D., *Semantics of Context-free Languages*, Mathematical Systems Theory 2, 2 (1968) and 5, 1 (1971).
- [9] KNUTH, D., *Examples of Formal Semantics*, Lecture Notes in Mathematics 188, Springer-Verlag, Berlin, Heidelberg, New York, 1971.
- [10] KOSTER, C., *Using the CDL Compiler-Compiler*, in: *Compiler Construction*, Lecture Notes in Computer Science 21, Springer-Verlag, Berlin, Heidelberg, New York 1974.

- [11] LOEPER, H. & P. BACHMANN, *Theorie und Technik der Übersetzerprogramme höherer Programmiersprachen*, BSB B.G. Teubner Verlagsgesellschaft, Leipzig, 1980.
- [12] LOEPER, H., M. HORN & W. NYDERLE, *Investigations on the Application of a Precedence-Controlled Syntactic Analysis Method to a Sublanguage of ALGOL 68*, EIK 11 (1975), H. 4-6.
- [13] LOEPER, H., H.-J. JÄKEL & W. OTTER, *Anwendung der Automatentheorie bei der Behandlung der Modusäquivalenz in ALGOL 68*, EIT 7 (1977), H. 5.
- [14] WIJNGAARDEN, A. VAN, et.al., *Revised Report on the Algorithmic Language ALGOL 68*, Acta Informatica 5, 1-3 (1975).

ESSAY ON COPYING

K. WRIGHT

ABSTRACT

When working with large data structures it is more efficient to copy a pointer to the structure than to copy the structure itself. Avoiding copies of large values can have a huge impact on the efficiency of some programs. The psychological impact of copying can exceed even its actual importance. I have found that almost without exception users react with utter horror to the suggestion that an array may be copied, even when this is obviously necessary for semantic consistency. Moreover the fear of copying warps the programming style of many people, leading them, for example, to create variable parameters which are not intended to vary. This paper explores the copying requirements of the definition of ALGOL 68 and techniques for reducing the amount of copying done.

1. PREVIOUS WORK

In spite of the importance of this subject for any implementation of ALGOL 68 there seems to be very little discussion of it in the publicly available literature.

BRANQUART et al. [3] have published a very complete description of the translation of every construct of ALGOL 68. This of course includes a description of when copying is done, but the discussion of this point is spread throughout the book. The scheme used there apparently works, but they do not explicitly describe how it was developed or what the alternatives are.

PETER SZOKE [4] discusses copying requirements and gives several pathological examples. Much of the discussion centers on programs in which assignation and dereferencing of the same name proceed collaterally. He wants to ensure that the result computed will be one of the set of possible results if copying were done as described by the (original) Report but with assignation and dereferencing treated as inseparable actions. In our view this is not mandated by the Revised Report, which refuses to specify what actions are inseparable.

P.G. HIBBARD et.al. [5] describe an implementation of ALGOL 68 that is designed to minimize the copying required for the most common operations. This implementation strategy is so unusual that the body of this paper does not consider it, but so important that it is discussed briefly in the appendix.

MARK RAIN [2] has suggested an alternative language with a different scheme of defining variables and pointers in an attempt to sidestep this issue. S.G. VAN DER MEULEN [6,7] has proposed a similar scheme as a strict extension to ALGOL 68. Both proposals involve the introduction of an unmodifiable reference to a value.

Neither proposal is formalized sufficiently to make the implications clear. In general the idea of a read only variable makes me uneasy. It seems to involve either run time checks of variability before each assignment, or the possibility of modifying a supposedly constant value through assignments to global variables or pointers. There may be a way around these difficulties, but I do not know what it is. I believe the ALGOL 68 definition of names is elegant and adequate from an abstract point of view. If one does not intend to assign to a name, then the name is nothing but an indirect method of accessing another value. The value could as well be

accessed directly. It is unfortunately difficult to specify just when copying is needed to support the abstraction. This difficulty alone may be sufficient reason to adopt a different scheme in a language meant to be a replacement for assembler.

There are two cases in which ALGOL 68 is too restrictive in its treatment of names. The first, as VAN DER MEULEN [6] points out, is the problem of flexible and transient names. It remains to be seen whether it is names or multiple values that need adjustment. The second is the impossibility of constructing recursive modes without inserting spurious ref's (or proc's). Perhaps we should allow declarations such as:

```
mode tree = struct(string val, union(tree, void) left, right);
void leaf = empty;
tree t = ("root", ("branch", leaf, leaf), leaf)
```

A complete exploration of these ideas would lead us far astray.

2. GENERAL PRINCIPLES

We define an "un-optimized" implementation as one in which each action prescribed by the Revised Report is translated in the same way regardless of the context in which it occurs. There is not a unique unoptimized implementation even when the particular features of storage layout are ignored. The appendix describes an alternative un-optimized implementation. The existence of such alternative implementations demonstrates that there is no hope of deducing the requirements for copying directly from the Revised Report. There are a few arbitrary decisions that must be made at the very outset.

In the body of this paper we limit consideration to a traditional implementation strategy. A name is represented by an address; some locations associated with that address contain a bit pattern representing the value to which it refers.

The primary action that requires a copy of a value is assignation. The bit pattern representing the value yielded by the source must be copied to the address representing the name yielded by the destination. Once the decision has been made that assignation will be implemented by holding the address which represents the name fixed and overwriting the contents of that address, it follows that some precaution will be needed to ensure that the value already stored there will not be lost if it is still needed.

For this reason a copy will also be required when a name is dereferenced.

Dereferencing is performed by copying the value referred to by the name onto the top of the stack (or some other safe place). The value must be copied since, in general, it is possible for the name to be altered while the value obtained is still in use. If a copy were not made altering the name would overwrite the value, causing it to change unexpectedly.

For example:

```
loc thing var:= initial value;
thing val = var; #var is dereferenced, its value is called 'val'#
var:= another value;
#Here we should have val = initial value. This requires that the
  initial value have been saved somewhere.#
```

Notice that the identity declaration itself does not require a copy of the value. Had we written:

```
thing val1 = some value;
thing val2 = val1; #Val1 is not a name, and is not dereferenced.#
```

no copy would have been required. Val1 in this case is not a name, and therefore can not be altered. Thus it does not matter whether a new copy of the value is made, or only one copy is kept simply remembering that both val1 and val2 access that same value. This is a major reason for preferring an identity declaration to an assignation where there is no intention to assign a different value. Similarly no copy is needed in the other cases when a value is ascribed to an identifier, i.e. when passing parameters or elaborating a conformity clause.

Given that the language definition does not require a copy to be done when an identity definition is elaborated, it is natural to try to arrange the storage layout in such a way that no copy is needed in order to do the ascription. This can, in fact, be arranged if there is an identifier stack in each stack frame set aside to hold the values ascribed to identifiers in the environ represented by that stack frame, and if there is a working stack set aside to hold intermediate results of the elaboration of constructs within that environ. All that is necessary is to place the working stack immediately following the identifier stack. Then after the elaboration of the source of the identity definition the yield is the only result on the working stack. The size of the identifier stack can then be increased

to include the newly created value, and the working stack moved up above the other end of the value.

In addition to copies made when assigning or dereferencing many implementations make copies of arguments to be passed to procedures or of results yielded. These copies are not in any way implied by the language definition, but are done only to allow procedures to communicate, to avoid holes in the stack, or for similar reasons. Since these copies are made solely for the convenience of the implementation it is always safe to eliminate them if some more convenient scheme is found. We are primarily interested in exploring the requirements of the language definition that are independent of the physical layout of storage, and so these copies will be, for the most part, ignored.

It may be possible to avoid making the copy of the value both during dereferencing and assignation. In the case of dereferencing this is done by remembering that the value which should have been yielded by the dereference action is actually still stored at the address of the name. In the case of assigning it is done by "moving the name", that is by remembering that the name now corresponds to the address at which the value is already located (thus changing the bit pattern used to represent the name). In either case this "remembering" may be done at compile time by updating the symbol table to reflect the actual positions of the objects in storage, or at run time by creating pointers to the objects in known locations. Which of these will be done depends upon the scheme used to locate objects in the run-time environment.

Moving a name means changing the address which represents the name. In addition the representation of all its subnames must be changed, since these are generally required to have some fixed relationship to the name. In addition we must ensure that the scope of the name is not changed.

When a copy which is called for by the un-optimized implementation is omitted, whether during assignation or dereferencing, the situation which results is the same. We have a name which refers to a value which is accessible independently of that name. We call such a name an alias. If we create an alias we must ensure the following rule.

Correct aliasing rule: An alias may not be altered.

There are two approaches to ensuring that this rule is enforced. We can ensure before creating any alias that there will be no occasion to alter the aliased name while the value is still accessible, or we can

ensure before doing any assignation that the name to be assigned to is de-aliased.

De-aliasing implies either moving the name or copying the value to which it refers. In either case we must ensure that all identifiers which access that moved object will now access the copy, and all names which refer to the moved object refer to the copy. Moving the name will be quite difficult at this point since there has been plenty of time for further references to the name to have been created. In addition we need a run-time check when assigning to ensure that the name assigned to is not an alias. If the decision were made to try to de-alias at run time the result would be very much like the implementation described in the appendix. In this case it would probably be much simpler to adopt the model of computation described there. Therefore we will concentrate on finding conditions which will be sufficient to ensure at the time an alias is created that it will not be altered.

One approach to doing this would be to do complete global flow analysis of the program to determine what names might be altered or required by each construct. The techniques for doing this are fairly well known, but difficult to apply. We will try the alternate approach of using information which the compiler is likely to already have available. In particular we make heavy use of mode and scope information. Before specifying such conditions we define some new terms.

3. DEFINITIONS

A name N is "descended from" a name M if N is a subname of M or descended from a subname of M. Two names are "related" if they are the same name, or if there is a name from which they are both descended.

A name is "altered" if it is made to refer to a value (possibly other than the one to which it originally referred). This can be done by assigning to the name, or by assigning to some name related to the name. Two names "overlap" if assignment to one of them can alter the other. Overlapping names are necessarily related but the converse is not true.

Two modes overlap if there could be two overlapping names which have those respective modes. The precise conditions for this are best expressed grammatically.

- a) where REF to MODE1 overlaps REF to MODE2:
 where MODE1 stows MODE3 and MODE2 equivalent ROWS of MODE3;
 where MODE2 stows MODE3 and MODE1 equivalent ROWS of MODE3.
- b) where MODE4 stows MODE6:
 where MODE4 equivalent MODE6, where true;
 where MODE4 equivalent FLEXETY ROWS2 of MODE5,
 where MODE5 stows MODE6;
 where MODE4 equivalent structured with
 PROPSETY1 MODE5 PROPSETY2 mode,
 where MODE5 stows MODE6.

The predicate "where MODE4 stows MODE6" holds if MODE4 might be the mode of some stowed value which contains a value of MODE6. To determine if two modes overlap we must check if either mode is that of a name referring to a row of some mode stowed in the other mode. This is to account for the possibility that overlapping names might be produced by selecting from and subscripting a name referring to a row of structures, and also the possibility of rowing a name selected from a structure.

The "representation" of a value is a specific bit pattern that the implementation uses for that value. There may be more than one "instance" of the representation of a given value in storage at once. In addition to the representations which are explicitly stored there may be some representations which are stored only as (usually short) algorithms for their construction. For example a name is represented by an address, but that address may be stored only in the form of a register number and displacement within some instruction in the program. Integral values may be stored only in load immediate instructions.

The Revised Report says that names, procedures, and values composed from them are the only values whose scope is limited. Nevertheless particular instances of the representation of a value may be stored on the stack and thus may have a limited lifetime. To describe this situation we introduce the concept of the scope of an instance. The scope of an instance is an indication of how long the instance will be in use. The scope of an instance stored on the stack is no older than the scope of the environ in which the action which yielded the instance took place. If the value is required outside that scope the compiler will of course be forced to generate code to create a copy of the instance in some older stack frame. The exact scope of an instance will depend upon the compiler; all we require

is that the scope be defined in such a way that the compiler will never produce instructions which attempt to make use of an instance after its scope has expired.

The scope of an alias is the scope of the aliased name or the scope of the instance of the value referred to by the aliased name, whichever is newer.

We say that a construct "can alter" a name if the compiler can not determine that the elaboration of that construct will not alter the name. Obviously this relation depends upon the compiler as well as the construct and the name. A more complex compiler will have a better knowledge of what can actually occur.

We want to be able to determine whether a given name will be altered. This task is not hopeless because ALGOL 68 does not allow names to be computed at will. Only a few actions can yield a name not related to the parameters of the action. Even these few actions are constrained to yield names that bear a definite relation to their parameters.

We say that a value V "exposes" a name N if there is some sequence of actions which, given V, yield a name which overlaps N. We say a construct exposes name N if the construct yields a name which exposes N.

Since a procedure call may result (indirectly) in a name being altered it is important to be able to determine what procedure is yielded by some constructs. Fortunately the production of new procedures is even more constrained than the production of new names. The definition of "exposes" will therefore be extended to procedures. A value V "exposes" a procedure if there is some sequence of actions which, given V, yield the procedure.

We say that one mode exposes another if a value of the first mode could expose a value of the second mode.

where MODE exposes MODE2:

where MODE equivalent PLAIN, where false;
 where MODE equivalent REF to MODE1,
 where MODE1 exposes MODE2 or MODE overlaps MODE2;
 where MODE equivalent FLEXETY ROWS of MODE1,
 where MODE1 exposes MODE2;
 where MODE equivalent structured with FIELDS mode,
 where MODE1 field TAG resides in FIELDS and
 MODE1 exposes MODE2;

where MODE equivalent procedure PARAMETY yielding MODE1,
 where MODE1 exposes MODE2 or MODE equivalent MODE2;
 where MODE equivalent union of MOODS mode,
 where MOODS contains MODE1 and MODE1 exposes MODE2.

4. CONDITIONS FOR SAFETY

A proposed alias is unsafe if some construct elaborated within the scope of the alias and subsequent to its creation can alter the aliased name. It is not required to worry about constructs which can alter the name, but are elaborated collaterally with the creation of the alias. This is because the Revised Report does not specify what actions are inseparable and does not specify the intermediate states of the action of assigning. We interpret this to mean that the result of a dereference and an assignment to the same name, or of two collateral assignments to the same name, is totally undefined. For example, if on some computer it is expedient to set a storage location to zero by loading the accumulator from that location and then doing a subtract from memory operation it may happen that (k:= 0, k:= 0) has the effect of negating k. In fact if the compiler can reliably determine that a name actually will be altered by a collateral action it would be helpful to print a warning message to the effect that the collateral actions interfere with each other.

{As an aside - the result of (a[i]:= 1, a[j]:= 2) is clearly undefined if i = j. The Revised Report describes assignment to a subname as equivalent to assigning to the entire name a multiple value which differs in one element. Taken literally this implies that the above collateral assignment is undefined even if i ≠ j. Is this a reasonable interpretation?}

To be safe we must avoid creating an alias unless we determine that it can not be altered. For this reason the following conditions are stated negatively. Of course the conditions given could be strengthened if the compiler is made more complex. A simpler compiler could check only some weaker conditions. The conditions stated below should give the general idea.

The only action which can alter anything directly is an assignation. In addition a call, formula, or deprocedured form can alter a name if the called procedure contains an assignation. An actual stowed declarer may also alter a name if the computation of the bounds involves an assignation. Such a declarer can be viewed as a procedure without parameters. We will

use the word "invocation" to describe a call, formula, deprocedured form, or actual stowed declarer.

An assignation can not alter the name N if the destination does not overlap N.

An invocation can not alter a name N if all of the following hold

- 1) none of its actual parameters (operands) expose N
- 2) none of its actual parameters expose a procedure that globally alters N
- 3) the called procedure does not globally alter N

A construct can not yield a name which overlaps N if any if the following conditions hold

- 1) the mode of N does not overlap the mode of the construct
- 2) the scope of the yield of the construct is not the same as the scope of N
- 3) in case the construct is a
 - a) serial clause - the last unit can not overlap N
 - b) assignation - the destination can not overlap N
 - c) selection - the secondary can not overlap N
 - d) slice - the primary can not overlap N
 - e) invocation - N is newer than the procedure and none of the actual arguments can expose N
 - f) cast - the enclosed clause can not overlap N
 - g) generator - N is not derived from the same generator
 - h) applied identifier - the corresponding defining identifier occurred in an identity definition the source of which can not overlap V (variable definitions and routine definitions can be treated as identity definitions. Without global flow analysis we are stymied by a parameter definition)
 - i) dereferenced form - N has never been assigned to a pointer
 - j) rowed form - the coerced can not overlap N
 - k) all others - can not

A construct can not expose a name or procedure V if any of the following holds

- 1) the mode of the construct does not expose the mode of V
- 2) the scope of the yield of the construct is older than the scope of V

- 3) in case the construct is a
 - a) serial clause - the final unit can not expose V
 - b) invocation - the procedure can not expose V and none of the actual arguments can expose V
 - c) selection - secondary can not expose V
 - d) slice - primary can not expose V
 - e) routine text - its unit can not expose V
 - f) cast - the enclosed clause can not expose V
 - g) rowed form - coerced can not expose V
 - h) dereferenced form - coerced can not expose V
 - i) united form - coerced can not expose V
 - j) applied identifier - the corresponding defining identifier occurred in an identity definition the source of which can not expose V (the remarks made in the rules for overlapping also apply here)
 - k) all others - can not

Each of these sets of conditions translate directly into an algorithm which proceeds by recursively decomposing the construct. At each step either the mode or the scope of the construct may allow us to conclude that the given condition can not hold. Eventually the complexity of testing may become so great that we simply give up and make the safe assumption that the condition can hold.

A procedure can not globally alter a name N if one of the following is true:

- 1) the routine text of the procedure is known and it contains no assignments or calls which can alter N.
- 2) the scope of the procedure is older than the scope of N
- 3) the program contains no routine texts that both
 - a) contain a construct which can alter N, and
 - b) have the same mode as the procedure

5. EXAMPLES

We now give several examples of the application of the rules stated above.

Consider the case of a simple assignation of the form "destination:=source", where the source and destination both yield names. Here it is very often the case that the copy associated with the dereference of the

source is superfluous. Since the value yielded by the dereference becomes inaccessible as soon as the assignation is complete, an alias is safe whenever the assignation itself does not alter the source. That is, whenever the names yielded by the source and destination do not overlap.

The example `a[1:3]:= a[2:4]` shows that the copy may be needed. (An alternate method of addressing this particular problem is to insert dynamic checks in the assignation to move the value in a particular order depending upon the way in which the slices overlap.)

If the yield of the assignation is not immediately voided it may also be required to copy the destination before assigning. This is because the Report specifies that the yield of the assignation is the yield of elaborating the destination before the actual assignation is done. Thus the assignation "`destination:= source`" must be treated as though it were written:

```
(ref thing n = destination, thing w = source; n:= w; n)
```

If the elaboration of the destination involves dereferencing it is possible that when the assignment is done it overwrites the dereferenced location. If so the contents of that location must be saved. The following example demonstrating this possibility is due to SZOKE [4].

```
(mode node = struct(int val, ref node link);
  loc node n:= (1, n);
  ref node x = ref node(link of n) := (2, loc node);
  x is n)
```

This should yield `true`, but without copying the destination it will go wrong. This is because `link of n` overlaps `ref node(link of n)`. If no copy is taken when `link of n` is dereferenced then it becomes an alias which unfortunately is altered by the assignment to the related name yielded by `ref node(link of n)`.

If `x` yields a name, but the procedure `p` requires a value of the mode referred to by `x` then in the call `p(x, ...)` we may want to alias `x`. The scope of this alias will be that of the environ established by the call. Thus the copy associated with the dereference of `x` may be omitted if the call of `p` can not alter `x`. This is undoubtedly the most important application of these optimizations.

Even though the alias created when the copy is omitted during an assignation is the same as that created when the copy is omitted during

dereferencing, it is much more difficult to remove the copy involved in assignment than that in dereferencing. If the name is moved all names which refer to it, or to subnames of it, must be changed, and all identifiers which access it must be made to access the new value. This is impractical unless it can be shown that there are no such names or identifiers.

If the name has just recently been generated this is particularly easy to show. This suggests an implementation of variable definitions with initialization which proceeds by elaborating the source and then, leaving the result where it lies, generating the name onto it. This makes the translation of a variable definition with initialization exactly like an identity definition, except that the variable identifier accesses the address of the newly allocated location rather than the contents.

Consider the possibility of elaborating assignments at compile time. In 'thing v:= expr' if 'expr' can be elaborated at compile time, the name may be moved to the location in the 'constant' table where the value is stored. This is safe if the scope of the instance in the table can be limited to the scope of elaboration of the assignment. The value is actually created and stored at compile time (say in the primal environ). In order to treat the scope of the instance as so limited we must ensure that the address of the instance can only be used once, during the elaboration of the assignment.

Note that in a multi-user environment where programs may be shared between users, any shared program must be considered to be part of a routine text which may be called by several users. In such a situation this optimization is prevented unless each user gets a separate instance of the "constant" table.

6. APPENDIX

The language of the Revised Report is abstract enough that it is possible to imagine several totally different approaches to implementation on a random access machine. To illustrate this we describe an implementation in which the translation of the relation "to refer to" of the Revised Report differs from that in most current implementations of ALGOL 68. This approach was developed and used by P.G. HIBBARD et al. and is more fully described in [5].

For every value created during the elaboration of the program one or more blocks of storage are allocated. The size and number of these blocks depends upon the mode of the value (and the bounds if it is a multiple value); the contents depend upon the particular value. We will call these blocks of storage "value blocks". The value is then represented by a single pointer to the value block. If the value is not a name then the contents of the value block are never altered. It is therefore never necessary to copy without change the contents of a value block. If the yield of some action already exists in some value block then it is always safe to use it.

If the value is a name then the value block contains a pointer that points to the value block of the value to which it refers. That pointer is changed when a new value is assigned to the name. Assignment does not change the value block of the value, it only replaces a pointer to it in the value block of the name. The value block of a subname contains a pointer to the parent name together with an offset or index. Assignment to a subname is implemented according to the letter of the Revised Report; i.e. a new stowed value is created which differs from the one originally referred to by the parent name in only one element. The parent name is then made to refer to that new value.

In its un-optimized version this implementation scheme is outrageously inefficient. A few optimizations can improve it to the point where it is competitive with the more traditional implementations. The most important of these optimizations is to keep a reference count with each block of storage. If the reference count of the original value referred to by the parent name is equal to one when a subname is assigned to, then instead of copying nearly the whole value and deleting the old one (which will no longer be referenced at all), the value block is updated in place.

The reason for describing this implementation strategy is to illustrate the point that there are a number of arbitrary decisions which must be made at the very outset which greatly affect the relative cost of various actions. It is only after choosing a particular model of un-optimized computation that it is possible to begin inserting optimizations which depend upon the context of the action. It is an impressive accomplishment to have written a language definition which makes the intended semantics clear enough that it is possible to discuss optimization while still allowing for such divergent approaches to implementation.

REFERENCES

- [1] VAN WIJNGAARDEN, A. et al., *Revised Report on the Algorithmic Language ALGOL 68*, Springer-Verlag, 1976.
- [2] RAIN, M., *Some formal aspects of Mary*, ALGOL Bulletin 34 (July 1972), p. 45-81.
- [3] BRANQUART, P., J.P. CARDINAEL, J. LEWI, J.P. DELESCAILLE & M. VANBEGIN, *An optimized translation process and its application to ALGOL 68*, Springer-Verlag, 1976.
- [4] SZOKE, P., *Some remarks on new instances and garbage collection*, in Proceedings of the Strathclyde ALGOL 68 Conference, SIGPLAN Notices 12, 6 (June 1977).
- [5] HIBBARD, P.G., P. KNUEVEN & B.W. LEVERETT, *A stackless run-time implementation scheme*, in Proceedings of the Fourth International Conference on the Design and Implementation of Algorithmic Languages, (ed. R.B.K. Dewar), Courant Institute of Math. Sc., New York, 1976.
- [6] VAN DER MEULEN, S.G. & M. VELDHORST, *Torrix I*, Mathematisch Centrum, Amsterdam 1978.
- [7] VAN DER MEULEN, S.G., *ALGOL 68 Might-Have-Beens*, in Proceedings of the Strathclyde ALGOL 68 Conference, SIGPLAN Notices 12, 6 (June 1977).

ON THE DESIGN OF AN ABSTRACT MACHINE
FOR A PORTABLE ALGOL 68 COMPILER

L.G.L.T. MEERTENS

ABSTRACT

This paper indicates a line of reasoning, the *cut principle*, that may be applied in the design of an abstract machine for a portable ALGOL 68 compiler.

1. INTRODUCTION

A portable program is a program that can be moved to a variety of computers with relatively little effort. The effort has to be compared to the effort of creating a brand-new program. One may have portable editors, compilers, and even operating systems. For compilers we run into a problem. Usually, moving a program implies that its meaning remains the same. But if the meaning of (the program which is) the compiler is unaffected in the act of moving it to another computer, it will not generate code for that computer. One simply obtains a cross-compiler. In fact, this is the easier part of moving a compiler.

One approach is to make the code-generation part of the compiler 'adaptable'. If the idea is that the compiler, together with some documentation, can be mailed elsewhere, one should realize that this strategy requires a thorough understanding of the working of the compiler by the recipient. Also, adapting the code generation does not suffice. The run-time environment must still be created. The assumptions concerning the environment underlying the code generation must be stated very clearly. A special way of making the code generator adaptable is to parametrize it: number of registers, size of words, etc. Though promising on paper, this approach is not really practicable. The variety among computers and their particulars are such that they are not readily expressible by means of a manageable number of parameters. Feeding a formalized description of the target computer may be sensible for a compiler-compiler, but would give rise to excruciatingly slow code generation for a direct compiler.

Another approach will be followed here. Design a 'machine-independent abstract machine' ('MIAM') that can be modeled on a variety of computers with moderate effort. Let the compiler generate object programs in MIAM code. The definition of the MIAM and its code provides a clear interface, both for the compiler writer and for the recipient of the compiler. If, moreover, the compiler itself is available in MIAM code (e.g., by writing it in its source language and once performing a bootstrap), the moving of the compiler and the adaptation to the new target combine into one act.

The construction of an ALGOL 68 compiler is a complex task. Even if one does not aim at portability, the definition of an intermediate abstract machine may help to reduce the complexity. The compiler design is then factored into two parts. So, in designing a portable ALGOL 68 compiler, there are two *distinct* reasons for introducing an abstract machine. The

desiderata (in terms of the abstract machine) for these two reasons are not a priori the same. An interesting question, especially from a practical point of view, is whether they can be combined and, if necessary, reconciled.

This paper investigates this question and indicates a line of reasoning, the *cut principle*, that may be applied in the design of an abstract machine for a portable ALGOL 68 compiler. This principle is next illustrated in a number of design decisions for a specific MIAM. (The fact that the source language is ALGOL 68 is extremely relevant for these decisions themselves, but far less so for the cut principle. It is expected that the same principle would provide guidance in the design of, say, a PL/I or ADA MIAM.) Also, the issue of providing a proper run-time support system is addressed.

2. DESIDERATA FOR THE ABSTRACT MACHINE

If the desiderata stemming from the two reasons for introducing an abstract machine (portability and reduction of complexity) do not comply, the portability desiderata should be weighed more strongly. This follows immediately from the essence of the portability idea. Moreover, should the two sets of desiderata turn out really irreconcilable, one should not hesitate to introduce two distinct abstract machines. But, as we shall see, the situation is not that bad.

As for portability, the MIAM should, in a sense, be as close as possible to the computers in the variety under consideration. Unless this variety is extremely restricted, it is not helpful to look at the 'union' or the 'intersection' of these computers. The first would yield an unwieldy monstrosity for the MIAM, whereas the second is bound to be empty. Rather, one should attempt to find the center in the space of abstract properties of the computers: an idealized architecture. For example, an indexing facility is common to a great variety of computers, but the actual details differ considerably. The MIAM should then contain an idealized indexing capability. Because of the presence of the indexing facility, the MIAM falls, in this respect, in the union of abstract properties. But, since we have an idealized version, it falls in the intersection too. So the proper abstraction is that in which the union and the intersection coincide as much as possible. The greater the variety considered, the higher the abstraction required, up to the level where portability becomes a pipe-dream.

Now, consider the problem of reducing design complexity. For the moment we assume that a fixed target computer is given. This is the solid ground atop of which the compiler is to be erected. At the other end, the 'ceiling', we have the 'hypothetical computer' in terms of which the semantics of ALGOL 68 is defined. We want to construct a well-chosen mid-level.

In the design of the compiler, a good many problems have to be solved. The hypothetical computer is able to climb up and down the 'program tree' in order to elaborate 'constructs' (parts of the tree descended from one node in the parse tree). Typically, a construct C is composed from other constructs, and in order to elaborate C its component constructs have to be elaborated first. These elaborations, which are performed 'collaterally', yield values, and from these values the yield of C is obtained. The hypothetical computer is able to deal with objects of arbitrary size and does not worry about relinquishing objects that have become inaccessible. In contrast, most typical present-day computers proceed essentially by serial execution of instructions. They have an essentially linear memory of small, fixed-size cells that are limited in number. Each instruction modifies one, or at most a few, of these cells. So typical problems that have to be addressed are the serializing of collaterality, modeling large objects in terms of cells, and designing a storage-allocation regime. For reasons of efficiency, an attempt must be made not to duplicate, copy or shift objects unnecessarily. It must be possible to free storage whose occupant has expired. These are but a few of the problems.

The details of the solution will, of course, depend on the actual target computer. Still, the solution of the major problems should, preferably, not depend on peculiar features of the hardware. On the contrary, one should make a strong effort to abstract from the details of the computer. This entails a potential loss of optimality. But a well-chosen partitioning should provide a support in designing code generation which more than compensates for this loss. Stronger even, it should direct one to spending one's optimizing efforts where they are most worth-while. The optimizations which are typical for ALGOL 68 and which require understanding of the global behaviour of constructs can be designed without distraction or hindrance by unimportant details. The 'peephole' optimizations, depending on the target computer, can be found relative to the (comparatively simple) specifications of the abstract machine, without danger of entanglement in intricate interaction between ALGOL 68-dependent and machine-dependent

properties.

So the abstract machine should be abstract in the sense that it reflects the typical low-level properties of the target computer (such as the linearity of a memory of cells), but in an idealized form, with the gory details stripped off. Now, if this is desirable if the actual target is known, it is compulsory if the target is still floating.

So a strong convergence displays itself between the two sets of desiderata. Whatever the reason for introducing an abstract machine before the final production of code for an actual target computer, in either case we want the abstract machine to model, in an idealized way, typical abstract properties of the target computers. And the abstraction criteria are essentially the same. Still, we do not have a good criterion which of the countless abstract properties we may choose to perceive (they are not present in the variety of computers in an objective sense) should be included in the MIAM. Before this issue is addressed, however, we should first turn our attention to an issue that has been disregarded until now.

3. THE RUN-TIME SUPPORT SYSTEM

In compiling a statement such as 'i:=j+k' the code generated for an actual computer might be something in the spirit of

```
LOAD descriptor of i;
LOAD descriptor of j;
CALL dereference subroutine;
LOAD descriptor of k;
CALL dereference subroutine;
LOAD descriptor of +;
CALL call subroutine;
CALL assign subroutine.
```

Might be. But this code will be frightfully inefficient. Most compiler writers will prefer trying to generate less treacly code. By generating straight in-line code not only does one do away with the overhead of the subroutine-call mechanism, but also with the overhead of interpreting, inside the subroutine, the situation met. Moreover, if one does a reasonable job, the resultant code is most likely less bulky too.

Conclusion: no subroutine calls. But this conclusion is unwarranted. Just consider the garbage collection capability needed for a full ALGOL 68 implementation. One surely would not want to have an in-line version of

the garbage collector at all positions in the code where the process might run out of memory. In other cases the choice may be less clear-cut, but still a good point can be made for a call instead of in-line code, e.g., for computing sines.

The collection of subroutines created this way forms the run-time support system. It 'supports' the object code. Abstractly viewed, the computer has been 'enhanced' by adding new capabilities in the form of new instructions. This means that the design of a MIAM is not so straightforward a task as one might conclude from the considerations of the previous section. New properties may be added more or less at will. Also, a new problem is raised (already hinted at in the introduction): that of the portability of the run-time system. The design and implementation of such a system is no mean task; if it is left to the recipient of the portable compiler, the portability is seriously impaired. For the time being it will be assumed that this problem has been solved in some way.

4. COMPILATION AS SYMBOLIC INTERPRETATION

One of the objections that is voiced, time and again, against the 'traditional' operational style of describing the semantics of programming languages, is that the description may obscure some very clever way of obtaining the same net effect. True as this may be, a 'reasonable' operational description is also a good handhold for the compiler writer. The process of code generation may be viewed as the symbolic execution of the source program by interpreting it, step by step, in accordance with the semantics. The 'low-level' facts that can be found out statically are derived during this symbolic execution. Where actual execution would be necessary, code is emitted, as modified by the facts already found. This idea is described in HANSON [7] with the misnomer 'lazy evaluation'. (An appropriate term would be 'lazy code generation'.) Essentially the same idea is implemented by the 'mvalues' of BOOM [2]. It is a special case of the more general 'partial computation principle' described in ERSHOV [5].

The code obtained this way does not take, in each case, the most complicated situation conceivable into account, but is customized to the actual complexity of the situation at hand. This is important, since the actual complexity tends to be rather small in the majority of cases (GRUNE [6]). One of the most important contributors to code simplification is the 'mode' of a construct, being a static summary of dynamic properties

of the possible yields. Other code improvements that are obtained in this way are the omission of scope checks or checks on nil in assignment and dereferencing in the majority of cases.

It should be stressed that each compiler writer applies the partial computation principle - either aware or unaware -. It is the very essence of compiling. The advantage of taking the viewpoint described in this section is that the consequences of design decisions become clearer. The decision to add a capability to the run-time support system is then the decision to stop symbolic interpretation at that level and to interpret, instead, at run time.

5. THE CUT PRINCIPLE

In designing a MIAM for ALGOL 68 we are faced with decisions of the form: should this capability be included as a 'primitive' property of the MIAM, or should it be modeled in terms of lower-level properties. For the code generator, the mid-level interface which is determined by the MIAM is its perception of solid ground. The 'substratum', where MIAM properties are expressed in lower-level primitives, is hidden to it. The actual MIAM properties must be used to model the properties of the hypothetical computer. This modeling is performed explicitly in the 'upper world'. (In the perception of the ALGOL 68 programmer, the world of the code generator is, of course, below the ground.)

The collection of these decisions makes a cut in the set of (potentially) realizable MIAM properties that may play a role in modeling the hypothetical computer. In many cases it is immediately obvious whether a particular item should be placed in the substratum, or be left to the upper world for realization. For example, multiplication of real numbers should be buried in the substratum. Computing the factorial function - even for the hypothetical computer only a potential capability - does not have a place in the MIAM.

In other cases things are not so clear. Take, e.g., slicing. This is not so primitive as real multiplication, but it is a pretty fundamental operation that is easily isolated. An other example is assignment. Some copying capability must need be present in the MIAM. But for what objects? For 'bytes'? For multiple values? Note that the notion of a copy is foreign to the 'revised' hypothetical computer. (To my taste, rightly so: the task of making copies follows from invariants of the modeling, not from the semantics itself. Not taking copies is only an optimization with respect

to a strategy for maintaining these invariants. So the revised semantics are an instance of the value of a less 'operational' description.)

Some principle to guide the decisions would be very helpful. It is here that the symbolic interpretation idea steps in. In order to follow this idea, the compositions expressed in the semantics must be modeled in terms of the MIAM. So, for example, stowed (structured and multiple) values must be modeled by composite objects. The decision how to perform this composition belongs to the upper world. Therefore, the 'underworld' should be completely unaware of the way of composition; its task is to realize the ground on which the code generator can be constructed, and it must, therefore, be completely insensitive to upper world design decisions. In some sense, this is a closure property of the upper world: once a property of the hypothetical computer is placed above the ground, it takes other properties with it.

We now have a criterion for deciding when to assign properties to one side of the cut. But when are properties assigned to the substratum? Keeping in line with the tradition of computer science, we choose for a minimal closure: if this is not prevented by the principles already set forth, properties needed to model the hypothetical computer are assigned to the MIAM. This is, in fact, very reasonable. Otherwise the design of the code generation would have to create new abstractions from the MIAM primitives, thereby creating effectively a new, higher-level, MIAM. Now, the only abstractions supported by the code generation are those that are meaningful in terms of the hypothetical computer.

Summing this up, we have the

Cut Principle: Composite properties of the hypothetical computer are not properties of the MIAM. No property of the MIAM depends on the way of modeling such composite properties. Within these limits, MIAM properties are as high-level as possible.

Simple as this principle may sound, it should be clear that it is not a straightforward yes-or-no test. Its application will be illustrated on various decisions.

6. MODELING ALGOL 68 VALUES

The 'primitive values of the MIAM are almost the same as those of ALGOL 68: integers and real numbers of different lengths, truth values, characters. Primitive 'pointers' are used to model names, but also for other purposes.

A new primitive value is the label. A routine may then be modeled by a pair, consisting of a label (for the construct of the scene) and a pointer (for the environ). Other than in the definition of ALGOL 68, L BITS and L BYTES are treated as primitive; their compositeness in the definition is considered a descriptive artefact. Strings are treated as multiple values, though.

According to the cut principle, composite values must be modeled by composition in the MIAM. The MIAM primitives must not depend on particular preferred ways of modeling. Therefore, the MIAM must have some general form of composition. To this purpose, 'offsets' are used. An offset, added to a pointer, gives a new pointer. Offsets are static entities, not run-time objects (like field-selectors in ALGOL 68); they are defined in pseudo-instructions in the MIAM-code. A cascade of such instructions allows to map a sequence of objects on an 'area' of memory. As a by-product, a static type is assigned to the thus composed sequence of types.

This can be used to model structured values, but also other composite objects, such as locales and environs, multiple values (consisting of a descriptor and a pointer to an area for the elements), routines and parameter passing.

By defining several maps for the same area, unions can be accommodated, and space may be assigned efficiently for the 'work stack' of anonymous temporary yields.

This idealized version takes the possibility into account that in the actual target computer some types cannot be assigned to just any address, but only to, say, addresses that are a multiple of four. Also, it does not assume that the mode of addressing is the same for different types. If a type with multiple-of-four addresses would leave holes, e.g., because the values use only three cells, these holes can be used in principle also. (Computers do exist where this would be useful.)

Some further primitives of the MIAM allow to map a sequence (of unspecified length) of objects of one same type on an area and to access dynamically the k-th object. By using this, together with the primitives mentioned above, a complete modeling of ALGOL 68 multiple values can be made.

7. STORAGE ALLOCATION

Given the fact that maps may be set up for accessing objects in a given area, we still need primitives for obtaining (access to) areas as a whole. The storage allocation regime must allow storage to be freed when an area has become inaccessible. In contrast to ALGOL 60, a stack regime is not sufficient, for three reasons:

- (a) Because of the parallel actions, environs may have to be switched, so that at least a 'cactus stack' is needed;
- (b) The size of objects in a 'stack frame' might increase unboundedly, because of flexible multiple values;
- (c) Objects may be created whose life-time exceeds that of the current stack frame, because of global-generators.

So another storage allocation regime is needed, e.g., one supported by a memory management system allowing garbage collection. Not only does the semantics of ALGOL 68 require a more general regime; it also allows realization by means of a memory management system that is simpler and more efficient than the most general system. The scope restrictions of ALGOL 68 are relevant here. For these to be exploited, they must be inherited in some sense by the MIAM, giving rise to invariants. Since memory management is then based on these invariants, they obviously define some interface between the upper-world modeling of the ALGOL 68 semantic actions when abstracting from memory management (assuming, e.g., an unlimited resource of areas) and the memory management support. According to the cut principle, the memory management operations that respect these invariants should not be abstracted from MIAM primitives, but must be MIAM primitives themselves.

The main primitives obtained this way are instructions for obtaining access to a fresh area, as it were newly created. Areas can be used to model locales and generators. (The sample-generators in variable-definitions can usually be accommodated in a locale, using an obvious optimization. On the other hand, it is convenient to use a separate area for unions and for the dynamic part of multiple values always, even if no generator is involved.) Areas that are used to model locales must be 'chained' explicitly, using two pointers (playing the role of 'upon' and 'around' in the ALGOL 68 semantics).

The usual mechanism for efficient access to various active locales in a 'block-structured' programming language is the display. Although a

display can be modeled with the MIAM primitives already described, this would be rather costly. On the other hand, introducing the display as a MIAM primitive constitutes a violation of the cut principle, since it would be a choice for one particular way of modeling a composite property of the hypothetical computer. Fortunately, it is quite simple to do away with the display (DEWAR [4]). The display is simply an array of the addresses of locales present in the 'around' (lexicographic) chain. It obviates the costly process of following that chain backwards each time for a non-local access. On the other hand, it requires the maintenance of an invariant, which is certainly not negligible in cost. The idea that allows to dispense with the display can be sketched in terms of ALGOL 68: a text of the form

```

BEGIN AMODE x = ...;
  ...
  BEGIN ...
    .
    .
    .
    BEGIN ...
      text using (x)
    END
  .
  .
  .
  END
  ...
END

```

is compiled as though the source text had read

```

BEGIN AMODE x = ...;
  ...
  BEGIN ...
    .
    .
    .
    BEGIN AMODE x' = x; ...
      text using (x')
    END
  .
  .
  .
  END
  ...
END.

```

The cost of following the around chain is incurred only once, at the inner identity-definition. Obvious variations on and refinements of this scheme are possible, but need not concern us at the moment. Notice, however, that the inner range now using x' may be contained in a routine-text. This works fine, but some care has to be exercised in order that no undue restriction of the scope of the routine results (because of the 'necessary environ').

Some attempts to assess the cost of a refined version of this scheme compared to the traditional solution suggested quite strongly that having no display is the more efficient approach.

8. PARALLELISM

One of the assumptions in the design of the MIAM is that the actual target computer has one processor. This permits a major simplification. Collateral elaboration must be modeled in terms of serial execution. (It appears, anyway, that programmers tend not to write constructs permitting collaterality except where they have little choice.)

A special case of collaterality is given by parallel actions. These can be synchronized by semaphores. If these are to be serialized (which is desirable anyway, for it keeps memory management simple), MIAM programs must be able to switch environs in the middle of a block. Now, in the hypothetical computer, there is no such thing as the 'currently active' process. Applying the cut principle, we find that the notion of a process and the capability of process switching should be MIAM primitives.

9. FLOW OF CONTROL

ALGOL 68 features a variety of constructs for governing the flow of control. The corresponding actions are composed of 'Steps'. So the cut principle suggests that these actions do not correspond to MIAM primitives, but are also modeled by more primitive actions: the simple and the conditional parameter.

In fact, the MIAM has a large variety of conditional jumps. This simplifies the modeling of various actions of the hypothetical computer. The standard relational operators are only available through these primitives; e.g., an assignment like

$$p := x > 0$$

has to be compiled as though the source text has

```
p:= IF x > 0 THEN TRUE ELSE FALSE FI.
```

(This decision is, of course, independent of the cut principle. It simply reduces the number of MIAM primitives.)

10. ASSIGNMENT

The modeling of composite values belongs to the realm of upperworld decisions and assignment in the MIAM depends on the way of modeling chosen. So, by the cut principle, full-fledged assignment cannot be a primitive. Instead, symbolic execution of assignment generates the code for copying values, tailored to the mode of the value. Some obvious optimizations for going through multiple values can be generated immediately in the object code. Also, in some cases a pointer may (or must) simply be copied, whereas in an other case the object pointed to has to be copied. Keeping track of this is clearly an upper-world task.

Therefore, the MIAM has only a copy primitive that is primitive in more than one sense. This choice is dictated by the cut principle for another reason also: assignment is already a composite action in the hypothetical computer.

11. INSTRUCTION FORMAT

The design of a machine as discussed here entails many more decisions than can be related to the cut principle. One of these decisions is that a MIAM program is built from 'instructions' of a classical type. Much can be said in favour of intermediate graph representations, and the cut principle would still be applicable. On the other hand, it is not particularly hard to build such a graph representation (including data flow analysis) from a MIAM program, which may be viewed as a concrete linearized representation of a graph.

However, the idea is that more or less conventional techniques, like macro processing (with the necessary bells and whistles), should be applicable to the problem of transforming MIAM code to actual target code. The general format of a MIAM instruction is given by:

```
instruction: keyword, comma, argument list, semicolon.
```

Most instructions have three or fewer arguments. At most one of these arguments is used to determine an access where the result of an operation is stored; such an argument is always the last one of the list. Some typical examples of instructions will be given, and some (in fact most) possibilities for arguments will be illustrated in these examples. The particular design choices made will not be defended; in many cases they are just some choice, neither better nor worse than other sensible choices.

(a) COPY, INT1, I6.2, T4;

This is a copy instruction. The first argument indicates that the object to be copied is a LONG INT (the '1' stands for the 'size'). The integer can be found by adding the sixth offset to the current top locale pointer next the second offset to a pointer found there. (In ALGOL-like style, $M[M[T+off6]+off2]$.) The 'I' corresponds to 'indirect'. The '6' and '2' are effectively tags denoting offsets. The copy has to be deposited at the site pointed to by a pointer in the top locale, found by applying an offset known by '4'. (Not at the fourth 'field' of the top locale; there is no implicit 'upreferencing' of result arguments.)

(b) COPY, PTR, &I6.2, &S5;

The '&' takes off one level of indirection. Not the integer would be copied, but a pointer to it. Similarly, '*' adds one level of indirection. The use of '&' and '*' is severely limited; e.g., '&&' or '**' is always illegal. The result is deposited in the topmost-but-one locale. (Only the top two locales and the bottom locale have the privilege of being accessible by arguments of a special form. It might appear that the use of 'S' violates the cut principle. But its meaning is defined independent of the fact that the MIAM is used to model ALGOL 68. The MIAM leaves, e.g., a good deal of freedom in modeling ALGOL 68 parameter passing, but S-arguments are very convenient for many ways of passing information from the caller to called MIAM code.)

In this instruction, it is clear from the appearance of the second argument that a pointer is involved. In fact, the first argument of a copy instruction is *always* redundant, but it is nice to have the relevant information locally available.

The execution could conceivably result in a scope violation. Remember

that the MIAM has inherited the scope concept. In a sensible implementation of the MIAM this should not be checked. The idea is that there exists a contract between the code generator and the MIAM. The code generator guarantees that it will never generate code that might result in a dynamically undefined situation. In return, the MIAM promises speedy execution. If the code generator cannot find a proof that the scope restrictions are satisfied, it has to emit explicit code for a dynamic scope check.

(c) IFIS, W13, X, A3;

This instruction compares pointers; it may be used to model ':='. The 'W' indicates that the argument concerns a temporary object that will never be inspected again. Otherwise, it is equivalent to 'T'. 'X' is a literal, roughly corresponding to NIL. The last argument is a label, but of a special kind. A jump to an A-label causes the execution of the program to be aborted. The number may be used to generate an appropriate error message.

(d) SCOPE, T13, &T14;

The first argument of a scope instruction determines a pointer. The 'scope' (an integer) of the area into which the pointer is pointing, is delivered in the position given by the last argument.

(e) SPAWN, 3, L88, L90, &T15;

A parallel action descriptor is created, using the labels given as arguments. The spawn instruction has a variable number of arguments. The first argument is a literal and determines the length of the variable part of the list.

These examples give some idea of the flavour. Many combinations of arguments are excluded (syntactically). For example, the following is illegal:

COPY, LAB, A3, *S5;

A-labels may only be used if the jump to the label cannot be deferred. Also, '*' cannot appear in a result argument; it is one level of indirection

too many. Thus, the MIAM is not orthogonal in design. The unorthogonalities have been chosen in such a way that it is expected that they are not cumbersome in the design of the code generation, and will often be helpful in the implementation of the code transformer. If the latter is not the case, it may be helpful that these unorthogonalities are pure restrictions, not quirks. So there exists an orthogonal closure of the MIAM. An implementation of that orthogonal closure is a valid implementation of the MIAM.

12. SOME EXPERIMENTS

In order to obtain an idea how efficient this approach is, some pieces of ALGOL 68 text were translated by hand into MIAM code, and next from MIAM code to actual machine code. These program fragments were chosen to incorporate some of the heavily used constructs (GRUNE [6]). An attempt was made to simulate rather simple-minded algorithms.

For the ALGOL 68S compiler on the PDP11/45 a speed-up was observed of a factor of 5.9. This high factor can be explained by the fact that the ALGOL 68S object code uses subroutine calls rather extensively. Also, the length of the code was decreased by some 6 percent.

For the CDC ALGOL 68 compiler on the Cyber 72 the code was sped up by a factor of 1.5, but at the expense of a code-length increase of 8 percent. The fact that speed-up was obtained here was a surprise; the CDC compiler was designed very specifically for the (rather anomalous) family of CDC computers to which the Cybers belong. On the other hand, the precision suggested by the figures quoted is misleading; it may well be that the MIAM approach would behave less well on a larger variety of programs.

13. CONSTRUCTING A PORTABLE RUN-TIME SYSTEM

ALGOL 68 programs are embedded in the standard environment created by the 'standard-prelude'. An approach to make this environment portable is to use the separate-compilation facility. This is most important for the transput. Instead of the text given in the Revised Report, the implementation model of ALGOL 68 Transput (VAN VLIET [13]) should be used. This approach requires extending ALGOL 68 with some additional capabilities, enabled only during compilation of the standard-prelude.

A quite different matter is the construction of a portable runtime support system for the 'enhanced' MIAM. This is possible by modeling the MIAM again on a lower-level (but still machine-independent and abstract) computer. This 'MIAC' already has many of the properties of the MIAM, but of course not those that are expressed in terms of the MIAC. The recipient of the compiler still has to model most MIAM instructions directly on his actual computer, but the 'hard' ones are obtained by implementing the MIAC. In this way parallel processing can be described, and also a complete memory management system.

The description of a complete memory management system for the MIAM in terms of MIAC code clearly demonstrated that (at least as far as memory management was concerned) the cut made by the MIAM was the proper one. First of all, the desideratum of reduction of complexity was entirely complied with. It turned out possible to single out all properties of the MIAM relevant to the memory management problem and to describe them in a simple abstract model of the MIAM. This abstract model, being free of details irrelevant to memory management, enables one to study the problem of memory management in isolation. A firm hold on the problem is thus obtained and possible solutions are more easily surveyed. Furthermore, invariants of the system can be proved in the model without great difficulty and be used subsequently to increase efficiency. The correctness of the entire memory management system can even be proved without excessive effort (JONKERS [9]).

The desideratum of portability was also satisfied, in the sense that an efficient machine-independent memory management system for the MIAM could be described in MIAC code. This system uses a modified version of the classical approach with a stack and a heap. Storage for all areas other than those corresponding to global-generators is allocated on the stack. This immediately implies that the stack is not a pure stack, because the stack behaviour may be obstructed as a consequence of the use of flexible multiple values or parallel actions (see Section 7). By popping inaccessible stack areas from the top of the stack as soon as they occur there, the stack behaviour is maintained as much as possible, however. If finally a clash between stack and heap occurs, the amount of garbage in the stack can be determined with relatively little effort. If that amount is large, as compared with the size of the store, the stack is compacted. Due to the inheritance of the ALGOL 68 scope restrictions by the MIAM, this compaction is strictly local to the stack (no pointers in the heap need be updated)

and can be quite efficient (JONKERS [8]). If the amount is relatively small, a full garbage collection is performed and both the stack and the heap are compacted. This also requires a marking phase and is considerably more expensive than a mere stack compaction.

14. RELATED WORK

There exists an equivalence between (abstract) machines and (programming) languages. Each design of an intermediate code is, in some sense, at the same time the design of an abstract machine, and vice versa. Often the choice between the two in presentation is only a matter of emphasis.

In the abstract machines designed for implementing ALGOL 68, a major distinction can be made between 'high'- and 'low'-level machines, in two respects. One distinction is with respect to the language level: an abstract machine has a high level if its primitives are clearly and directly related to ALGOL 68. The other distinction is with respect to the machine itself: it has a low level if its architecture is close to a computer. (The latter distinction is becoming increasingly fuzzy.)

The intermediate code in the implementation described by BRANQUART et al. [3] and the abstract machine of KOCH & OETERS [10] are high-level in both respects; mapping these on conventional architecture still entails much work, but a good efficiency can be reached. Direct hardware implementation is just becoming feasible.

The abstract machines described in TANENBAUM [12] and LANE [11] have a high language-level and a low machine-level. Built in hardware, they would relate to ALGOL 68 as the Burroughs B6600 and B6700 to ALGOL 60. Implementation on conventional machines is relatively easy, but only techniques like micro-programming or emulation can prevent a substantial loss of efficiency.

Low-level in both respects is ZCODE, described by WALKER, BOURNE & BIRRELL [14]. The MIAM described here fits best in this category. However, in both respects it has a curious mixture of high- and low-level properties. For example, the scope concept is directly related to ALGOL 68, and the existence of parallel action descriptors as a primitive is unconventional. ZCODE is an object code for ALGOL 68C and does not cover the full language. Moreover, it has been designed with machines with very regular registers in mind. Its use for other machines presents severe difficulties (BIRRELL [1]).

No assumption about registers, or even their existence, is made in the MIAM.

Still another possibility is to use an abstract machine that has not been specifically designed for ALGOL 68, such as JANUS. For the problems one may encounter with this approach, see BOOM [2].

15. CONCLUSION

A sketch has been presented of the design of an abstract machine for an ALGOL 68 compiler that serves two purposes at the same time: achieving a high degree of portability, and reducing the complexity of the design of the code generator. It appears that this abstract machine may be implemented quite efficiently on many present-day computers.

If one views the abstract-concrete scale as one that scores the recognizability of the ALGOL 68 origin, the machine described has a curious mixture of 'high'- and 'low'-level properties. And yet, the design has been guided by one single principle. Indeed, if one views the distinction between 'abstract' and 'concrete' in terms of commitment, then the abstract machine is abstract in the sense that it attempts not to commit the compiler writer in the freedom of developing a code-generation strategy, including the optimizations that are reasonable for an ALGOL 68 implementation, nor the recipient of the portable compiler in the approach, including optimization, for realizing it on the actual target computer.

A 'feasibility proof' for the MIAM has not been given. Such a proof would consist of the construction of an ALGOL 68 compiler with the MIAM as target, together with the construction of 'code transformers' from MIAM code to a variety of actual computers. One problem that has come up should be mentioned. In mapping the MIAM to machines with registers, one should hope to be able to keep variables temporarily in registers in loops (e.g. the internal variables used for controlling the traversal of multiple values). To do this, it must be proved that these variables are not affected in the mean-time by other instructions: their result arguments may not be an 'alias' of the variables concerned. In many cases the necessary information is readily available on the ALGOL 68 level, but is buried in the MIAM code. It seems possible to retrieve it, but this appears to require the use of algorithmic analysis techniques that are more complicated than was intended for code transformation. Adorning the MIAM code with the information from the ALGOL 68 level threatens not only the cut principle,

but the whole idea of having a clean interface. A partial, probably satisfactory, solution, complying with the cut principle, would be to give some offsets a new attribute, indicating that non-transient pointers to locale sites corresponding to these offsets will never be set up.

It is conceivable, and in fact likely, that during the construction of a feasibility proof other problems will present themselves.

Acknowledgement. The present research has been strongly influenced by studying the systematic treatment of BRANQUART et al. [3].

REFERENCES

- [1] BIRRELL, A.D., *Problems in implementing ALGOL 68C*, in Proc. of the 1975 Int. Conf. on ALGOL 68, G.E. Hedrick, ed., Oklahoma State University, Stillwater, 1976.
- [2] BOOM, H., *Code generation in ALGOL 68H: an overview*, Report IW 103, Mathematical Centre, Amsterdam, 1978.
- [3] BRANQUART, P., J.-P. CARDINAEL, J. LEWI, J.-P. DELESCAILLE & M. VANBEGIN, *An Optimized Translation Process and its Application to ALGOL 68*, Lecture Notes in Computer Science 38, Springer, 1976.
- [4] DEWAR, R.B.K., *oral communication*, March 1979.
- [5] ERSHOV, A.P., *On the partial computation principle*, Information Processing Letters 6 (1977) 38-41.
- [6] GRUNE, D., *Some statistics on ALGOL 68 programs*, SIGPLAN Notices 14, 7 (June 1979) 38-46.
- [7] HANSON, D.R., *Code improvement via lazy evaluation*, Information Processing Letters 11 (1980) 163-167.
- [8] JONKERS, H.B.M., *A fast garbage compaction algorithm*, Information Processing Letters 9 (1979) 26-30.
- [9] JONKERS, H.B.M., *Designing a machine-independent storage management system*, Report IW 148, Mathematical Centre, Amsterdam, 1980.
- [10] KOCH, W. & C. OETERS, *An abstract ALGOL 68 machine and its application in a machine-independent compiler*, in GI-5. Jahrestagung, J. Mühlbacher, ed., Lecture Notes in Computer Science 34, Springer, 1975.

- [11] LANE, H.J., *An ALGOL 68 machine and translator*, UCLA-ENG-7369, Comp. Science Dept., UCLA, Los Angeles, 1973.
- [12] TANENBAUM, A.S., *Design and implementation of an ALGOL 68 virtual machine*, Report IW 4, Mathematical Centre, Amsterdam, 1973.
- [13] VAN VLIET, J.C., *ALGOL 68 Transput, Part II, An Implementation Model*, Tract 111, Mathematical Centre, Amsterdam, 1979.
- [14] WALKER, I., S.R. BOURNE & A.D. BIRRELL, *ALGOL 68C Implementation Guide*, Comp. Lab., Cambridge, 1974.

AN IMPLEMENTATION OF MODULAR COMPILATION IN ALGOL 68

G.J. FINNIE & M.C. THOMAS

ABSTRACT

This paper describes the modules and separate compilation facility of the RS family of Algol 68 compilers, and shows how it has been implemented in the RS Algol 68 compiler for ICL 2900 series machines. Although the major part of the paper concentrates on the implementation within the ICL compiler, the paper starts with a view of the user requirement for modules and separate compilation, and with some discussion of the facilities provided by the RS system and how they compare with the official IFIP scheme.

1. INTRODUCTION

Modular compilation systems have been around in one form or another for many years, with schemes ranging from the totally permissive library mechanisms of FORTRAN to the restrictive ideas of Simula. Ideas about modularity and its place in the programming process developed rapidly in the years which followed the publication of the Algol 68 Report, yet unfortunately no modular compilation proposal was included in the 1974 revision. As a result a number of alternative schemes for Algol 68 modular compilation facilities have been implemented, and only recently has an official recommendation been published [2]. As yet there are no implementations of the recommendation (although we understand that the CDC compiler will be enhanced to follow the recommendation soon).

The publication of the definition of Ada has stimulated interest in modular compilation and separate compilation, and this interest has been further fuelled by the commitment of NAG Ltd to produce a proper Algol 68 NAG library. Although this library will be implemented initially in the ICL Algol 68 compiler (2900 Algol 68, a member of the RS family of revised Report compilers) it is obviously highly desirable that the library is readily usable with all the major Algol 68 compilers.

No modern programming language can be considered suitable for serious use on major programming projects unless it provides facilities for secure modular program development. Most current implementations of Algol 68 provide such facilities; as yet, few implementations of other languages do so. Unfortunately the Algol 68 position is marred by the differences in the implementation of these facilities in different compilers.

This paper describes the modules and separate compilation facility of the RS family of Algol 68 compilers, and shows how it has been implemented in the RS Algol 68 compiler for ICL 2900 series machines. Although the major part of the paper concentrates on the implementation within the ICL compiler, the paper starts with a view of the user requirement for modules and separate compilation, and with some discussion of the facilities provided by the RS system and how they compare with the official IFIP scheme.

2. A USER'S VIEW OF MODULAR COMPILATION SYSTEMS

This section of the paper provides a background to the subjective

assessment of the RS modular compilation scheme which follows. It will readily be apparent that in two areas at least, one of the present authors disagrees fundamentally with the authors of the IFIP recommendation for modular compilation, and with the designer and implementors of the RS system as well.

In what follows, the topic of 'modular compilation' (where program text is constructed in lexically-parallel modules with special visibility rules) will be treated as if it were largely identical with 'separate compilation' (where the resulting modules are presented to the compiler as separate compilation units). The view is taken that forcing the programmer to compile each module individually to filestore imposes no great inconvenience and that the resulting simplification of the user-interface more than justifies it. However, the point is essentially trivial and should not obscure what follows.

2.1. Modular compilation facilities

A modular compilation system should assist the programmer by easing the development of software in 'natural' modules whilst providing the visibility rules and checking which prevent common errors.

A good modular compilation system should provide:

- separate compilation of declarations of any program object (declarations modules).
- separate compilation of nested closed clauses (cf [2]).
- a natural mechanism for compiling procedures to be called with parameters from outside the language environment.
- mechanisms to ensure that necessary recompilations of dependent modules are done after a change, and that unnecessary recompilations are avoided.
- mechanisms whereby multiple library preludes can be provided, in a way which allows the programmer to decide whether they are to be used separately or in some combination.
- no silly restrictions and no surprises.

The visibility of identifiers across module boundaries should be under the control of the implementor of the module containing their declarations. There must be no way in which the user of a precompiled module can gain access to identifiers which were not made public by the module's implementor.

2.2. The problems

Two problems commonly arise in the implementation of modules systems. The first concerns the scope of declarations in 'declarations modules' and the related subject of when two separate copies of a declarations module will exist in a program, and when two or more applications of a declarations module identifier actually identify the same instance of the module. Consider for example the following program fragments (written using the RS syntax):

```
DECS d1:
  INT a
  KEEP a
FINISH
```

```
DECS d2 USE d1:
  PROC p = INT:a;
  INT b
  KEEP p,b
FINISH
```

```
PROGRAM w USE d1,d2
BEGIN
  .....
  .....
END
FINISH
```

How many copies of d1 are invoked (and therefore how many INTs have been declared)? It is easy to invent examples where it is desired that the DECS module be shared: it is equally easy to invent examples where such sharing would be disastrous. The requirement is clear - both alternatives must be available, but the default must be the safe one. This means that DECS modules must only be shared if the programmer has explicitly requested that this be done.

That this is the correct decision can be seen clearly if the case of a library-prelude providing simulation facilities is considered. It is perfectly possible that such a prelude will use a random number generator for its own internal purposes, and that such a random number generator will itself be available as a library declarations module. A typical module might

look as follows:

```

DECS normal random:
REAL last random := 0.4919723;
PROC random = REAL: (..... last random..... ; last random := .....);
.....
KEEP last random, random, ... , ...
FINISH

```

This module has side-effects; it has internal memory.

Suppose now that the user of the library-prelude also has a need for a random number generator, and knows of the library version. It can be made available by a simple 'USE normal random' but if now the DECS module is shared (i.e. a single instance of 'last random' is used by prelude and program), the program is likely to behave incorrectly.

Advocates of module sharing will argue that the fault lies with the author of the DECS module: it should have been written so that the user was forced to declare a local 'last random', so that the module had no side-effects and could safely be shared. This is to miss the point. Firstly, it is the duty of a language designer to make language features error-resistant, if possible. Secondly, the DECS module above is the natural way to express the random number generator, leading to tidier programs in most cases. Certainly the NAG implementors were very resistant to the idea that the side-effect-free form should be used. Finally, there is a philosophical objection. Sharing modules in this way violates the information-hiding interface between the prelude and the user program, requiring that the user knows something about the implementation of the prelude, and increasing the coupling between the modules.

If the module has no side-effects and can safely be shared this is easy to determine at compile-time and the compiler should be required to perform the optimisation. If the module is intended to be shared, its name, or the identifiers to be used in common, should be passed explicitly through the module interface (or KEEPLIST). The scope of such declarations is then the same as if they had been included in the program at the point of the USE that invoked them; this is a safe rule and one that is easy to explain and remember.

2.3. Problem two - communication outside the language environment

The second problem concerns the case where the Algol 68 program is required to be a procedure which is called from another language. The Report does not define a syntax for the necessary 'program-with-parameters-returning-MOID', yet this is a frequent requirement once Algol 68 is used in earnest. It arises immediately if Algol 68 is used as a systems programming language in a system which is not wholly written in Algol 68. It also arises in applications programming wherever it is desirable to use Algol 68 for some routines but impracticable to write the whole application (or, particularly, the main program) in Algol 68. The problem is not entirely one to do with modular compilation, yet it is related both in the users' minds and in the detail of the necessary implementation.

The problems are not straightforward. Firstly there must be an acceptable syntax (the solutions adopted by RS implementations will be described later). Secondly there must be a mechanism for the prelude and postlude code to be elaborated at the appropriate time. (It is very embarrassing to have to find a way to write an Algol 68 procedure to do transport which avoids the standard files being opened and closed each time the procedure is called.) Thirdly, it becomes desirable to be able to generate objects which have a scope so global that they can survive control returning outside Algol 68 to the calling routine. Finally it should still be possible to write routines in a modular way, using a hierarchy of nested closed-clause modules. Some of these closed-clause modules may be user or library preludes, so that the externally visible parameter interface would not necessarily be associated with the outermost module. Possible solutions to this can be designed, but it is important that only one level in the hierarchy is allowed access to the parameter interface, and then only in one place, otherwise unsafe side effects may occur.

These problems have been solved in different ways by different implementors, but it would be useful if there were an agreed standard for future implementors to follow.

The next section of this paper describes the RS modular compilation system facilities and syntax. It will be seen that the solutions adopted to both these problems fail the requirements outlined above. A later section comments briefly on these failures and compares the solutions with the IFIP recommended scheme.

3. FACILITIES OF THE RS MODULAR COMPILATION SYSTEM

3.1. Introduction

The RS modular compilation system is designed to provide a powerful and secure method of program development.

Three types of module are provided; *declarations modules* which enable modes, procedures and other items to be declared and compiled in advance of their use in other modules; *closed clause modules* which may be complete programs or may be nested within another and have 'holes' where inner modules are later to be inserted; *composition modules* which assemble a number of closed clause modules together. Closed clause and composition modules are collectively known as *program modules*, in distinction to declarations modules which can never form a complete program on their own.

3.2. Keeplists

The programmer defines which indicators declared in one module are available to another using *keeplists*, and checks are made that these defined interfaces are adhered to. The keeplist is written as a sequence of indicators separated by commas. When an operator is included, the modes of its operands must also be specified in brackets after the operator name, in order to distinguish between different versions of the operator. For example,

```
MATRIX, * (REAL, MATRIX), m1, m2
```

3.3. Declarations modules

The simplest form of declarations module is

```
DECS decstitle:
  body
KEEP keeplist
FINISH
```

Here decstitle is some identifier to name the module, body (which is not enclosed by BEGIN and END) consists of Algol 68 declarations and other phrases useful for initialisation purposes, and keeplist is as described earlier. Indicators in the keeplist may be used by other closed clause or declarations modules; access to the keeplist of a declarations module is acquired by the USE clause so that the heading of a declarations module that uses one or more others becomes

DECS decstitle USE decstitlelist:

where decstitlelist names the other modules required, separated by commas.

There is only one instance of each declarations module and it may be used by any number of other modules. In order to free the user from having to consider the order of elaboration of a set of declarations modules and to prevent side-effects, the restriction is enforced that the outer level of a declarations module may not contain procedure calls or labels, nor use any references kept from another module.

3.4. Closed clause modules

A simple program will normally consist of a single closed clause module, possibly supported by one or more declarations modules, taking the form

```
PROGRAM progtitle USE decstitlelist
closed clause
FINISH
```

where progtitle is an identifier to name the module and the USE clause is only required if declarations modules are used.

More complex programs may be broken down into simpler components and written as a hierarchy of nested closed clause modules. The HERE clause (treated as a VOID unitary clause) is used to specify a 'hole' in a module and the indicators to be made available to the module filling the hole. It takes the form

```
HERE holename (keeplist)
```

where holename is some identifier to name the hole, and keeplist, as before, specifies the kept indicators.

If a module contains holes, their names must be listed in the module heading, and if the module is to be nested within another it must specify, through a CONTEXT clause, the name of the hole in which it is to fit, so that the general form of the closed clause module is

```
PROGRAM (holenamelist) progtitle1
CONTEXT holename IN progtitle2
USE decstitlelist
closed clause including HERE clauses
FINISH
```

3.5. Composition modules

A composition module contains no actual Algol 68 text of its own; it merely provides a specification for the assembly of a hierarchy of previously compiled closed clause modules. The form of a composition module is

```
PROGRAM progtitle
COMPOSE nest
FINISH
```

where progtitle is an identifier to name the composition module itself and nest specifies the modules to be assembled by pairing up formal holenames with actual modulenames as demonstrated in the following example.

Given a program module starting

```
PROGRAM (x1, x2) x
```

and a set of inner modules with the headings

```
PROGRAM a CONTEXT x1 IN x
PROGRAM (b1) b CONTEXT x2 IN x
PROGRAM c CONTEXT b1 IN b
```

then the following composition module would specify their assembly

```
PROGRAM comp
COMPOSE x (x1 = a, x2 = b(b1 = c))
FINISH
```

A composition module does not contain a CONTEXT clause; the context that applies to it is the one specified in its outermost closed clause module.

3.6. Partial composition

A composition need not fill all the holes in its constituent modules; it may leave some to be filled in a later composition. A composition module that contains unfilled holes is known as a *partial composition*, and is specified by pairing one or more of the constituent holenames not with an actual module name but with a new holename of its own, introduced by the symbol HERE. Thus, if we omit module c from the example in the previous section, we obtain the partial composition

```
PROGRAM (h) pcomp
COMPOSE x(x1 = a, x2 = b(b1 = HERE h))
FINISH
```

No explicit keeplist is written for a hole in a partial composition; the available indicators are defined to be all those kept en route from the outermost module to the symbol HERE in the composition so that in the above example the context 'h IN pcomp' provides all the indicators kept at 'x2 IN x' as well as those at 'b1 IN b'. This combination of keeplists is the main purpose of partial composition, effectively allowing a module to be compiled in several contexts simultaneously, a facility which is useful when a program must run inside several independent environmental packages such as might be provided for simulation or graph-plotting.

3.7. Declarations modules in a context

Declarations modules, like closed clause modules, may include a CONTEXT clause in their heading which will provide access to indicators kept at the specified hole. In order to use such a declarations module, the using module must also have access to those same kept indicators, and therefore the context specified by the using module must either be the same as that of the declarations module or be a dependent context resulting from partial composition (which as we have seen would supply the same kept indicators and more besides).

The context of a declarations module also determines its lifetime. For example, with the set of modules

```

PROGRAM (h) p1
BEGIN
  ...
  TO 20 DO HERE h(...) OD;
  ...
END
FINISH

DECS d1 CONTEXT h in p1:
INT i := 0
KEEP i
FINISH

DECS d2:
INT j:= 0
KEEP j
FINISH

```



```

PROGRAM p2 CONTEXT h IN p1 USE d1, d2
BEGIN
  print ((i += 1, j += 1))
END
FINISH

```

the value of *i* printed is always 1, whereas that of *j* will range from 1 to 20, since a new instance of module *d1* (but not of *d2*) is created each time the hole *h* is entered.

3.8. Preludes and the void context

Any closed clause or declarations module that has no explicit context specification is regarded as being compiled in the context of the standard prelude, which may be thought of as a closed clause module within which programs are automatically composed by the compiler.

It is clearly desirable that items from the standard prelude are available not only to the outermost level of a program but also to any nested modules. This idea has been generalised to arrive at the notion of a *prelude* as a truly outermost module with certain special properties which may be exploited in order to set up non-standard preludes for particular applications. These are

- (i) Items kept at a hole in a prelude and from any declarations modules at that context are available to *all* dependent modules to any depth of nesting.
- (ii) A closed clause module that specifies a prelude context may be composed within the prelude or within any dependent context.

A closed clause module is designated as a prelude by the special context specification `CONTEXT VOID`.

Declarations modules may also specify `CONTEXT VOID` in which case they may be used by *any* other module. This is the limiting case of the general rule given in 3.7 for the use of declarations modules in a context.

4. THE ICL 2900 IMPLEMENTATION: COMPILE-TIME

4.1. The RS compiling system

The RS compiling system is described in detail in [1]. Basically it may be regarded as shown in Diagram 1. The RS compiler itself, which is complet-

ely machine-independent, is conceptually surrounded by a 'shell' which provides the necessary interfaces to the host system, for example procedures to read lines of Algol 68 source, to output error messages and, more important to the current discussion, to obtain information about already compiled modules. These 'shell' procedures, which are actually passed as parameters to the RS compiler, must be provided by each separate implementor of an RS system, in addition to the machine-dependent 'translator' which is responsible for converting the 'stream language' output by the RS compiler into object code for a particular machine. (Further details of the 'stream language' may be found in [1]).

The 'shell' interfaces concerned with modular compilation are

- give module details - provides the RS compiler with information about the properties of some specified module
- give spec - provides the RS compiler with information about a keeplist

The stream language, passed from the RS compiler to the translator via the procedure 'output', contains elements which provide information pertinent to modules, including both information about the current compilation (such

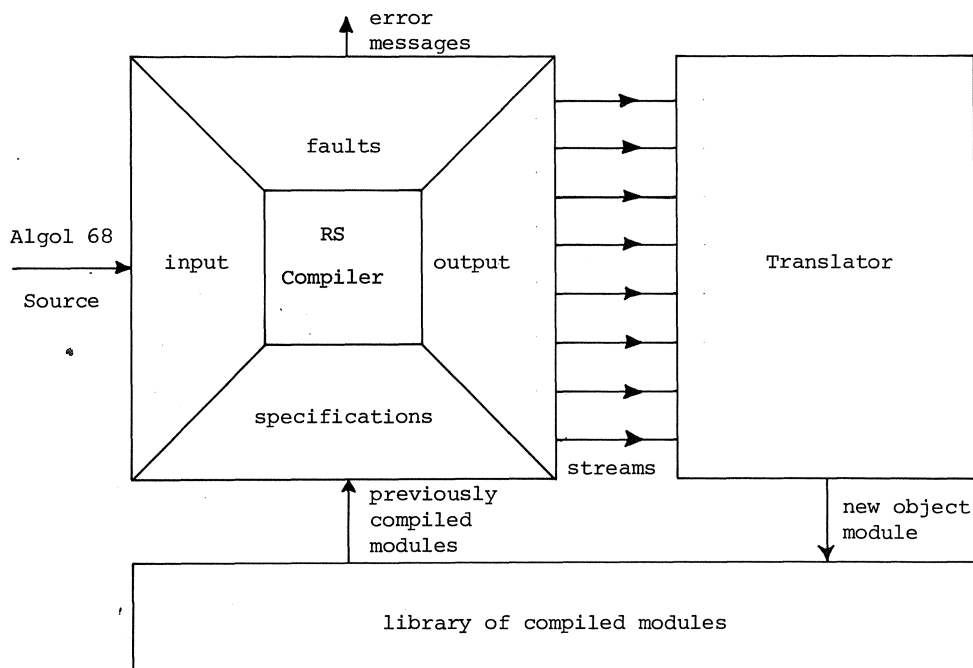


Diagram 1 - The RS compiling system

as the name and type of the current module) and information about other modules, (such as details of available kept identifiers).

4.2. Storing module information

The facilities of the RS modular compilation scheme clearly imply that when a module is compiled, the object file produced must contain not only the instructions to be obeyed at run-time but also information to allow the module to be utilised during the compilation of some subsequent module. For example, the object file associated with a declarations module must contain, in some form or another, a description of the module's keeplist so that when that module is named in a USE statement the compiler can determine what items are included in the keeplist and check that the second module is using them correctly.

In terms of OMF, the 2900 Object Module Format, the logical place to store this sort of information is in the 'diagnostic data records' which are held at the end of an object file.

The compiler writer is completely free to decide the format of these records, their primary purpose being to provide information for a post-mortem report in the event of a run-time error. Since the diagnostic data records are not accessed by the 2900 loader, additional information that is required only at compile-time may be stored there without introducing any overhead on program loading.

Information used at run-time by the modular compilation system is stored in the Procedure Linkage Table (PLT) area of each module. This area is used in the standard way to hold inter- and intra-module linkage information, for example external references which are 'fixed up' by the loader to virtual addresses when the module is loaded.

4.3. Information written to object module

When a module is compiled, the following information is written to the diagnostic data records of the object file created, for use during the subsequent compilation of other modules:

- (i) a description of the properties of the module as a whole, including
 - the name of the module
 - whether the module is a declarations module or a program module, and, if the latter, the number of holes it contains

- the name of the surrounding context of the module (both holename and modulename)
 - the name of the prelude context of the module, if different from the above.
- (ii) a specification of each of the keeplists provided by the module; for a declarations module there will be only one of these, for a program module one for each hole in the module. Except for a keeplist associated with a hole in a partial composition, the specification is a coded representation of the identifiers, modes, etc, in the keeplist. The translator can treat this as a single character string since the RS compiler is responsible for the encoding and decoding of this information. For the partial composition case, where the actual keeplist available is the concatenation of several individual keeplists (see 3.6 above) the names of the contributing contexts are instead recorded as the specification of the composite keeplist.

The details necessary to construct the above information are obtained by the translator from elements of the stream language output by the RS compiler.

A unique name is also generated for each keeplist specification and for the module itself. These names are also written to the information in the object file to provide a 'time-stamping' facility for compatibility checking. (See 4.5 below.)

4.4. Use of module information

The module information stored in the object file is utilised as follows:

- (a) When the RS compiler encounters a CONTEXT statement in the source program, it calls the shell procedures 'give module details' and 'give spec' which prompt the shell to search the filestore for the named program module and to extract the specification of the keeplist associated with the named hole. The objects in the keeplist are then determined and effectively treated as though they had been declared in the current compilation. If the keeplist is a composite one (i.e. the context is a hole in a partial composition) then the process is invoked recursively for each contributing context. The name of the prelude context to be associated with the current compilation is obtained from the prelude context recorded in the module named in the CONTEXT statement, and its specification is also found and processed in a similar way.

- (b) Similarly, a USE statement causes the compiler to locate (via the shell) the named declarations module, extract its keeplist specification and 'declare' the associated identifiers, etc. The context of the declarations module is checked to be consistent with the rules described earlier (3.7).
- (c) When a composition module is compiled, all the modules named are located and their context specifications checked to ensure that they can be put together in the specified way. The context information pertaining to the outermost module is carried across to the resultant composition module.

4.5. Unique names for compatibility checking

To ensure complete compatibility of modules, a unique name is generated for each keeplist in a module and also for the module itself. The names are constructed from the time of day down to a resolution of 128 microseconds, which is considered to be sufficiently 'unique' for this purpose. On successful compilation of a module, the translator searches for a previous version of that module and, if one can be found, compares its interfaces (keeplists, context, etc) with those of the new module. If a keeplist is identical in the two modules (i.e. contains the same identifiers, with the same modes, in the same order), then the unique name of the old keeplist is carried over to the new one, a new unique name being created otherwise (or of course if no previous module was found). The unique name of the module itself is only carried forward if *all* its interfaces (including contexts, number of holes, etc.) are identical with the old version.

The purpose of this scheme is to ensure that when a module is recompiled and any of the external interfaces to the module have been changed, then any previously compiled modules dependent on those interfaces must also be recompiled. Conversely, if a module is recompiled without changing its interfaces, there is no need to recompile any dependent modules.

The RS compiler performs certain compatibility checks, for example that the context of a used declarations module is compatible with that of the using module, or that the modules named in a composition can be assembled in the specified way. However, these checks use only the names as they appear in the source program; the translator additionally checks the corresponding unique names to ensure complete compatibility.

Run-time compatibility is also ensured by the use of module unique names. The unique name is made an alias of the object file and all external references from one module to another are via the unique name rather than the user's name for the module. This means that if a module has been changed incompatibly and recompiled then any dependent modules cannot pick up the incompatible version until they too have been recompiled.

4.6. Access to the standard prelude

The standard prelude for the 2900 Algol 68 system is itself, like most of the product, written in Algol 68 and uses the modular compilation system. It consists of two parts

- (i) A closed clause module with a single hole, this hole being the place where the user program is to be called. The module declares and opens files `standin`, `standout` and `standback` and declares the procedures that operate on these files (`read`, `print`, etc). It also initialises the 2900 diagnostic subsystem to allow run-time errors to be handled properly.
- (ii) A set of declarations modules which provide declarations of all the other items of the Algol 68 standard prelude (other transport procedures, mathematical functions, environment enquiries, etc). The modules are all at `CONTEXT VOID` so that any other module, regardless of its own context, may access them. Only those modules that are actually required by a program need be loaded when the program is run.

As part of the construction of the complete compilation system, the keep-lists and module information of the standard prelude modules are processed by a special utility program to build up a lookup table of the items they contain. This table is utilised by the compiler directly (via the shell) so that there is no need to access the actual modules at compile time; the items effectively become 'built-in' to the compiler. The utility program used for creating the lookup table is also issued to installation management thus providing the facility to extend or replace the standard prelude with an 'installation prelude'.

4.7. Automatic composition

In order to cut down the work required by the user, the translator will under certain circumstances perform an automatic composition of the module being compiled. In particular, the process is applied to automati-

cally compose simple programs within the standard prelude.

Automatic composition is only possible if the module is a program module that contains no holes and each of its successively enclosing contexts out to the outermost level (CONTEXT VOID) is derived from a module with exactly one hole. In other words, the module must be the last (innermost) in a complete set of modules, each nested one within another. In such cases, the modules required for the composition can be determined unambiguously by the translator.

The process operates by finding each successively enclosing context until the outermost level is reached. At each stage a check is made that the module concerned has only one hole and, by checking unique names, that it is compatible with the module filling the hole. The process is abandoned if any of these checks fail or if any of the modules cannot be found.

4.8. Separate compilation of procedures

The RS implementors' documentation does not currently define a syntax for the compilation of Algol 68 programs which are callable as procedures with parameters passed from outside Algol 68. As a result two different solutions have been adopted.

Currie, in his implementation for the FLEX machine, has made the parameter interface to procedures KEPT by DECS modules visible to the operating system. This solution relies on the nature of the FLEX system and in particular on the abolition of all scope restrictions on the FLEX machine [3].

The ICL 2900 and Honeywell Multics compilers have adopted the solution that where a program consists simply of BEGIN routine-text END then the routine will be entered directly at run-time. Thus a separate compilation of

```
proc p = (real a) real: (.....)
```

would be achieved by

```
PROGRAM p (
  (REAL a) REAL: (.....) )
FINISH
```

A CONTEXT and USE-list may be specified as usual.

Note that LINDSEY & BOOM [2] do not address this requirement.

4.9. Module Sharing

As explained in 3.7 above, the scope of a DECS module is determined by its CONTEXT. This means that within any CONTEXT only one copy of a DECS module will ever exist and all USEs for this module will identify the same instance. As will be explained later, this involves action at run-time to ensure that DECS module bodies are elaborated at the correct point in the module hierarchy.

The LINDSEY & BOOM scheme [2] shares modules whenever it can be detected statically that this is possible. In most, but not all, cases this will lead to the same result as in the 2900 compiler implementation. Unfortunately (in the view of one of us) this creates exactly the unsafe sharing and the unnatural programming style deprecated earlier.

5. RUN-TIME ASPECTS

5.1. The Module Controller

The control of run-time aspects of the modular compilation system is carried out by a special module known as the Module Controller.

The semantics of declarations modules, in particular the rules concerning the lifetime of declarations modules in a context, imply that, at least in a stackbased system, the body of a declarations module must be elaborated at the time that its context is created, which is not necessarily the point at which it is first used by another module. Consequently, one of the main functions of the Module Controller is to ensure that declarations modules are elaborated at the right time and in the right order.

The information required by the Module Controller is obtained from the Procedure Linkage Table (PLT) area of each of the modules to be run. These areas contain external references from one module to another and their structure enables the relationship of modules in a composition and the declarations modules used to be determined. All runnable programs contain as part of their entry sequence a call to the Module Controller, making available to it the PLT of the top-level composition, now of course loaded into virtual store and with all external references resolved.

The PLT reflects the structure of the hierarchy of modules to be run, and is essentially a tree structure starting from an object of mode COMP, where


```

MODE COMP      = STRUCT (REF CCMODULE body, REF [ ] COMP holes),
CCMODULE      = STRUCT (CCODE code, REF [ ] USED decs),
USED          = STRUCT (REF DECSMODULE d, INT level),
DECSMODULE    = STRUCT (DCODE code, REF [ ] USED decs);

```

The modes CCODE and DCODE represent the code to be executed for a closed clause module and declarations module respectively. The mode USED provides information about declarations modules used, including the level of their context relative to the context of the using module.

The Module Controller traverses this structure and creates a new structure in which the declarations modules, instead of being attached to the modules which use them, are associated with the holes defining their contexts. CONTEXT VOID declarations modules are treated as a special case and added to a separate list. The revised structure can be regarded as an object of mode NEWCOMP, where

```

MODE NEWCOMP   = STRUCT (REF CCODE code, REF [ ] HOLE holes),
HOLE           = STRUCT (REF NEWCOMP body, REF [ ] DCODE decs);

```

The declarations modules at each context (represented in the 'decs' field of a HOLE) are sorted so that if any module uses another at the same context, then the latter precedes the former in the list, thus reflecting the order in which the modules must be elaborated. The case of mutually recursive declarations modules (an error) is also detected at this stage.

In fact the actual data structures involved are more complicated than those described here (which take no account of, for example, partial composition) but the description is sufficient to give an idea of the process involved.

5.2. Running the modules

After constructing the revised data structure described above, the Module Controller performs the execution of the complete program by traversing this structure, calling each module at the appropriate place. First of all the CONTEXT VOID declarations modules are elaborated, followed by the outermost closed clause module (of necessity CONTEXT VOID and normally the standard prelude module). Each time a hole is reached, a procedure in the Module Controller is called to elaborate declarations modules at the new context and then call the closed clause module filling the hole.

5.3. Use of the stack

The 2900 Series has a stack-based architecture and the standard procedure call and exit mechanism assumes the use of separate 'stackframes' for each procedure, with link information and procedure parameters in defined locations at the base of each frame. The current stackframe is defined by two dedicated registers Local Name Base (LNB) and Stack Front (SF), and on normal procedure exit these are reset to their previous values, effectively deleting the newer stackframe and reverting to the older one.

The 2900 Algol 68 implementation uses this mechanism not only for procedure and operator calls but also for calls to external modules. However, the normal exit mechanism cannot be used with declarations modules since it is necessary to retain their outermost stackframe. (Note that it is not sufficient merely to retain the items in the keeplist because kept procedures may require access to local data which is not kept.) The method adopted is to explicitly reset LNB and jump to the return address *without* resetting SF. This means that the stackframe of the declarations module becomes absorbed in that of the procedure (in the Module Controller) that invoked it. This special exit applies only of course when the module itself is elaborated, and not on calls to kept procedures.

5.4. Access to keeplists

The items made available to a module from other modules must necessarily originate in keeplists. Each keeplist may be either from a declarations module or from a module contributing to the local or prelude context.

The items from each of the keeplists available to a module are always copied into its own outer stackframe so as to make access to those items more efficient (this being particularly desirable for standard prelude items). The module obtains the address of each keeplist (the items in which are always in consecutive stack locations) through pointers set up in its own stackframe by the Module Controller, effectively as parameters to the module.

The Module Controller maintains dynamically, for each existing context, a table of pointers to the keeplists associated with that context, viz. the keeplist of the context itself and the keeplists of the declarations modules at the context. The tables for successively nested contexts are chained together so that composite keeplists resulting from partial compositions may be found. When a hole in a closed clause module is reached, its keeplist

is constructed on the stack and a procedure in the Module Controller is entered, passing as parameters a pointer to the keeplist and the internal number associated with the hole. This procedure creates a new keeplist table which is added to the end of the current chain and initially contains only the pointer to the hole keeplist. The declarations modules at the new context are then elaborated in turn, each returning a pointer to its keeplist which is added to the table, and finally the closed clause module filling the hole is itself elaborated. Each module called is given pointers to the new table (defining its local context) and to the table defining its prelude context. It is also given an index table to enable it to extract only the keeplists of those declarations modules that it actually requires.

6. EXAMPLE OF A SET OF RS MODULES

It is instructive to take an example set of RS modules and consider what happens both at compile-time and run-time. Take, for example, a set of modules with headings

- (a) DECS d1 :
- (b) DECS d2 USE d1 :
- (c) PROGRAM (h1, h2) p USE d1
- (d) PROGRAM p1 CONTEXT h1 IN p USE d2
- (e) DECS d3 CONTEXT h2 IN p :
- (f) PROGRAM p2 CONTEXT h2 IN p USE d2, d3
- (g) PROGRAM c COMPOSE p(h1 = p1, h2 = p2)

The modules may be compiled in the order shown, but the required order is not completely fixed. For example, it is immaterial which of the modules p or d2 is compiled first, though both must be compiled before p1.

During each compilation, the compiler examines the specifications of the previously compiled modules named in the CONTEXT and USE clauses. For example, when compiling p1, the specifications of modules d2 and p, and of hole h1, will be read. Finally, when the composition c is compiled, modules p, p1 and p2 are examined and found suitable for combination in the specified way. The standard prelude is also included automatically as the outermost module in the composition.

When the program is to be run the Module Controller (initially entered from module c) examines the relationship between each of the modules, as recorded in their PLTs. The declarations modules are then associated with

their correct levels; for example, module d2, instead of being attached to modules p1 and p2, is 'floated up' to the hole defining its context, in this case the default context corresponding to the hole in the standard prelude.

The Module Controller then proceeds to elaborate the modules as required. Firstly, the standard prelude is entered, and the standard files are opened. At the point corresponding to the hole for the user program, the Module Controller is entered again. Modules d1 and d2 are then elaborated (in that order), each module containing a non-standard exit to preserve its stackframe, before module p is called. A similar process occurs at each of the holes in p, so that at hole h2, module d3 is elaborated followed by p2. Each module called is given a set of pointers to enable it to locate the keeplists it requires.

On exit from each closed clause module, control passes back through the Module Controller to the surrounding closed clause module, until eventually a return is made to the standard prelude module which closes the standard files and returns to JCL level.

7. POSTLUDE

It will be seen that the facilities provided by the RS system are broadly equivalent to those proposed by LINDSEY & BOOM. We very much regret that the definition [2] was not available to the implementors of the RS compiler in time to be adopted; however, since compatibility between RS implementations is seen as most important, future RS implementations will certainly keep to the RS syntax.

Nevertheless the ICL 2900 compiler implementation shows that modular and separate compilation facilities similar to those proposed by LINDSEY & BOOM can be implemented quite efficiently in a production-quality compiler.

The problems of module sharing and "programs with parameters" remain. It is to be hoped that the considerable effort going into Ada design and implementation will lead to further developments in both these areas, and that the designers of the next generation of Algorithmic Languages adopt a consistent set of secure and convenient solutions.

ACKNOWLEDGEMENTS

The RS compilation system was designed by I. Currie and J. Morison at RSRE, Malvern, England.

The ICL 2900 RS implementation was designed by M. Austin, G. Finnie, R.L. Hutchings, J. Rees, E.W. Taylor and M.C. Thomas at Oxford University and SWURCC, England.

Everything written about the RS compiler owes an immeasurable debt to S.G. Bond and P.M. Woodward, for the influence of their experience and instincts on the design, and particularly for the clarity and elegance of their documentation and explanations.

REFERENCES

- [1] BOND, S.G. & P.M. WOODWARD, *Introduction to the RS portable ALGOL 68 compiler*, (RSRE Technical Note 802).
- [2] LINDSEY, C.H. & H.J. BOOM, *A Modules and Separate Compilation Facility for ALGOL 68*, ALGOL Bulletin 43.
- [3] CURRIE, I., *Personal Communication*, RSRE Malvern UK.

PROGRAMMING LANGUAGES FOR A COURSE IN DATA STRUCTURES

V.J. RAYWARD - SMITH

ABSTRACT

The case is made for using Algol 68 as a core language in the teaching of a post foundation data structures course.

1. INTRODUCTION

Most universities that offer a degree in Computing run a post foundation course on data structures. The precise content of this course may vary but it usually includes all the material in CS2 and CS7 of Curriculum '78 [8]. When teaching the course it is important to introduce to the student various facilities in diverse programming languages to enable him to understand and manipulate all types of data structure. The teacher of such a course is faced with a dilemma; too many new programming languages confuse the student and yet many data structures appear to demand specialist treatment (e.g. ICON [14] or SNOBOL [15] for string processing, LISP [39] for list processing). In this paper, a case is made for using Algol 68 [38] for the main programming language for the course and restricting these specialist languages to an appreciation.

It is generally agreed that any programming language used in the teaching of data structures or algorithm design and analysis must support structured programming [10] and recursion. This is certainly supported by looking at those textbooks published in the last decade which might be used as course material (see Table 1). Many of these claim that since no suitable language exists for the manipulation of all data structures, the best option is to use a specially designed Algol-like language. However, if a real programming language is used as the core language for the course then the advantages are obvious. Any such language must be supported by good compilers with clear error diagnostics together with adequate literature suitable for student use. For these reasons we can sadly discount both ADA [22,33] and Simula 67 [3]. The main contestants for the core language appear to be PL/I [1], Algol 68 [38] and Pascal [23]. Each of these can claim to have been derived to a greater or lesser extent from Algol 60 and each is block structured, has high level control structures including recursion and has typed data. A considerable amount of literature exists which compares these and other programming languages in general [2,27,28,31,32,36,37]. The purpose of this paper is to focus attention on data structures and the suitability of a language for the teaching thereof.

Without doubt, Pascal is the easiest of the three languages mentioned above to learn while both PL/I and Algol 68 require more effort even if only because of their increased range of facilities. Pascal also has the advantage of being particularly easy to compile and is currently available on cheap

Table 1

A selection of books on data structures published in the last decade.

<u>Author</u>	<u>Publisher</u>	<u>Title</u>	<u>Year of Publication</u>	<u>Main programming language used in text</u>
A.T. Berztiss	Academic Press	Data structures: theory and practice	1975	FORTRAN
M. Elson	Science Research Associates	Data structures	1975	A high-level hypothetical programming language* (+ LISP, SNOBOL)
C.C. Gotlieb & L.R. Gotlieb	Prentice Hall	Data types and structures	1978	Algol-like
M.C. Harrison	Scott, Foresman and Company	Data structures and programming	1973	FORTRAN and PL/I
E. Horowitz & S. Sahni	Computer Science Press	Fundamentals of data structures	1976	Algol-like
H.A. Maurer	Prentice-Hall	Data structures and programming techniques	1977	PL/I
J.L. Pfaltz	McGraw-Hill	Computer data structures	1978	Algol-like
M. Shave	McGraw-Hill	Data structures	1975	Algol W
J.P. Tremblay & P.G. Sorenson	McGraw-Hill	An introduction to data structures with applications	1976	PL/I

* unfortunately the rather limited control structures in the language result in a proliferation of the infernal GOTO.

micro-processors such as Apple II. Thus it is likely to become even more widely adopted than it is at present and is already being used in British schools and technical colleges as well as in universities. The simplicity of Pascal also makes for good error diagnostics and thus it is not surprising that the language is becoming widely accepted as an introductory programming language both in Europe and USA. At the post foundation level, one can expect students to have already acquired a proficiency in at least one programming language. With increasing probability, this programming language will be Pascal. At UEA, we have found that we can successfully convert such students to be reasonably proficient programmers in elementary Algol 68 in just six contact hours. This can only be achieved if the teachers concerned do not fall into the pitfall of teaching advanced facilities. For example, a teacher might expect to cover the material in the first five chapters of McGettrick's

excellent book [26] although much of chapter four is probably best delayed until the course moves on to a discussion of arrays and record structures. Alternative introductory texts are [5,6]. A similar strategy could be expected to work introducing students to PL/I using any of a much greater selection of introductory texts [7,11,21, to name but a few] but more time would probably be required partly because of the difficulty of learning all the default mechanisms of the language. If a student's first programming language does not support structured programming then conversion to Algol 68 or PL/I can be expected to be more traumatic and to take longer. Even introducing Pascal may then be a difficult task, although at UEA we have found that students of moderate ability who have only used FORTRAN and COBOL still find Pascal an easy language. A student with the undoubted benefit of the knowledge of Algol 68 or PL/I can be expected to acquire Pascal from one of the introductory texts [e.g. 12,35] with little more than bedtime reading.

One of the key concepts to be included either explicitly or implicitly in a data structures course is that of extensibility. As Brian Meek defines it [27], a truly extensible language would allow the programmer means within the syntax to define his own keywords, his own meanings for them and his own language constructs. At a lower level, extensibility implies that the core facilities of the language are general enough to provide means of designing special facilities for use in particular applications. Algol 68 meets this need by providing the programmer facilities for defining his own modes and operators. In a data structures course such facilities are invaluable (see, for example, sections 2.3 and 2.6 of this paper). PL/I does not offer such facilities but rather tries to provide everything the programmer might need. Not only is this approach bound to fail but it also makes the language too large and overly difficult to master.

2. DATA STRUCTURES

In this section, the facilities to construct and manipulate some examples of data structures in the three languages, PL/I, Algol 68 and Pascal, are discussed and compared. Particular emphasis is placed on fundamental differences which will affect teaching.

2.1. Atoms

A complex data structure can be viewed as being composed of simpler data structures which in their turn may be further composed of even simpler structures. Eventually, however, without going into machine representation, the constituent parts can be broken down no further and these parts are called atoms. Atoms correspond to Algol 68 objects of primitive mode bool, int, real and char. Any programming language offers basic facilities for creating, storing and manipulating atomic information but even at this level, the three languages under consideration differ.

The main difference between PL/I and Algol 68 is the inclusion in PL/I of attributes for arithmetic data items other than modes. For example, an object of mode REAL has to also have its base, scale and precision either specified or assumed by default. Boolean values and character values also differ in that in PL/I a Boolean is not a primitive object being regarded rather as a BIT string of length 1 and similarly a simple character is regarded as a CHARACTER string of length 1.

Pascal, like Algol 68, is a typed language and makes clear distinctions between constants with read-only access and variables with both read and write access. The atoms of Pascal include objects of type Boolean, integer, real and char but also there is a facility whereby the programmer can define his own unstructured type called a *declared scalar type*. For example,

```
type days = (mon,tues,wed,thurs,fri,sat,sun);
var d : days
```

declares a variable d which can refer to any of the values mon through to sun. Boolean is also regarded as a scalar type by defining

```
type boolean = (false,true)
```

Scalar types in Pascal can be defined as above by enumerating all the possible values or a type can be defined as a subrange of any other already defined scalar type, e.g.

```
type workday = mon .. fri
```

defines workday as a subrange of days. Subranges of integers and characters can also be used but not of reals. As pointed out in [4], scalar types can be handled in any programming language by suitable encoding. Moreover, given suitable ascription, the coding can look quite similar to that of Pascal. To illustrate this claim, the days example might be written in Algol 68 as

```

mode days = int;
days mon = 1,tues = 2,wed =3,thurs = 4,fri = 5,
sat = 6,sun = 7;
days d

```

This is clearly still not as good as the Pascal version since Pascal offers both better protection for the programmer and the potential for better storage utilization as well as the closing of the gap between data structures and the real world objects which they represent. Proposals for extending Algol 68 to include facilities for manipulating scalar types on lines similar to that proposed by Hoare [16] have been made in [4].

2.2. Arrays

The facilities for creating and manipulating arrays in Algol 68 and PL/I are remarkably similar. Both languages allow arrays of arbitrary dimension and elements which may be non-atomic. Moreover, the bounds of the array, each determined by an expression whose value is available at allocation time, may take negative as well as positive integer values. Elements of arrays are selected using subscripts in the usual way although in Algol 68 they are enclosed in square brackets while in PL/I round brackets are used so that the syntactic form of an array reference in PL/I is identical with that of a function call. Both languages permit a subarray of a multidimensional array to be obtained by fixing some subscripts. For example, if A is a two-dimensional array then in PL/I, the cross section A(3,*) would denote the third row while in Algol 68 this would be denoted by the slice A[3,]. In PL/I, however, the cross section cannot itself be subscripted but in Algol 68 the slice can. Algol 68 also offers the trimming facility for extracting a subarray; no such facility exists in PL/I. PL/I and Algol 68 both allow assignment of arrays which may include cross sections/slices providing the arrays on the left and right side of the assignment have the same shape and size. PL/I will also accept array names as components in a general expression on the right-hand side of an array assignment. For example, if A,B are two real vectors of the same shape and size, then the PL/I assignment $A = 2*B$ will result in each a_i being assigned the value of $2*b_i$. Although Algol 68 does not automatically provide such overloading facilities, an operator can be defined in Algol 68 as having differing effects dependent on the mode of its operands. Thus, in this example, the operator $*$ could be defined in Algol 68 as follows [see 25].

```

op * = (real r, []real u) [] real:
  † the product of a scalar and a vector †
  (int m = lwb u, n = upb u;
  loc [m:n] real ru;
  for i from m to n do ru [i]:= r * u [i] od;
  ru)

```

When teaching about arrays, it is necessary to talk about headers and to illustrate how headers are created and altered as a result of particular pieces of code. In the course given at UEA [20], diagrams are used not unlike the abstract model of arrays described in [2].

In Pascal, facilities for handling arrays are rather basic. The major criticism is that the bounds of the array must be known at compile time which is certainly a retrograde step being reminiscent of FORTRAN and included for ease of compilation rather than for programmer convenience. Like PL/I, the array must have a fixed number of components there being no equivalent of the Algol 68 flex facility. More seriously, there are no facilities for slicing or trimming arrays although as with Algol 68 and PL/I, arrays can be used in assignment statements. The one advantage Pascal has over Algol 68 and PL/I is that the index does not have to be integer-valued. An array can be indexed by any scalar type. Thus, the following is a valid Pascal declaration and assignment.

```

sick: array [days] of boolean;
sick[mon]:= true

```

Pascal also offers limited facilities for handling packed arrays which means the compiler will economize storage requirements at the expense of additional execution time. These are more general than those provided in Algol 68 through the bits and bytes modes.

2.3. Sets

The dominant concept in mathematics is that of a set yet in programming languages facilities for directly representing sets are rarely offered. Many algorithms, especially in combinatorial applications, are expressed in terms of sets so to some programmers this is a real handicap. The usual technique to overcome this lack of means of direct representation is the use of vectors of bits to represent sets.

Pascal is one of the few languages which does offer facilities for

directly representing sets. Sets can be defined by either enumerating the set elements or by constructing new sets from other sets using operators + (union), * (intersection) or - (set difference). Relational operators applicable to set operands in Pascal enable tests for (in)equality and set inclusion. The operator in is used to test for set membership. Although Pascal does not specify a maximum cardinality for the sets it manipulates, implementors of the language have imposed such a limit. Clearly, it is required that all basic set operations are relatively fast and the best way of achieving this is to represent sets by vectors of bits. The vectors are usually some fixed multiple of the wordlength which dictates an upper bound on the cardinality of the set. If the set is larger than this then the set is best represented as an array of sets.

The drawback in representing sets as vectors of bits is that the universe must be fixed so that the position of a bit representing a given element in the vector can be determined. Alternative techniques which may be more suitable according to the particular application are representing sets as linearly linked structures or by using hash tables. It is relatively easy (and a good student exercise) to design and implement an Algol 68 library prelude for set manipulation. The library prelude can be based on any representation of sets and could even use more than one. Once the library prelude has been established, the Algol 68 programmer has all the necessary facilities for manipulating sets and need not himself trouble about the method of representation. No such protection can be offered for the poor PL/I programmer.

2.4. Record structures

Facilities for creating and manipulating record structures are fairly similar in all three of the languages under discussion. In this section, attention is focused on structures which do not include pointers, discussion of that topic being delayed until section 2.7.

Both Algol 68 and Pascal essentially view a record structure as a one-level concept but allow substructures to any depth. PL/I, on the one hand, defines a record structure in a way similar to COBOL's data division by numbering the individual levels, e.g. by prefixing 1 to the structure

name, 2 to variables at the next level, etc. Thus, the record structure described by the tree in Figure 1 may be defined in Algol 68 by the declaration:

```

struct (struct ([1:2]char title,initials,
               [1:10]char family) name,
       [1:3]char department) employee

```

and in Pascal by a similar type declaration. In PL/I, the same structure would be defined using the DECLARE statement:

```

DECLARE 1 EMPLOYEE,
       2 NAME,
       3 TITLE CHAR(2),
       3 INITIALS CHAR(2),
       3 FAMILY CHAR(10),
       2 DEPARTMENT CHAR(3);

```

PL/I and Pascal select the fields using a top-down method while Algol 68 (like COBOL) uses a bottom up

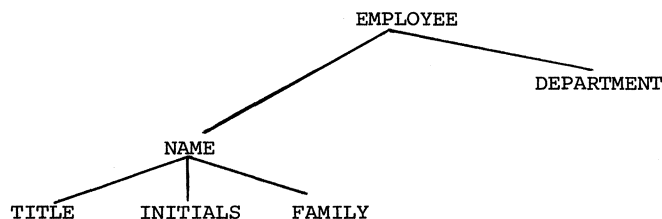


Figure 1

Thus, in PL/I, the title of the employee is retrieved using EMPLOYEE.NAME.TITLE while in Algol 68, title of name of employee is used. In PL/I, if no ambiguities arise, a substructure can be referenced using just the substructure's name and not using the qualified form. Hence, in our example, simply TITLE may be sufficient. Unfortunately, Algol 68 insists on the fully qualified form and this can result in untidy and unnecessarily lengthy code. Pascal overcomes the problem using a with statement. Within the component statement of a with, one can denote a field of the record variable appearing after the with using only its field identifier. In all these languages, one can assign to and

compute with the fields of a structure and also manipulate structures in their entirety. Also, unlike COBOL, all three languages allow arrays of structures.

In some circumstances, records may vary in structure even though they are said to be of the same type. For example, a record describing an employee may include a field CAR if and only if the employee has a certain DEPARTMENT. In Pascal, such a record type would be specified as consisting of several *variants*. This means that different variables, although said to be of the same type, may assume structures which differ in either the number and/or types of components. In Algol 68, this can be achieved using union but no such facility exists in PL/I.

2.5. Files

The student on a data structures course will need to have experience of both sequential and direct access files and to have a general understanding of peripheral devices and the means by which programs can obtain access to such devices. Most students reading for a computer science degree will also be undertaking a course on data processing where this particular topic will be treated in far more detail. Nevertheless, it is dangerous to omit files as they are an important type of data structure and cannot be omitted from a course purporting to study that topic.

In Algol 68, the central concept is that of a book which consists of a sequence of characters organised into lines which in turn are organised into pages. Any book has a logical end and an identification label to distinguish it from other books. A book may be referred to within a program by a field of a structured value with special mode file. The field selectors for file, however, are not available to the programmer who is therefore given a set of predefined procedures for their manipulation. The program accesses a book by means of a channel which may limit the number of pages, lines and characters of the book. Initially every particular program is provided with one book to be read via stand in channel, one to be written to via stand out channel and one to be used as a standard backup via stand back channel. These are opened in the standard prelude but the programmer can open further files and other channels as he wishes. Books can be accessed either sequentially or using random access. Moreover, the file may be compressible, which means pages may have a variable number of lines and lines a variable number of characters. Transput may be either as characters or in binary form

and can be formatted or not. Algol 68 thus has a number of basic facilities for manipulating external files which together with a suitable library prelude can make it an attractive language even for commercial data processing. If peripheral devices are not to be used, then internal books are used. These are of mode ref [] char, ref [[]] char or ref [][][] char. For example, [5] [50] [120] char bk; declares a book, bk, of five pages each of fifty lines of one hundred and twenty characters. Internal books are only used for character transport and are not compressible.

The Algol 68 programmer is always clear whether a book to which he is referring is or is not stored on an external device. The unfortunate Pascal programmer does not know how the files are associated with secondary storage and peripherals, the details of which are completely implementation dependent. A file in Pascal simply specifies a structure consisting of a sequence of components all of which are of the same type. Thus,

```
type f = file of integer
```

specifies a file of integers. Associated with the file variable, f, is a buffer variable f↑ of the component type, in this case integer. f↑ can be considered a "window" through which one can inspect existing components or append new components via the basic file handling operators of reset, rewrite, get, put, read and write. There are two standard files, input and output. If read and write are used without indication of a file parameter then, by default, files input and output are, respectively, assumed. These files and also any others existing outside the program must be passed as parameters in the program heading. The only facilities offered in pure Pascal for manipulating segmented files are those for textfiles, i.e. files whose components are characters. Such files are divided into lines using special textfile operators. In Pascal 6000-3.4 [23] facilities exist for manipulating more general segmented files.

Like Algol 68, PL/I has extensive facilities for reading and writing data. The Data files of PL/I are one of three types - input, which supplies data to the program, output, which stores the result of the program or update which, to a certain extent, does both. There are also two distinct contexts in which the data files are used: STREAM I/O where data are considered to form a continuous sequence (or stream) of individual values and RECORD I/O where the data are considered to be a collection of records. Under STREAM I/O, it is the individual value rather than the individual record which is the fundamental unit of input or output. STREAM I/O

is essentially sequential but RECORD I/O also allows indexed files and varieties of direct access files including those with hardware keys.

In PL/I, there are three types of STREAM I/O called LIST-directed, EDIT-directed and DATA-directed. The LIST-directed techniques are used for sequences of data values in free format, separated from each other by blanks. They correspond to Algol 68 formatless transput. Algol 68 formatted transput is provided by EDIT-directed I/O where the data is a sequence of records identically formatted. DATA-directed format has no direct Algol 68 counterpart and is available merely to save the necessity of unnecessary formatting. In this case, the data consists of a stream of items of the form <name> = <value>.

In RECORD I/O, there are two different modes of accessing files: SEQUENTIAL input and output, and DIRECT input, output and update. The PL/I file declaration of the form

```

DECLARE filename FILE RECORD { SEQUENTIAL } { INPUT }
                             { DIRECT   } { OUTPUT }
                                         { UPDATE }

```

is used to define a file's attributes. Constructs are provided for processing INPUT and OUTPUT RECORD files SEQUENTIALLY and for processing RECORD files DIRECTLY whether for INPUT, OUTPUT or UPDATE providing that each record in such a file has an associated uniquely identifying key field.

2.6. Strings

Pascal, PL/I and Algol 68 all have facilities for manipulating vectors of fixed length as described in section 2.3. The essential difference between a string and such a vector is that its length is variable; this concept of a string most often arises in the context of character handling and thus this section will concentrate on character strings. In Pascal, there are no facilities for handling such strings - to the Pascal programmer a string is just a packed, fixed length array of characters. PL/I and Algol 68, however, do recognise the distinction between strings and vectors although there is a fundamental difference in their approaches. The PL/I declaration

```

DECLARE S CHAR(10) VARYING;

```

defines S as a variable length string, with a maximum of ten characters. The current length of S is given by $\text{LENGTH}(S)$ and can never exceed 10. Any attempt to assign a longer string to S will result in the assigned string being truncated to be of ten characters. In Algol 68, the declaration

```
flex [1:10] char s
```

initially allocates sufficient space to store up to 10 characters but during execution this length will be automatically extended or contracted such that any string assigned to s can be accommodated. The declaration

```
string s
```

is equivalent in Algol 68 to

```
flex [1:0] char s
```

In [19], Housden lists the fundamental string operations and constructs an abstract model for string manipulation. PL/I and Algol 68 both match up to this ideal only in part. Both provide facilities for creation of strings as outlined above, both have facilities for interrogating the length of a string (LENGTH in PL/I, upb in Algol 68), and both provide a concatenation operation ($||$ in PL/I, $+$ in Algol 68). It is in the manipulation of substrings that most of the difficulties arise.

The PL/I substring selector function SUBSTR has the general form $\text{SUBSTR}(S,I,J)$ and when applied to a string S with integers I,J such that $1 \leq I \leq J \leq \text{LENGTH}(S)$ provides access to the I th through to the J th characters of S . No copies of the string are made; SUBSTR essentially provides addressability and thus the length of $\text{SUBSTR}(S,I,J)$ is fixed at $J - I + 1$. If a string of length greater than this is assigned to $\text{SUBSTR}(S,I,J)$ it is truncated and similarly a string shorter than this is padded with blanks. Thus a substring of a string is itself not a string (not being variable in length)! This is also true in Algol 68 if a substring is accessed using a trimmed array. However, in [19], Housden describes an Algol 68 library prelude which provides a facility for defining and manipulating substrings which are truly variable in length.

For searching for a given string within a subject string, PL/I provides the function INDEX . INDEX returns the index to the start of the leftmost substring of its first argument that matches its second argument. If no such match exists, 0 is returned. In Algol 68, the procedure char in string

of mode proc (char, ref int, string) bool is provided. When applied to argument (c,i,s) the procedure delivers true if c is contained in s, in which case the index of its first occurrence in s is assigned to i and otherwise delivers false. An Algol 68 procedure equivalent to the PL/I INDEX is not provided but is easy to write, see [19].

In the last decade, the specialist language for describing string manipulation algorithms has been SNOBOL [15] and a course on data structures will probably include an appreciation of that language and in particular of patterns and pattern matching. If the core language for the data structures course is Algol 68, then the teacher is particularly fortunate. He need not go into unnecessary details of SNOBOL syntax - all the string processing and pattern matching facilities of SNOBOL are available in a library prelude written by HOUSDEN and KOTARSKI [18]. Experience with Algol 68 patterns has been reported in [34] where it is used to write a logic teaching package. More recently, ICON [14] has been receiving considerable interest as a potential successor of SNOBOL4. By employing *generators* which are capable of producing alternative values, together with a goal-driven method of expression evaluation, ICON provides the string processing facilities of SNOBOL4 without the complexities associated with patterns. It would be an interesting and worthwhile exercise to produce an Algol 68 library prelude for ICON-like string processing.

2.7. Lists

To represent any linked structure the programmer must use pointers. This can be achieved implicitly by the use of array indices but this technique offers very little protection to the programmer and debugging can be a horrendous task. The three languages under consideration all recognise this fact and provide facilities which enable the programmer to manipulate storage addresses.

Any Algol 68 programmer has to master the use of references early in his programming life - in fact, as soon as he wishes to use a variable rather than a constant. Orthogonality of the language implies that any valid Algol 68 mode, amode, can be prefixed by ref to produce another valid mode, ref amode, and each of these modes are distinct. Thus ref real is different from ref int and different again from ref ref real. Any variable whose mode begins with two successive refs represents a pointer. For example,

```
char c; ref char p := c;
```

declares an object c of mode ref char and a pointer p of mode ref ref char which points to the space referred to by c. Any pointer variable can be assigned nil which means that it points to no space. In Algol 68, := (or is) and ≠ (or isnt) provide the facility for testing whether two objects both of the same mode, ref amode, are or are not identical. Linked structures are defined in Algol 68 using record structures containing reference modes and are manipulated using pointer variables. The following code illustrates the use of Algol 68 references to construct and manipulate a simple linear linked list of characters.

```

mode item = struct (char val, ref item next);
mode list = ref item;
list empty = nil;
list ℓ := empty;
proc push = (char c) void:
  † inserts c at the head of the list †
  ℓ := heap item := (c,ℓ);
proc pop = char:
  † delivers the head of ℓ and resets ℓ to its tail †
  if ℓ is empty then print ("failpop",newline);skip
  else char c = val of ℓ; ℓ := next of ℓ; c
fi

```

This example illustrates the use of the Algol 68 heap generator, heap item. When this was encountered, storage for an object of mode item was reserved not on the main stack as was used for ℓ but in a different region of store called the heap. Space created by heap generators remains available as long as required and the programmer does not have to explicitly release space no longer required. This implies that any Algol 68 implementation must use some form of garbage collection to retrieve redundant space on the heap.

The use of pointers may come naturally to the Algol 68 programmer but for the PL/I programmer it entails the study of additional features not necessarily regarded as part of the core of the language. Indeed, the whole facility enabling the manipulation of BASED variables and POINTERS was only added to PL/I late in its development. Every PL/I variable has a storage class attribute (see Table 2) which, unless explicitly declared, is assigned AUTOMATIC by default. In PL/I, POINTER is a single data type and can be regarded as being equivalent to the Algol 68 mode which is the

union of all references. Thus, POINTERS do not offer so much protection to the programmer as Algol 68 references but can avoid the complexities of using the Algol 68 union in general list processing.

The simplest use of POINTER is illustrated by the code

```
DECLARE P POINTER, C CHARACTER(1);
P = ADDR(C);
```

which assigns to P the address of C. This is equivalent to the Algol 68 code char c; ref char p := c. Corresponding to the Algol 68 nil, there is a special value NULL which is not the address of any data in the computer and can be assigned to any POINTER variable.

Table 2
Storage Class Attributes in PL/I

<u>Storage Class</u>	<u>Description</u>
STATIC	Storage is allocated once only and any initial value is assigned when the program is loaded for execution.
AUTOMATIC	Storage is allocated and any initial value is reassigned each time the procedure or block containing the declaration begins execution.
CONTROLLED	Storage is allocated for the variable upon execution of an <u>ALLOCATE</u> statement. Several instances of a variable, X, may exist since a new instance is defined each time <u>ALLOCATE X</u> is executed. A reference to X then refers to the most recently allocated instance of X and the next most recently allocated instance is accessible only after a <u>FREE X</u> is executed.
BASED	As in <u>CONTROLLED</u> case, storage is allocated by the program but unlike the <u>CONTROLLED</u> case, all existing copies of a <u>BASED</u> variable may be referenced at any time using a <u>POINTER</u> variable.

In list processing, it is the use of POINTERS which point to BASED variables which is most important. Taking the example programmed in Algol 68 above, the corresponding PL/I declaration would be

```
DECLARE 1 ITEM BASED,
        2 VAL CHARACTER(1),
        2 NEXT POINTER;
```

Each item in the list has to have its memory explicitly allocated by a statement such as

```
ALLOCATE ITEM SET(P);
```

This statement allocates storage suitable for an item and sets the pointer P to its address. Each time the statement is executed, a new memory area will be allocated and P will be assigned the address of the new memory area. Since, in this application the items are to be linked together, after the first execution of the statement, P is assigned to an initial pointer, L, by

```
L = P;
```

The pointer field should be set to NULL using

```
L → NEXT = NULL;
```

Subsequently, if L points to the current head of the list, the new item can be pushed onto the list using

```
P → NEXT = L;
```

```
L = P;
```

The explicit allocation and release of storage for BASED variables in PL/I, although awkward for the programmer, does obviate the necessity of a garbage collector.

Pascal, like Algol 68, distinguishes between its pointer values according to the type (mode) of object to which they refer. A pointer type, in Pascal, pointing to elements of type t, is said to be *bound* to t. The value nil is an element of every pointer type and points to no element at all.

The Pascal declaration

```
var p: ↑ char
```

declares p as a pointer variable bound to char. This means that p is a reference to a variable of type char and p↑ is used to denote that variable.

The dereferencing coercion of Algol 68 is made quite explicit by the use of the upward arrow (and what a pity it is not a downward facing arrow which would make it so much easier to explain).

A variable pointed to by a pointer type is created using the standard procedure `new`. The call `new(p)` allocates a variable of the correct type and assigns its address to `p`. The Pascal operator `<>`, corresponding to the Algol 68 `≠`, enables the programmer to test for inequality of pointer types. The Pascal code for our example is:

```

type link = ↑item;
   item = record
       val: char;
       next: link
   end;
var l: link;
procedure push (var c: char);
   var p: link;
begin new(p);
   p↑.val:= c; p↑.next:= l;
   l:=p
end; {push}
function pop: char;
begin if l <> nil then
   begin pop:= l↑.val;
       l:= l↑.next
   end
end {pop}

```

The space allocated to dynamic variables in Pascal accessed through a pointer remains available as long as it can be accessed and is not explicitly released by the programmer. Although implementations vary, Pascal promises no garbage collection and thus, when list processing, the Pascal programmer may well be advised to maintain an explicit list of free items and avoid the use of `new` altogether. Pascal does have one advantage over Algol 68 - the facilities provided for creating the correct amount of space for a variant structure and delivering a pointer to the space means that there is no need for an equivalent to the cumbersome Algol 68 union in general list processing.

Euclid [24] is an interesting language based on Pascal and primarily

designed to allow program verification. For this reason the use of pointers is severely constrained in Euclid; all pointers must refer to dynamic variables which are allocated as part of a *collection*. A collection is a group of variables of the same type and just as an index value uniquely determines an element of an array so a pointer into a collection uniquely determines a variable of that collection.

3. CONCLUSIONS

In Section 2, the traditional core items in a data structures course have been considered and the facilities for their manipulation in the languages PL/I, Algol 68 and Pascal have been reviewed. No language is perfect for the teaching of all these topics but, overall, Algol 68 does appear to be the best (see Table 3). Its real strength is extensibility; where Algol 68 lacks explicit facilities, they can easily be incorporated as a library prelude, e.g. [4,13,18]. This paper has not been written to suggest that only Algol 68 should be taught in the course; it is essential to review many languages in a data structures course, comparing and contrasting each of their facilities. Certainly most of the languages mentioned in this paper need to be mentioned but it is not necessary for the student to program in many of them. In fact, if the student can program well in Algol 68, that is

Table 3.

A comparison of the languages

FACILITY	PL/I	Algol 68	Pascal
Availability of compilers	0	0	+
Ease of learning	-	0	+
Literature	+	0	0
manipulation of:			
atomic information	0	0	+
arrays	+	+	-
sets	-	0	+
record structures	0	+	+
files	+	0	-
character strings	0	0	-
linked structures	-	+	0

+ = good; 0 = fair; - = poor.

probably sufficient. The orthogonality of the language means that the modes of operators and functions associate them directly with the data structures on which they act. This results in the students readily appreciating the important concept of abstract data types as illustrated by the class construct in Simula [3,9].

As claimed in [33], the ADA language has been designed with three overriding concerns: (i) a recognition of the importance of program reliability and maintenance, (ii) a concern for programming as a human activity and (iii) efficiency. The difficulties of achieving all these goals simultaneously are immense and the design of ADA has been influenced by a large number of programming languages but most obviously by Pascal and its various derivatives (e.g. Euclid). It has preserved the clarity of Pascal but nevertheless has most of the facilities described in this paper for the manipulation of data structures. In particular, the package module provides a useful construct for abstract data types with the facility of information hiding whereby the user may be prevented access to some of the internal entities. Looking into the future, one sees an increasing use of parallel processing for which both Algol 68 and PL/I offer limited facilities although these are seldom implemented. ADA tasking offers facilities similar to Hoare's communicating sequential processes [17]. In summary, ADA appears to be a possible successor to Algol 68 as a core language but, at this stage of ADA's development, there are still many questions to be answered. Until we have working compilers and some experience of the language, the practicality of ADA in a teaching environment is difficult to assess. The next three years should answer these questions and if ADA proves as versatile a language as its designers hope, it will be adopted widely in both academic and commercial circles. Algol 68 has never achieved that claim - although accepted by academics as a useful teaching and research tool, the non-academic world remains hostile to a language which it sees as an inefficient monster serviced by a cult of devotees headed by mysterious high priests who understand two-level grammars. Although exaggerated, much of this criticism is fair, the language (like PL/I) does require a large compiler, does attract devotees and the report is quasi-incomprehensible. Perhaps it should be accepted that the full language is mainly for academics where its usefulness in teaching at the post foundation level and in research can be fully explored. This paper has been a contribution to the campaign for the use of Algol 68 in such an environment. At the foundation

level, Algol 68 cannot be recommended. The extensibility of the language means that a novice programmer will undoubtedly fall into the pitfall of using constructs which he cannot be expected to understand nor appreciate. One solution to this problem is the adoption of an Algol 68 subset with a suitably efficient compiler. Designing such a subset whilst preserving orthogonality is not an easy task although some notable attempts have been made [e.g. 30].

REFERENCES

- [1] ANSI X3-53, *Programming language PL/I*, (1976).
- [2] BARRON, D.W., *An introduction to the study of programming languages*, (Cambridge Univ. Press, 1977).
- [3] BIRTWISTLE, G.M., O.-J. DAHL, B. MYHRHAUG & K. NYGAARD, *SIMULA BEGIN*, (Auerbach Publishers Inc., Philadelphia, 1973).
- [4] BLACK, A. & V.J. RAYWARD-SMITH, *Algol H - A superlanguage of Algol 68*, *Algol Bulletin*, 42, (May, 1978).
- [5] BRAILSFORD, D.F. & A.N. WALKER, *Introductory Algol 68 programming*, (Ellis Horwood, Chichester, 1979).
- [6] COLIN, A.J.T., *Programming and problem-solving in Algol 68*, (Macmillan, London, 1978).
- [7] CONWAY, R. & D. GRIES, *An introduction to programming: A structural approach using PL/I and PL/C*, (Winthrop, Englewood Cliffs, N.J., 1973).
- [8] CURRICULUM COMMITTEE ON COMPUTER SERVICE, *Curriculum '78 - Recommendations for the undergraduate program in computer science*, *CACM* 22: 3 (March 1979), 147-166.
- [9] DAHL, O.-J. & C.A.R. HOARE, *Hierarchical program structures*, in *Structured programming*, by O.-J. Dahl, E.W. Dijkstra & C.A.R. Hoare, (Academic Press, London, 1972).
- [10] DIJKSTRA, E.W., *Notes on structured programming*, in *Structured programming*, by O.-J. Dahl, E.W. Dijkstra & C.A.R. Hoare, (Academic Press, London, 1972).

- [11] FIKE, C.T., *PL/I for scientific programmers*, (Prentice-Hall, Englewood Cliffs, N.J., 1970).
- [12] FINDLAY, W. & D.A. WATT, *Pascal: an introduction to methodical programming*, (Pitman, 1978).
- [13] GARSIDE, G.R. & P.E. PINTELAS, *An Algol 68 package for implementing graph algorithms*, *Computer Journal*, 23: 3 (August, 1980) 237-242.
- [14] GRISWOLD, R.E., D.R. HANSON & J.T. KORB, *The Icon programming language: an overview*, *SIGPLAN Notices*, 14: 4 (April, 1979) 18-31.
- [15] GRISWOLD, R.E., J.F. POAGE & I.P. POLONSKY, *The SNOBOL4 programming language*, 2nd edition, (Prentice-Hall, Englewood Cliffs, N.J., 1971).
- [16] HOARE, C.A.R., *Notes on data structuring*, in *Structured programming*, by O.-J. Dahl, E.W. Dijkstra & C.A.R. Hoare, (Academic Press, London, 1972).
- [17] HOARE, C.A.R., *Communicating sequential processes*, *CACM* 21: 8 (August, 1978) 666-677.
- [18] HOUSDEN, R.J.W. & N. KOTARSKI, *Character string pattern matching in Algol 68*, *SIGPLAN Notices* 12: 6 (June, 1977) 144-152.
- [19] HOUSDEN, R.J.W., *On string concepts and their implementation*, *Computer Journal* 18: 2 (1975), 150-156.
- [20] HOUSDEN, R.J.W. & V.J. RAYWARD-SMITH, *An information structures course based on Algol 68-R*, paper presented at the Conference on Experience with Algol 68, Liverpool University, 1975 (copies available from the authors).
- [21] HUGHES, J., *PL/I programming*, (Wiley, Chichester, 1973).
- [22] ICHBIAH, J.D. et al., *Rationale for the design of the ADA programming language*, *SIGPLAN Notices* 14: 6B (June, 1979).
- [23] JENSEN, K. & N. WIRTH, *Pascal: user manual and report*, second edition, (Springer-Verlag, New York, 1975).

- [24] LAMPSON, B.W., J.J. HORNING, R.L. LONDON, J.G. MITCHELL & G.J. POPEK, *Report on the programming language Euclid*, SIGPLAN Notices 12: 2 (February, 1977) 1-79.
- [25] LINDSEY, C.H. & S.G. VAN DER MEULEN, *Informal introduction to Algol 68* (North-Holland, London, 1977).
- [26] McGETTRICK, A.D., *Algol 68: A first and second course*, (Cambridge Univ. Press, 1978).
- [27] MEEK, B., *Fortran, PL/I and the Algols*, (Macmillan, London, 1978).
- [28] NICHOLLS, J.E., *The structure and design of programming languages*, (Addison-Wesley, Mass., 1975).
- [29] PAGAN, F.G., *A practical guide to Algol 68*, (Wiley, Chichester, 1976).
- [30] PAGAN, F.G., *Nested sublanguages of Algol 68 for teaching purposes*, SIGPLAN Notices 15: 7 and 8 (July-August, 1980) 72-81.
- [31] PETERSON, W.W., *Introduction to programming languages*, (Prentice-Hall, Englewood Cliffs, N.J., 1974).
- [32] PRATT, T.W., *Programming languages: design and implementation*, (Prentice-Hall, Englewood Cliffs, N.J., 1975).
- [33] Preliminary ADA Reference Manual, SIGPLAN Notices 14: 6A (June, 1979).
- [34] RAYWARD-SMITH, V.J., *The use of Algol 68 pattern matching to describe a formal logic system*, Algol Bulletin 44 (May, 1979).
- [35] SCHNEIDER, G.M., S.W. WEINGART & D.M. PERLMAN, *An introduction to programming and problem solving with Pascal*, (Wiley, Chichester, 1978).
- [36] TUCKER, A.B., *Programming languages*, (McGraw-Hill, New York, 1977).
- [37] VALENTINE, S.H., *Comparative notes on Algol 68 and PL/I*, Computer Journal 17: 4 (1974), 325-331.
- [38] VAN WIJNGAARDEN, A., et.al., *Revised report on the algorithmic language Algol 68*, (Springer-Verlag, New York, 1976).
- [39] WEISSMAN, C., *LISP 1.5 primer*, (Dickenson Publishing Co., Belmont, California, 1967).

CONTEXT-FREE GRAMMARS AND DERIVATION TREES IN ALGOL 68*)

V. LINNEMANN

ABSTRACT

It is shown how context-free grammars and corresponding derivation trees can be used by application programmers if adequate language tools are provided. These language tools are applicable to symbol manipulation problems, especially formula manipulation, and to program generators. The language tools are defined on the basis of ALGOL 68, and they have the remarkable property that syntactically and semantically correct programs operate only on syntactically correct derivation trees.

*) This work was supported by a scholarship from the German Academic Exchange Service and was undertaken while the author was with the Computer Systems Research Group of the University of Toronto.

1. INTRODUCTION

This paper deals with language tools for generating and manipulating of syntactical structures, esp. programs written in high-level languages. In order to show what is meant by this two small problems are given in this introduction which can be solved in only a rather cumbersome way by using conventional programming methods. In addition, conventional programming languages do not aid very much in detecting errors as soon as possible, i.e. at compile-time.

Let us assume we have a very simple programming language which contains only arithmetic expressions and whose syntax is given as follows (the empty word is denoted by ϵ):

```

<expr>      ::= <expr> + <term> | <term>
<term>      ::= <term> * <factor> | <factor>
<factor>    ::= <number> | <id> | (<expr>)
<number>    ::= <digit> <number> | <digit>
<id>        ::= <idhead> <idtail>
<idhead>    ::= <letter>
<idtail>    ::= <idtail1> <idtail> |  $\epsilon$ 
<idtail1>   ::= <letter> | <digit>
<letter>    ::= a|b|c| ... |z
<digit>     ::= 0|1|2|3|4|5|6|7|8|9

```

The identifiers $\langle id \rangle$ in this language denote variables which contain input values. The output of a program in this language is the value of the arithmetic expression. Now let us assume we want to write a program which reads an integer value n and produces a program

$$(\dots (a_0 * x + a_1) * x + a_2) * \dots * x + a_n),$$

i.e. we want to write a very simple program generator, for example for $n = 2$ the program should print the program

$$((a_0 * x + a_1) * x + a_2).$$

Such a program generator could be written in ALGOL 68 (see [2] or [12]) as follows:


```

BEGIN
  INT n; read(n);
  TO n DO print("(") OD;
  print("a0");
  FOR i TO n
  DO
    print((" * x + a", i, "))
  OD
END

```

One area where program generators are required is the area 'Automatic Programming' where programs are assembled automatically using more or less descriptive statements of the problem to be solved (see WILLIAMS [13], MANNA [8] or GOLDBERG [1]). If program generators are written in such a straightforward way, the following disadvantages become obvious:

- a) The programming task is kind of awkward, esp. counting the brackets is not very convenient.
- b) The syntactical correctness of the generated programs is not guaranteed, for example if we replace the statement

```
TO n DO print("(") OD
```

by the statement

```
TO n-1 DO print("(") OD
```

the program remains correct as far as the language specifications for ALGOL 68 are concerned, i.e. the compiler accepts and translates the program, but the arithmetic expressions which can be generated show a bracket mismatch. ALGOL 68 was designed as a general-purpose language. If you want to design a special-purpose language for program generation, then special features become appropriate such that the syntactical correctness of all generated programs can already be checked by the compiler which compiles the program generator. As far as the example is concerned this means that the language tools do not allow writing programs which generate expressions with a bracket mismatch. This paper shall show how such language tools can be defined by using methods from the theory of formal languages.

The second example is a simple formula manipulation problem. Suppose we want to write a program which reads an arithmetic expression which con-

tains the variable x . The program is supposed to produce the formal derivation using the variable x conforming to the rules of usual mathematics. For example the input

$$x * x$$

should produce the output

$$1 * x + 1 * x.$$

If we try to write a program for this problem using a conventional programming language we have to do a lot of programming only for reading the expression, checking it and transforming it into an appropriate internal form, i.e. we have to write a parser manually. In order to avoid this additional language tools shall be provided in the sequel.

2. A SYNTAX-ORIENTED APPROACH

The new programming tools shall be explained in terms of the simple programming language mentioned in the introduction. The language tools shall be added to the programming language ALGOL 68, the tools can be defined for other base programming languages which allow a static type checking in a similar way.

By using the grammar for simple arithmetic expressions mentioned in the introduction, we add some new data types to ALGOL 68. The new types and the corresponding values are summarized in the following table:

<u>Type</u>	<u>Values</u>
EXPR	$L(\langle \text{expr} \rangle)$ = set of all words of terminal symbols derivable from $\langle \text{expr} \rangle$, i.e. set of all correct expressions
TERM	$L(\langle \text{term} \rangle)$
FACTOR	$L(\langle \text{factor} \rangle)$
NUMBER	$L(\langle \text{number} \rangle)$
ID	$L(\langle \text{id} \rangle)$
IDHEAD	$L(\langle \text{idhead} \rangle)$
IDTAIL	$L(\langle \text{idtail} \rangle)$
IDTAILEL'	$L(\langle \text{idtailel} \rangle)$
LETTER	$L(\langle \text{letter} \rangle)$
DIGIT	$L(\langle \text{digit} \rangle)$

That means, we define new data types corresponding one-to-one to the nonterminal symbols of the underlying grammar. Values of these types are computed by a language construction called *generating expression*, for example generating expressions for values of type EXPR look like

EXPR GEN σ NEG .

σ is a string which is derivable starting with the nonterminal $\langle \text{expr} \rangle$ by means of the underlying grammar, but adequate syntactical positions can be occupied by values of the new data types, separated by the separators { and }. For example the following piece of program contains valid generating expressions:

```

EXPR e; TERM t; CO a variable e of type EXPR and a variable t of
      type TERM are declared CO
e:= EXPR GEN x + y NEG; CO e gets the expression x + y CO
t:= TERM GEN v * w NEG; CO t gets the term v * w CO
e:= EXPR GEN {e} + a + b + {t} NEG CO e gets the expression
      x + y + a + b + v * w CO

```

In order to define formally the syntactical positions where the insertion of values in generating expressions is allowed, we augment the grammar by the following rules:

```

<expr>      ::= {EXPR}
<term>      ::= {TERM}
<factor>    ::= {FACTOR}
<idhead>    ::= {ID}
<idtail>    ::= {IDTAIL} | {ID}

```

For example the symbol {EXPR} stands for the set of all ALGOL-68 expressions which deliver a value of type EXPR, enclosed in { and }. By means of the augmented grammar the set of all correct generating expressions is formally defined. We shall call such a grammar a *generating grammar*. In addition to generating expressions we introduce a monadic operator *idt* which takes INT-values and delivers a corresponding IDTAIL-value by computing the decimal notation. Using these tools, we can solve the first problem mentioned in the introduction, namely generating polynomial expressions, as follows (We use one obvious abbreviation: The specification of the type of a generating expression is omitted where it is uniquely defined by using the context, for example

```

FACTOR f:= GEN a0 NEG
stands for
FACTOR f:= FACTOR GEN a0 NEG ):

```

```

BEGIN
  INT n; read(n);
  FACTOR f:= GEN a0 NEG;
  FOR i TO n
  DO
    f:= GEN ({f} * x + a {idt i}) NEG
  OD;
  print(f)
END.

```

It is obvious that only correct expressions can be generated if the new language tools are used, for example an assignment

```
f:= GEN {f} * x + a {idt i} NEG
```

is *syntactically* wrong because of one missing bracket, and the compiler can detect this error. Moreover, parenthesis structures appear always statically in the program generator and not dynamically as in the solution mentioned in the introduction. This avoids bracket counting and enhances readability. Clearly, the concept is not limited to the simple expression grammar, generating grammars can be defined for "real" programming languages.

Additional operators, like `idt` in the example, can be defined by special statements in addition to the generating grammar, for example by

```
COP idt = INT, STRING TO IDTAIL,
```

which means that the operator `idt` should convert an `INT` - or `STRING` - input-value to the same format as if the value would be printed (i.e. decimal notation), check whether it is derivable starting with `<idtail>` and deliver a corresponding `IDTAIL`-value. The implementation of these operators can be derived automatically by using such a definition. Saying it in another way, you can view a generating grammar and corresponding `COP`-Declarations as a new way of defining special data types.

Now let us turn to the second problem mentioned in the introduction. We generalize the proposed language tools for program generation as follows:

1) The corresponding values of the new types are not only strings but derivation trees corresponding to the underlying grammar; that means strings with appropriate syntactic information. The generating expressions generate derivation trees instead of simple strings, and if a value of a new data type is read, an appropriate derivation tree is constructed.

2) In order to work with these trees we need an additional tool for traversing a tree which preserves the property of static types. This tool is called *root inspection* and it shall be described by means of examples using the expression grammar from the introduction:

Assume we declare

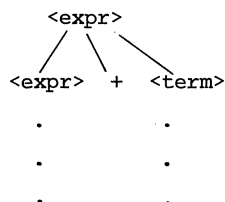
```
EXPR e, e1; TERM t1, t2;
```

Then the expression

```
ROOT e INTO (e1: t1, t2)
```

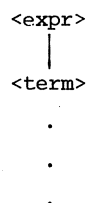
is a root inspection, and it works as follows:

The input tree is the tree stored in the variable e. If this tree is a tree



i.e. if the first alternative for the nonterminal <expr> is used, then the root inspection delivers the INT-value 1, e1 gets as value the <expr>-subtree on the left, t1 gets the <term>-subtree on the right.

If the tree in the variable e is a tree



i.e. if the second alternative for the nonterminal <expr> is used, the root inspection delivers the value 2 and the <term>-subtree is put into the variable t2. Root inspections for the other data types are defined in a similar way.

For the sake of simplicity we define in addition to root inspections a dyadic operator `top`. This operator can be called as follows:

`a top b,`

where `a` delivers a variable of type `A`, `b` delivers a value of type `B`, and `A` is the data type which corresponds to the nonterminal `<A>`, `B` is the data type which corresponds to the nonterminal ``. `top` works as follows: `a top b` delivers the boolean value `TRUE` if `b` is a tree which corresponds to a derivation

$$\langle B \rangle \rightarrow^* \langle A \rangle \rightarrow^* \sigma.$$

In this case, the subtree which corresponds to $\langle A \rangle \rightarrow^* \sigma$ is assigned to the variable `a` (if several trees exist, the smallest one is chosen). If `b` doesn't satisfy this condition, the boolean value `FALSE` is delivered and `a` remains unchanged.

It is obvious that root inspections and the `top`-operator can be used for traversing derivation trees and that derivation trees are always correct corresponding to the underlying grammar.

In addition, the ALGOL 68 procedures "read" and "print" can be used for variables of the new data types. "read" reads a string terminated by the symbol `NEG` and tries to produce a corresponding derivation tree. If this is not possible, a runtime error occurs, otherwise the tree is assigned to the variable. The procedure "print" prints the leaves of the tree from left to right.

The problem of computing the formal derivation of an expression mentioned in the introduction can now be solved as follows:

```
BEGIN
  PROC exprderivation = (EXPR e1) EXPR:
  BEGIN
    EXPR e; TERM t;
    CASE
      ROOT e1 INTO (e: t, t)
    IN
      GEN {exprderivation(e)} + {termderivation(t)} NEG,
      GEN {termderivation(t)} NEG
    ESAC
  END;
```

```

PROC termderivation = (TERM t1) TERM:
BEGIN
  TERM t; FACTOR f;
  CASE
    ROOT t1 INTO (t:f,f)
  IN
    GEN ({termderivation(t)} * {f} +
      {factorderivation(f)} * {t} )
    NEG,
    GEN {factorderivation(f)} NEG
  ESAC
END;

OP cfactor = (EXPR e) FACTOR:
BEGIN
  CO cfactor puts brackets around e if e isn't already a factor CO
  FACTOR f;
  IF
    f top e
  THEN
    f
  ELSE
    GEN ({e}) NEG
  FI
END;

PROC factorderivation = (FACTOR f1) FACTOR:
BEGIN
  EXPR e;
  CASE
    ROOT f1 INTO ( , , e)
  IN
    GEN 0 NEG,
    IF f1 = "x" THEN GEN 1 NEG ELSE GEN 0 NEG FI,
    cfactor (exprderivation(e))
  ESAC
END;

```

```

CO Mainprogram CO
EXPR e; read(e);
e:= exprderivation(e);
print(e)
END

```

This example shows how to combine generating expressions and root inspections: The root inspections analyze the tree, and the generating expressions use this information for generating another tree. It is important that the compiler can do many checks very easily, due to the static types it can guarantee that the program works only with correct trees corresponding to the underlying grammar, no runtime checks are required for example to check how many subtrees a tree has or how the root of the tree is labelled.

The next example gives an idea of how to construct a translator using the described tools. Let us assume we want to write a translator which translates a simple arithmetic expression into a corresponding program for a stack machine. In addition, the translator is supposed to fold constant subexpressions by computing the value at compile time. We augment the generating grammar from the previous example by the following rules:

```

<stackexpr> ::= <stackexpr> <stackexpr> <operator> |
              LOAD <id> | LOAD <number> | {STACKEXPR}
<operator> ::= ADD | MULT

```

The procedure *fold* which does the folding of constant subexpressions is defined as follows, the procedure *compute* which computes the value of a constant subexpression is omitted:

```

PROC. fold = (STACKEXPR e) STACKEXPR:
BEGIN
  STACKEXPR opr1, opr2; OPERATOR op; ID id; NUMBER number1, number2;
  IF
    ROOT e INTO (opr1: opr2: op, , ) = 1
  THEN
    IF ROOT opr1 INTO ( , , number1) = 3 and
      ROOT opr2 INTO ( , , number2) = 3
    THEN
      number1:= compute(number1, number2, op);
      GEN LOAD {number1} NEG
    END
  END
END

```



```

    ELSE
      e
    FI
  ELSE
    e
  FI
END

```

Now we can write the translator, the procedure *expr* is the solution:

```

PROC expr = (EXPR e1) STACKEXPR:
BEGIN
  TERM t; EXPR e;
  CASE ROOT e1 INTO (e: t, t)
  IN
    fold( GEN {expr(e)} {term(t)} ADD NEG ),
    term(t)
  ESAC
END;

PROC term = (TERM t1) STACKEXPR:
BEGIN
  FACTOR f; TERM t;
  CASE ROOT t1 INTO (t: f, f)
  IN
    fold( GEN {term(t)} {factor(f)} MULT NEG ),
    factor(f)
  ESAC
END;

PROC factor = (FACTOR f1) STACKEXPR:
BEGIN
  EXPR e; ID id; NUMBER nu;
  CASE ROOT f1 INTO (nu, id, e)
  IN
    GEN LOAD {nu} NEG,
    GEN LOAD {id} NEG,
    expr(e)
  ESAC
END

```

For the input $(2 * 3 + c * d) * e$ *expr* delivers the STACKEXPR-value

```
LOAD 6 LOAD c LOAD d MULT ADD LOAD e MULT.
```

3. IMPLEMENTATION ISSUES

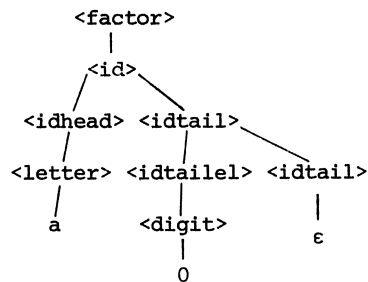
In order to be able to do syntax checks in linear time it is necessary to restrict the type of the underlying grammars. This restriction depends on the availability of parser generators. If, for example, an LL1 generator is available, one would allow only LL1 grammars for the definition of the new data types. This parser generator, perhaps in a modified form, would be used to generate a translation procedure for translating generating expressions and for the "read" procedure.

As far as derivation trees are concerned, it is obvious that the implementation of generating expressions in a straightforward manner, i.e. by copying and substituting complete trees, is a rather slow and memory consuming operation. This can be avoided by using pointers and by using trees in a recursive manner. This will be clarified by an example: Assume the compiler has to translate the following program, one of the previous examples:

```
BEGIN
  INT n; read(n);
  FACTOR f := GEN a0 NEG;
  FOR i TO n
  DO
    f := GEN ( {f} * x + a {idt i} ) NEG
  OD;
  print(f)
END.
```

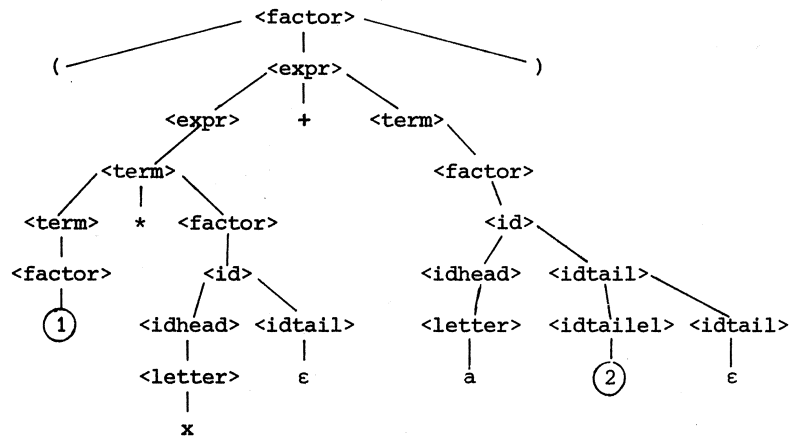
The compiler generates tree structures corresponding one-to-one to the generating expressions in the program as follows:

1) GEN a0 NEG :



This tree is called t1.

2) GEN ({f} * x + a {idt i}) NEG :

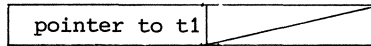


This tree is called t2.

The circled numbers are indices for runtime arrays where pointers for the actual subtrees are stored. A variable of one of the new data types contains a pointer pointing to a tree structure and a (possibly empty) pointer pointing to an array where pointers to subtrees are stored which have been substituted during run-time. The circled numbers are indices for these arrays.

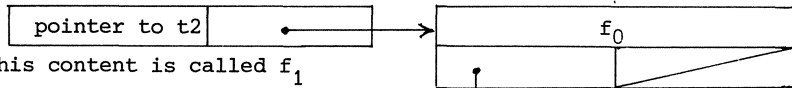
A trace for the example program would show the following contents for the variable f:

a) Initialization:



This content is called f_0 .

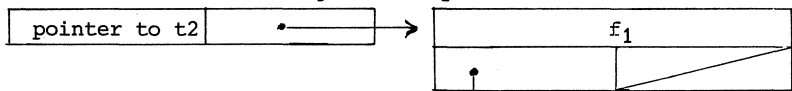
b) After the first run through the loop:



This content is called f_1

pointer to a tree generated
by idt

c) After the second run through the loop:



pointer to a tree generated
by idt

The implementation of root inspections is straightforward.

The implementation techniques are described in more detail in [7].

4. CONCLUDING REMARKS

The basic ideas for incorporating context-free grammars and derivation trees into high-level languages with a static type concept, esp. ALGOL 68 have been introduced. Due to space limitations, many issues were not treated here. For example, in [5] and [7] methods are given for inserting operators like `idt` automatically, and a generating grammar for ALGOL 68 is given in [5] thus providing the basis for writing program generators for ALGOL-68 programs and thus showing that the tools are useful not only for artificial programming languages but also for existing ones. In [7] a method is given for generating lists, for example statement lists, in a more descriptive manner.

Hopefully this paper has shown that it makes sense to use methods from the theory of formal languages in order to provide type-safe programming tools by using static type-checking methods for writing program generators and programs which manipulate syntactic structures.

ACKNOWLEDGEMENT

I would like to thank Prof. E.C.R. Hehner from the University of Toronto for critical comments.

REFERENCES

- [1] GOLDBERG, P.C., *Automatic programming*, in: Programming Methodology 4th Informatics Symposium IBM Germany, Wildbad 1974, Lecture Notes in Computer Science 23, 1974, 347-361.
- [2] LINDSEY, C.H. & S.G. VAN DER MEULEN, *Informal Introduction to ALGOL 68* North Holland/American Elsevier 1977.
- [3] LINNEMANN, V., *Syntaxgesteuerte Generierung von ALGOL-68-Programmen* Informatik-Bericht Nr. 7706 Techn. Univ. Braunschweig (West Germany), Nov. 1977.
- [4] LINNEMANN, V., *Syntaxgesteuerte Generierung von ALGOL-68-R-Programmen*, in K. Alber (ed.): 5. Fachtagung Programmiersprachen der GI, Braunschweig, West Germany 1978, Informatik-Fachbericht 12, 145-156. An Abstract of this paper can be found in English in: Zentralblatt für Mathematik und ihre Grenzgebiete, vol. 373, 19. Nov. 1978, p. 427, no. 68015.
- [5] LINNEMANN, V., *Sprachelemente zur Generierung und Umformung syntaktischer Strukturen auf der Basis von ALGOL-68 und deren theoretische Untersuchung*, Doctoral Dissertation Techn. Univ. Braunschweig, West Germany 1979.
- [6] LINNEMANN, V., *Kontextfreie Grammatiken und Ableitungsbäume als Hilfsmittel bei der Programmierung*, Angewandte Informatik 2/1980, 60-66.
- [7] LINNEMANN, V., *Context-free grammars and derivation trees as programming tools*, Technical Report CSG-117, Computer Systems Research Group Univ. of Toronto, Canada, August 1980.
- [8] MANNA, Z. & R.A. WALDINGER, *A deductive approach to program synthesis*, ACM Transactions on Programming Languages Vol. 2 No. 1 (Jan. 1980) 90-121.
- [9] MAURER, H. & W. STUCKY, *Ein Vorschlag für die Verwendung syntaxorientierter Methoden in höheren Programmiersprachen*, Angewandte Informatik 5/1976, 189-195.

- [10] SALOMAA, A., *Formal Languages*, Academic Press 1973.
- [11] SILVERBERG, B.A., *Using a grammatical formalism as a programming language*, Technical Report CSRG-88 Computer Systems Research Group University of Toronto, Canada, January 1978.
- [12] WIJNGAARDEN, A. VAN et al., *Revised Report on the Algorithmic Language ALGOL 68*, Springer 1976.
- [13] WILLIAMS, M.H., *A question-answering system for automatic program synthesis*, SIGPLAN-Notices Vol. 11, No. 7 (July 1978) 63-68.

AN ALGOL 68 PRELUDE FOR THE IMPLEMENTATION OF TEST GENERATION ALGORITHMS

S.D. BUTLAND

ABSTRACT

This paper describes a set of software tools used in the development of algorithms to generate diagnostic test patterns for digital networks.

Data structures, operators and procedures have been defined in ALGOL 68 to facilitate the implementation of test generation algorithms. All internal data organisation is handled by library procedures, and the algorithms themselves operate on conceptual objects rather than on components characteristic of individual logic families. A deliberate attempt has been made to generalise the algorithms and their corresponding data structures, to permit extension to arbitrary elemental logic functions.

The problem of test generation is seen to be one of reverse simulation. Whereas most simulation procedures operate on a stimulus/response mode, test generation procedures define a 'response' - required internal conditions sufficient to propagate a fault or set of faults to the outputs of a network - and then seek to identify input conditions sufficient to generate these internal conditions. Extensive use is made of list processing techniques both in the initial search for paths through which faults are to be propagated, and also to determine the mutual consistency of a given set of proposed internal conditions.

1. INTRODUCTION

The problem of test generation can be stated fairly simply: Given a network of inter connected logic elements, and a set of potential faults associated with each element and with each connection, define a set of input conditions sufficient to test for the presence of each fault. A fault is covered by a set of tests if for at least one input condition generated, its presence will necessarily cause the output function realised by the network to differ from the fault-free network.

In this discussion the following terminology will be used:

element an individual logic node.

gate type a functional unit. The term gate type is used to describe the general structure and behaviour of a class of elements. Simple gate types realise the functions AND NOR EQUIVALENCE etc., but more complex functions can also be realised e.g. flip-flops.

A gate type can also describe the behaviour of a group of elements.

line a connection between elements.

time slice a sequential network is modelled as a cascade of identically structured combinational networks. Each copy of the network describes its state at a separate interval in time (time slice), having both external (primary) inputs and internal inputs (feed-back lines).

See esp. [1] ch. 3.

logic value the basic logic values used will be \emptyset , 1 and x (don't care). However, other values may be defined, e.g. enable and disable logic values are described in terms of a change of state from \emptyset to 1 and are used in describing the behaviour of a flip-flop.

linestate a linestate is a structured object (t, ℓ, v) specifying a time slice t , a line ℓ and a logic value v .

2. THE PROBLEM OF TEST GENERATION

The most successful methods used to solve the problem of test generation have taken the general form:

1. Identify a fault to be detected.
2. Establish internal linestates sufficient to propagate the fault selected to the primary output level.
3. Identify input linestates sufficient to generate the required internal conditions. [1,2,4-6]

The problem of the derivation of test sequences is not a simple one, and entails the solution of many sub-problems. For example

- * the identification of internal conditions sufficient to detect a fault;
- * the derivation of all implications of an arbitrary set of internal conditions;
- * the selection of one of potentially many input patterns sufficient to realise required internal conditions;
- * the representation of a sequential network in which a single elemental fault may occur more than once in a sequence of input patterns;
- * the simulation of the behaviour of a network both fault-free and faulty;
- * the development of diagnostic procedures.

Additional constraints may also be placed on the test generation procedures. For example, it may be required to derive

- * a near-minimal test set
- * tests which explicitly distinguish faults from each other.
- * tests which cover multiple faults.

3. CHARACTERISTICS OF SOFTWARE FACILITIES REQUIRED

A software environment is required in which algorithms can be developed to resolve problems arising in the process of test generation. It would seem to be highly advantageous for the algorithms to operate on conceptual objects rather than on specific and limited hardware representations of network functions. To this end a set of data structures and their associated operators and procedures have been developed in ALGOL 68 with the following objectives:

- * to handle an unlimited set of elemental functions;
- * to avoid any specific reference to the way in which a network is represented in store. Algorithms are designed to operate on objects like gatetypes elements and linestates, and these are delivered by basic procedures when requested. Thus the internal representation of a network, accessed directly by the library procedures, can be altered

without affecting the algorithms themselves.

- * to handle an open-ended set of logic values. 5-[6] and 9-[4,5] value logic systems have already been defined to represent the behaviour of a potentially faulty network. Further work is in hand at the University of Bradford on the analysis of indeterminable logic values in a sequential network [3]. It is anticipated that extensions of multiple-value logic systems may prove a rich area for further development in the context of test generation.

The general relationship between the data specifying a network and an algorithm is given in fig. 1.

It should be noted that the original specification of a network can be given in various formats, and that this data is then structured in a standard data format comprising 2 separate sections:

The gate definition defines the gate types used in the network.

Information about each gate type includes a specification of the characteristics of an element of that type. This includes

- i) the structure of an element (how many inputs, how many outputs, whether an input is triggered by an enable signal etc.)
- ii) the physical properties of an element (delay value)
- iii) the logical function realised by an element (in the form of a truth table).

The network definition defines the constituent elements of a network and their type, the inter-connections, and the primary inputs and primary outputs etc.

4. FUNCTIONS PERFORMED BY THE TEST GENERATION PRELUDE MAGNUM

The prelude MAGNUM has been designed

- * to insulate the programmer designing test generation procedures from both external and internal representations of a network for which tests are to be derived;
- * to provide a set of basic data types, operators and procedures;
- * to permit the possibility of over-riding some of the default operators and procedures.

Each of these features will be discussed in turn.

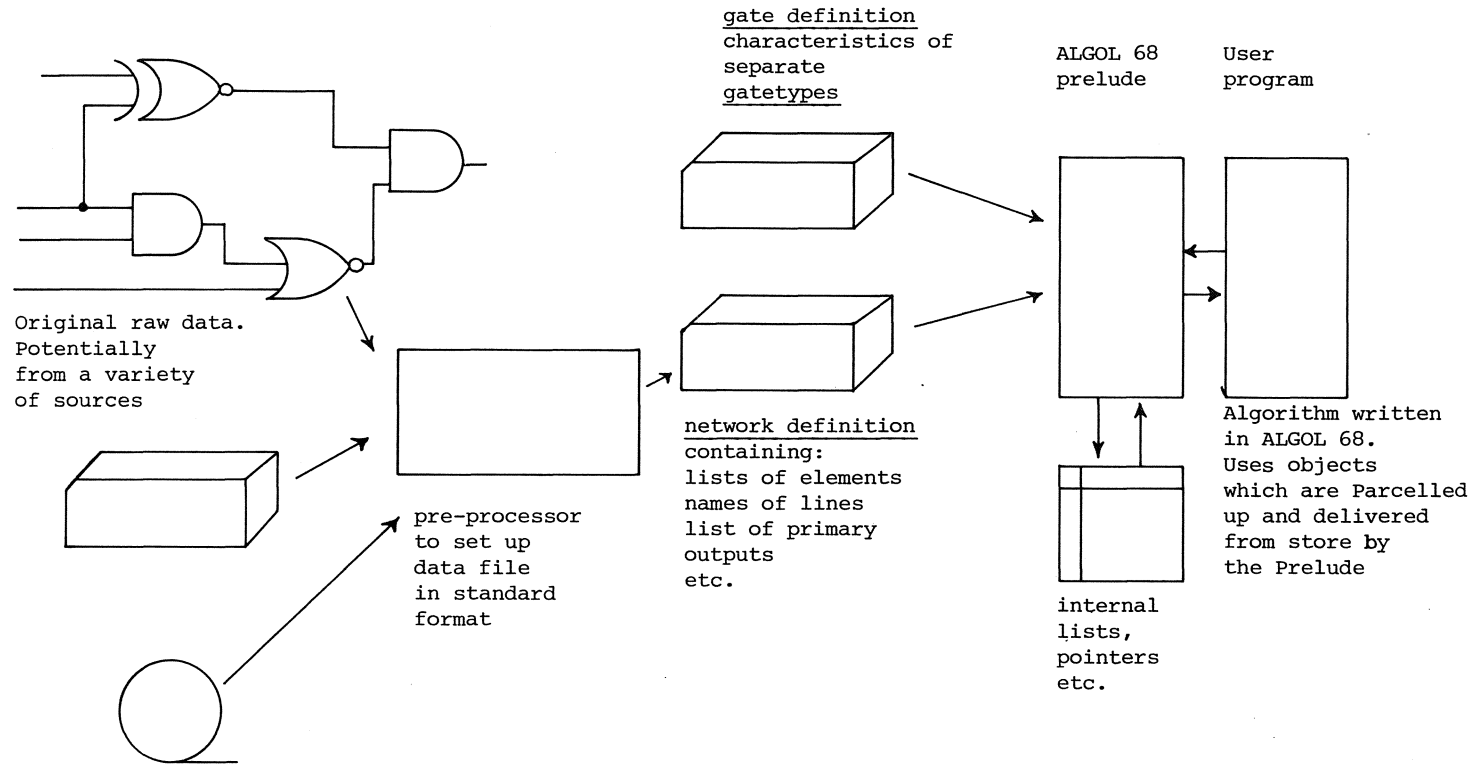


fig. 1 relationship between user program manipulating objects like elements and linestates and their external and internal representation.

4.1. Internal and external representation of data

The code executed by the prelude MAGNUM before entry to the user program reads in the data constituting the gate definitions and the network definition, sets up internal representations of that data, and generates appropriate indexes etc. (c.f. fig.1).

Thus the specification of the n elemental functions used in a network are read in and set up in an array [1:n] gatetype gates, where gatetype is defined as:

```
mode gatetype = struct ([1:12] char c giving a 12 character name c,
    ref [,] int schema c referencing an appropriate truth table c,
    int cycrange c defining the # of time-slices covered by the gate
        type c,
    delay c defining the delay value c,
    nops c # of output waveforms c, .....);
```

Other privately defined selectors describe the structure of the data in the network definition, and the relative time slice for which the logic values are valid (c.f. figs. 2a,2b).

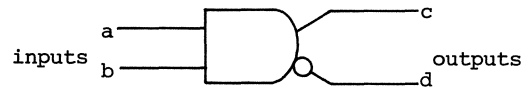
Publicly available variables describing the general characteristics of the data constituting the network under analysis include:

```
int nlines    c # of lines c,
    nels       c # of elements c,
    npops      c # of primary outputs c,
    npips      c # of primary inputs c,
    maxcyics   c max. # of time slices c,
    ....;
```

[1:npops] int pops c a row containing the line numbers of primary outputs c;

[1:npips] int pips c a row containing the line numbers of primary inputs c;

Fig. 2a. a 2-input, AND/NAND gate

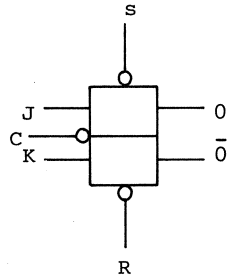


Selector	Reference																
name	"AND/NAND"																
schema	<table border="1"> <thead> <tr> <th>c</th> <th>d</th> <th>a</th> <th>b</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>∅</td> <td>1</td> <td>1</td> </tr> <tr> <td>∅</td> <td>1</td> <td>0</td> <td>x</td> </tr> <tr> <td>0</td> <td>1</td> <td>x</td> <td>0</td> </tr> </tbody> </table>	c	d	a	b	1	∅	1	1	∅	1	0	x	0	1	x	0
c	d	a	b														
1	∅	1	1														
∅	1	0	x														
0	1	x	0														
cycrange	1																
delay	3																
nops	2																

Privately available information includes a specification that each of the logic values is considered to operate over the same time-slice, and the characteristics of each line a - d are:

line	characteristic
a	input
b	input
c	output
d	inverse output

Fig. 2b. a J-K flip flop



Selector	Reference																																																																																								
	"JKFF"																																																																																								
Schema	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>0</th> <th>$\bar{0}$</th> <th>0'</th> <th>J</th> <th>K</th> <th>C</th> <th>S</th> <th>R</th> </tr> </thead> <tbody> <tr><td>\emptyset</td><td>1</td><td>\emptyset</td><td>\emptyset</td><td>\emptyset</td><td>E</td><td>D</td><td>D</td></tr> <tr><td>1</td><td>\emptyset</td><td>1</td><td>\emptyset</td><td>\emptyset</td><td>E</td><td>D</td><td>D</td></tr> <tr><td>\emptyset</td><td>1</td><td>X</td><td>\emptyset</td><td>1</td><td>E</td><td>D</td><td>D</td></tr> <tr><td>1</td><td>\emptyset</td><td>X</td><td>1</td><td>\emptyset</td><td>E</td><td>D</td><td>D</td></tr> <tr><td>1</td><td>\emptyset</td><td>\emptyset</td><td>1</td><td>1</td><td>E</td><td>D</td><td>D</td></tr> <tr><td>\emptyset</td><td>1</td><td>1</td><td>1</td><td>1</td><td>E</td><td>D</td><td>D</td></tr> <tr><td>\emptyset</td><td>1</td><td>\emptyset</td><td>X</td><td>X</td><td>D</td><td>D</td><td>D</td></tr> <tr><td>1</td><td>\emptyset</td><td>1</td><td>X</td><td>X</td><td>D</td><td>D</td><td>D</td></tr> <tr><td>1</td><td>\emptyset</td><td>X</td><td>X</td><td>X</td><td>D</td><td>E</td><td>D</td></tr> <tr><td>\emptyset</td><td>1</td><td>X</td><td>X</td><td>X</td><td>D</td><td>D</td><td>E</td></tr> </tbody> </table>	0	$\bar{0}$	0'	J	K	C	S	R	\emptyset	1	\emptyset	\emptyset	\emptyset	E	D	D	1	\emptyset	1	\emptyset	\emptyset	E	D	D	\emptyset	1	X	\emptyset	1	E	D	D	1	\emptyset	X	1	\emptyset	E	D	D	1	\emptyset	\emptyset	1	1	E	D	D	\emptyset	1	1	1	1	E	D	D	\emptyset	1	\emptyset	X	X	D	D	D	1	\emptyset	1	X	X	D	D	D	1	\emptyset	X	X	X	D	E	D	\emptyset	1	X	X	X	D	D	E
0	$\bar{0}$	0'	J	K	C	S	R																																																																																		
\emptyset	1	\emptyset	\emptyset	\emptyset	E	D	D																																																																																		
1	\emptyset	1	\emptyset	\emptyset	E	D	D																																																																																		
\emptyset	1	X	\emptyset	1	E	D	D																																																																																		
1	\emptyset	X	1	\emptyset	E	D	D																																																																																		
1	\emptyset	\emptyset	1	1	E	D	D																																																																																		
\emptyset	1	1	1	1	E	D	D																																																																																		
\emptyset	1	\emptyset	X	X	D	D	D																																																																																		
1	\emptyset	1	X	X	D	D	D																																																																																		
1	\emptyset	X	X	X	D	E	D																																																																																		
\emptyset	1	X	X	X	D	D	E																																																																																		
<p>where 0 refers to the current output,</p> <p>0' refers to the previous output</p> <p>E refers to 'enable'</p> <p>D refers to 'disable'</p> <p>\emptyset refers to logic \emptyset</p>																																																																																									
cycrange	3																																																																																								
delay	10																																																																																								
nops	2																																																																																								

Privately available information includes a specification of the relative time slice appropriate to each signal and the characteristics of each line:

line	characteristic	time slice
J	input	2
K	input	2
C	trigger input	1/2
S	trigger input	1/2
R	trigger input	1/2
O	output	3
\bar{O}	inverse output	3
0'	previous output	2

4.2. Basic data types, operators and procedures

4.2.1. Data types

In addition to the mode gatetype specified in 4.1, other modes are defined. Some of the more frequently used modes include:

```

mode element = struct
    (int gatetype c defines which element in "gates" defines
        the functional characteristics of the
        element (see 4.1) c,
    ref [ ] int inputs c row of the input lines c,
        outputs c row of the output lines c);

mode linestate = struct
    int cycle c defines time slice c,
        line c defines line number c,
        value c defines logic value c);

```

4.2.2. Procedures and operators

A selection of procedures available in MAGNUM is described here to indicate the manner in which the user program can identify and manipulate objects.

4.2.2.1. Procedures to identify elements

```

proc backref = (int l) int:
    c delivers the element number of which line l is the
        output. The value delivered refers as follows:
    <0 l is an inverse output. abs backref(l) gives the
        element number of which l is the inverse output.
    =0 l is a primary input
    >0 gives the element of which l is the output c

proc frontrefs = (int l) [ ] int:
    c delivers a row of all element numbers fed by l c

proc findel = (int i) element:
    c delivers element i c

```

4.2.2.2. Manipulation of logic values

Multiple value logic systems have been developed independently in the context of test generation in order to describe

- i) the behaviour of a faulty network [4-6].
- ii) internal logic conditions which cannot in principle be determined [3].
- iii) the behaviour of trigger signals in which a change of state is required to activate a logic block (c.f. fig. 2b above).

A software environment designed to facilitate the development of further algorithms must be capable of handling an open ended set of logic values. The prelude therefore represents separate logic values in an n-valued logic system by integers in the range 1-n. It assumes a basic 10-valued logic system, but permits the introduction of other systems (see 4.3 below).

The following basic procedure delivers an appropriate logic value:

```
proc findval = (int t,  $\ell$ ) int: c delivers the logic value of line  $\ell$ 
                        at time t c
```

Values are set using the procedure

```
proc setval = (linestate  $ls$ , int newval) void:
```

where ls gives the current known value of a line at a given time slice, and newval defines the new value to be assigned.

The following procedures are defined to operate on logical values:

```
proc compatible = (int val1, val2) bool:
```

which delivers true if values are consistent with each other and false otherwise.

```
proc cover = (int val1, val2) int:
```

this procedure delivers the value covering val1 and val2, where val1 is thought of as containing val2, but not vice-versa.

```
proc intersect = (int val1, val2) int:
```

this procedure delivers the common value between val1 and val2.

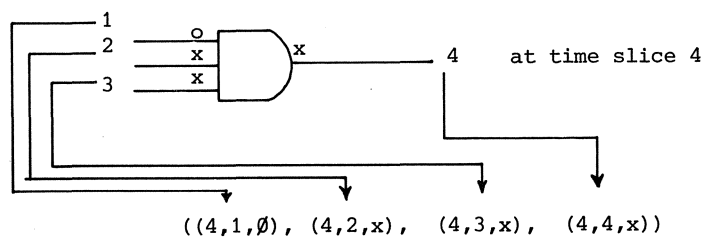
4.2.2.3. Manipulation of functional characteristics

A central procedure FACTOTUM is used which takes as arguments an element, a 'schema (truth table), the time slice corresponding to the output, and a procedure.

The current known state vector

A central item of data used in many distinct operations is a row of linestates, describing the current known state of lines on an element. See fig. 3.

fig.3 Formation of a current known state vector



It should be noted that the number of linestate elements in a current known state vector corresponds to the number of columns in the truth table defining the function of the element.

Factotum forms the current known state vector of the element at the time slice given, and activates the procedure specified with parameters giving the element number, the time slice, the schema and the current known state vector.

The formal specification of factotum is:

```
proc factotum = (element e, [,] int schema, int time,
                proc (int, int, [,] int, [ ] linestate) void f) void:
```

This procedure is used whenever the functional characteristics of an element are to be examined. For example, to determine whether any further implications can be derived from the current known state vector associated with a given element in a given time slice, the truth table associated with that element is examined. All rows covering the logical values so far established are identified.

If there are no such rows, then there exists no pattern of input and output values which are consistent with the values currently holding, and an inconsistent set of linestates has been established within the network. If a set of rows is identified then the values in each of the columns are intersected in turn. If the intersection of these values is non-null, then the common value can be established.

This procedure takes the general form:

```
proc implication = (int time, elno, [,] int schema, [ ] linestate ckls) void:
```

c the following example demonstrates the general characteristics of the procedure making no ostensible use of the parameters "time" and "elno". These are however used to determine whether certain computations have already been performed to eliminate identical computation. Their use would, however, only confuse the essential characteristics of the algorithm:

the truth table "Schema" is examined row by row and compared with the corresponding values established in the current known linestate vector ckls. All rows in schema covering ckls are identified, and these are then examined by columns to identify common values.

c

```
begin
```

```
  [1:upb schema] int common c to hold indices of rows covering ckls c;
```

```
  int ctcom:=  $\emptyset$  c counts # of rows found c;
```

```
  for i to upb schema do c look at each row in turn c
```

```
    bool in:= true c set false if row not covered by schema [i,] c;
```

```
    for j to ckls while in do c examine each known value in turn c
```

```
      in:= consistent(val of ips[j], schema[i, j])
```

```
    od c 'in' is set false if any value not consistent in row c;
```

```
    if in then common [ctcom plusab 1]:= i fi
```

```
  od;
```

```
  if ctcom /=  $\emptyset$  then
```

```
    for j to upb ckls do c examine columns of each possible row c
```

```
      int comval:= schema[com[1], j];
```

```
      for i from 2 to ctcom do
```

```
        comval:= intersect(schema[com[i], j], comval)
```

```
      od;
```

```
      if comval /= logx then setval(ckls[j], comval) fi
```

```
    od
```

```
  else
```

```
    inconsistent value found
```

```
  fi
```

```
end;
```

This procedure is involved for element *e* at time slice *t* by
 factotum(*e*, schema of gates[type of *e*], *t*, implication)

4.3. Extensions to further logic values

The prelude MAGNUM has been designed with a view to being used in the development of further test generation algorithms which may require an extension or complete replacement of the logic values used in the current system.

Some of the internal procedures themselves manipulate logic values, so an environment is required in which the default logic handling procedures can be replaced by new ones. This is ensured by designing the internal procedures so that they use ref proc s rather than procedures themselves. The user may therefore design his own procedures to use an original set of logic values, and may reset the default procedures to those of his own. Examples of how new logic values may be introduced are given below:

Publicly accessible procedures

```
proc compatible = (int val1, val2) bool:
proc cover = (int val1, val2) int:
proc intersect = (int val1, val2) int:
```

These procedures described in 4.2.2.2 above define the relationship between pairs of logic values. The procedures used by internal processes are thus defined:

```
proc (int, int) bool compat; compat:= compatible;
proc (int, int) int cov; cov:= cover;
proc (int, int) int inter; inter:= intersect;
```

These default settings can be over-ridden by statements of the form:

```
compat:= new proc1;
cov:= new proc2;
inter:= new proc3;
```

Publicly accessible procedure:

```
proc valch = (int val) [char]:
```

delivers a row of characters by which a value is represented on output.
 The characters delivered by default correspond to those given in 4.2.2.2.

The procedure used by internal processes is:

```
proc (int) [] char vrep;      vrep:= valch;
```

If different values are to be represented, these may be handled by defining a new procedure to deliver a row of characters, and obeying an assignment of the form:

```
vrep:= new procedure;
```

Publicly accessible procedure:

This procedure takes as a parameter an object of mode pointers which defines the state of various lists before the latest set of assignments have been made. It works through these lists, determining the consistency (or otherwise) of the assignments just made with each other and with other conditions already holding in the network.

"Consistent" is set to an internal procedure which handles logic values currently in use. New consistency procedures can be defined, and the assignment

```
consistent:= new consistency procedure
```

will ensure that a new procedure will be used.

Default procedure	Internally referenced procedure/s	Comment
compatible cover intersect (see 4.2.2.2 above)	<pre> proc (int, int) bool compat; proc (int, int) int cov; proc (int, int) int inter; </pre>	The relationship between any pair of logic values, represented by discrete integers, can be re-defined, and the default procedure will be over-ridden by a statement of the form: <pre> compat:= new proc1; cov:= new proc2; inter:= new proc3; </pre>
<pre> proc valch = (int v) []char: </pre> <p>this delivers a row of characters by which a value is represented on output. The characters delivered correspond to 0, *1 Ø(D) etc.</p>	<pre> proc (int) []char vrep; </pre>	If different values are to be represented, these may be handled by defining a new procedure to deliver a row of characters and an assignment of the form <pre> vrep:= new procedure </pre>

Other characteristics of ALGOL 68 used in the process of test generation

The above example demonstrates the facility within ALGOL 68 to structure data in a form appropriate to an algorithm, and to provide basic procedures to manipulate such data. These characteristics are common to many

other applications, but other features of the language make it singularly appropriate as a vehicle for the solution of problems relating to test generation.

Recursion Many of the sub-problems to be solved in the general context of test-generation are much simplified if recursive facilities are used.

For example, the procedure to search backwards through the network for a previously untested fault entails establishing a proposed set of conditions at a node (an element), and then examining other elements feeding that node. If inconsistent values are postulated at any stage of the search, or if a path is abandoned, the network must be set to the state it was in before establishing the rejected path. This process is much simplified if a recursive procedure is used, where each level of recursion contains its own information about the state of the network on entry.

List processing

When a path is traced through the network, each further extension may require the specification of a set of linestates whose length is unknown. This information is most conveniently represented as a linked list, thus avoiding the need to set an upper limit on the number of implications which may be established for any path section.

Bit manipulation

While the internal structure of data held in store is of no public concern, information can be packed in bit form and accessed by the prelude. This preserves the generality of the facilities offered, since they are all at the 'object' level, but also offers the possibility of efficient data storage where this is a major problem.

Summary

A set of modes and procedures have been developed in ALGOL 68 to permit the development of test generation algorithms. A deliberate attempt has been made to provide facilities which are not specifically tied to any hardware representation of elemental functions, and to be capable of handling an unlimited range of logic values. The functional characteristics of elements are described externally to the library in the form of a truth table. This offers complete generality, since both combinational and sequential elemental types can be handled, but only at the expense of processor time.

The real advantage of this approach is seen in its extension to complex elemental functions, where the behaviour of a group of elements can be described and manipulated as a single functional unit.

It is anticipated that these procedures can be used to design and develop alternative test generation strategies, with a view to comparing their effectiveness for different structured networks.

REFERENCES

- [1] BREUER, M.A. & A.D. FRIEDMAN, *Diagnosis and Reliable Design of Digital Systems*, Pitman 1977.
- [2] BUTLAND, S.D., *Determining the mutual consistency of internal line-states within a network*, Computers and Electrical Engineering, Vol. 6 pp. 69-78, 1978.
- [3] BUTLAND, S.D., *Indeterminable Logic States in the Simulation of Digital Networks*, To be presented at United Kingdom Simulation Council Conference on Computer Simulation, Harrogate, May 1981.
- [4] CHA, C.W., W.E. DONATH & F. ÖZGÜNER, *9-v Algorithm for Test Pattern Generation of Combinational Digital Circuits*, IEEE Transactions on Computers, Vol. C-27 No. 3 March 1978, pp. 193-200.
- [5] MUTH, P., *A 9-Valued Circuit Model to Generate Tests for Sequential Circuits*, Presented at Symp. Fault-Tolerant Computing, June 1975, Paris.
- [6] ROTH, J.P., et al., *Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits*, IEEE Transactions on Computers, Vol. C-16 pp. 567-579, Oct. 1967.
- [7] THOMAS, J.J., *Authorised diagnostic test programs for digital networks*, Computer Design pp. 63-67 Aug. 1971.

A PROGRAMMING SYSTEM FOR INTERVAL ARITHMETIC IN ALGOL 68

G. GÜNTHER & G. MARQUARDT

ABSTRACT

The problem of error bounds in digital computation is solved here by using interval arithmetic. A short description of the underlying theory is given. It is shown how easily the interval arithmetic can be implemented in ALGOL 68 yielding an efficient and easy to use interval programming system for real values. Two extensions of this package are presented, a complex interval arithmetic and a system with vector and matrix operations (TORRIX-INTVAL).

1. INTRODUCTION

One possibility of determining the error in digital computation is the application of interval analysis. Instead of operating with real numbers, intervals are used. They are represented by two real numbers, a lower and an upper bound for the exact value. By this method different kinds of errors can be taken into account: roundoff errors, errors in conversion of real numbers, uncertainty in input data, inaccuracies in mathematical formulas. Some important properties of interval arithmetic will be outlined, more detailed information can be found in [1] or [2].

The set of real interval numbers is

$$I(\mathbb{R}) \equiv \{[a, b] \mid a \leq b, a \in \mathbb{R}, b \in \mathbb{R}\}.$$

The basic arithmetic operations between intervals are defined in the following way:

Let $\otimes \in \{+, -, *, /\}$, then

$$(1.1) \quad [a, b] \otimes [c, d] = \{x \otimes y \mid x \in [a, b], y \in [c, d]\}$$

with $0 \notin [c, d]$ for the division.

If $x \in [a, b]$ and $y \in [c, d]$ are the two exact but unknown real numbers, then $[a, b] \otimes [c, d]$ contains the exact result $x \otimes y$, as can be shown.

When the real variables of a real-valued function f are replaced by intervals and the real operations are replaced by the corresponding interval operations, one gets the interval extension F of f . The relation

$$F([a, b]) \supseteq \{f(x) \mid x \in [a, b]\} \text{ holds.}$$

If f is defined and continuous on $[a, b]$, then $F([a, b])$ will again be an interval.

For practical computation rules derived from definition (1.1) are more useful than the definition itself:

$$(1.2) \quad [a, b] + [c, d] = [a + c, b + d]$$

$$(1.3) \quad [a, b] - [c, d] = [a - d, b - c]$$

$$(1.4) \quad [a, b] * [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$$

$$(1.5) \quad [a, b] / [c, d] = [a, b] * [1/d, 1/c].$$

The operation is undefined for $0 \in [c, d]$.

The implementation of the operators $*$ and $/$ can be simplified by examining the signs of the endpoints. Details of this case analysis can be found in [3].

One problem in interval analysis is the so-called dependency of intervals. For example, in real arithmetic, $x * x = x^2$, but this is not true for intervals: if $X = [-2, 2]$, then $X * X \equiv \{x * y \mid x \in X, y \in X\} = [-4, 4]$ and $X^2 = \{x^2 \mid x \in X\} = [0, 4]$. In cases like this, it is comparatively easy to eliminate dependency, but in most cases it is not, for instance, computing $X * \sin(X)$, where X is a small interval containing 0.

Since the operations in (1.2) - (1.5) contain real (infinite-precision) arithmetic, they cannot be implemented on a digital computer. Thus it is necessary, to round correctly the results obtained by the common computer arithmetic. The optimal rounding procedure maps the left endpoint x to the smallest machine representable real number less than or equal to x (∇x) and the right one y to the largest machine representable number greater than or equal to y (Δy). So the final result is guaranteed to contain the exact real arithmetic result. Consequently, for example, the formula (1.2) would be modified to

$$[a, b] + [c, d] = [\nabla(a + c), \Delta(b + d)]$$

(A complete discussion about different kinds of rounding may be found in KULISCH [4].)

Due to these problems, the interval equivalent of a stable real algorithm might eventually be unstable in the sense, that the result intervals may become unacceptably wide. But for a lot of problems stable interval algorithms have been constructed.

There are different implementations of interval arithmetic. Two of them are the TRIPLEX-ALGOL-Compiler [3], developed at the University of Karlsruhe, and a FORTRAN subroutine package [5], developed at the University of Wisconsin, compatible to the AUGMENT precompiler [6].

2. AIMS

Our aim is to develop an easy to use programming system for interval

arithmetic. It is desirable for the user to handle the interval objects in the same manner as the integer or real objects. Thus an interval program must have the same structure as any other program except that the data type real is substituted by the data type for intervals. For practical applications it is desirable that the following features should be implemented:

- (i) A data type for intervals (intval)
- (ii) The operator symbols +, -, *, /, ** for the interval arithmetic
- (iii) The relational and logical operators for intervals, e.g.
=, ≠, <, ≤, >, ≥, disjct, c, ε
- (iv) special operators, e.g. width, halfwidth, sup, inf, mid, ∩, ∪
- (v) standard functions, e.g. sin, cos, exp, sqrt
- (vi) denotations for interval constants
- (vii) special interval constants, e.g. intvalpi, maxintval
- (viii) input-output routines for interval objects

3. REALIZATION

Inspired by the richness and flexibility of ALGOL 68, we decided to use ALGOL 68 as implementation language for our interval system.

Particularly the following features of ALGOL 68 support the implementation:

- (i) The data type (mode) intval can be defined by the mode declaration:

mode intval = struct(real inf, sup)
- (ii) ALGOL 68 allows the definition of operators and the use of operator symbols, e.g. +, -, *, /, **, for different data types and also between them in mixed mode expressions. By that the items (ii) - (iv) of the general requests in section 2 can be fulfilled.
- (iii) Aided by the orthogonality of ALGOL 68 we can realize easily most of the other requests, e.g. functions and procedures for intervals can be defined, the structure display can be used as denotation for intval constants, special constants can be defined, each interval object can be handled as a whole, and in addition the parts can be selected. Thus the remaining requests are satisfied.
- (iv) ALGOL 68 is extensible by means of library preludes. In consequence our work to develop an interval system in ALGOL 68 was

reduced to the creation of an interval library prelude.

4. PROBLEMS AND REQUIREMENTS

The shortcomings of our system are caused by the fact that the mode intval is user defined and not a standard mode like the modes real, int, compl. Therefore some operations which have to be done at compilation time cannot be implemented. The main deficiencies are:

- (i) No automatic widening from real to intval. Thus intval $i := 1.0$ is not allowed. We have to use a special operator:

```
intval i := widen 1.0
```

or an intval denotation

```
intval i := (1.0, 1.0)
```

- (ii) No correct conversions of intval denotations. The statement

```
i := (0.1, 0.1)
```

is handling a point interval, which in this case does not contain the proper value because of the internal representations of real numbers. Thus it is necessary to have special conversions for the lower and upper bounds of intval denotations during compilation time.

We can circumvent this problem by means of string denotations:

```
i := widen "0.1"
```

because it is possible to convert the string "0.1" correctly by a runtime routine. But this solution seems too clumsy to us, and is not implemented in our system.

- (iii) No correct conversion from real to intval objects. The mixed mode operations between intval and real objects, e.g. in the expression

```
i + 0.1 (i to be an intval variable)
```

cannot be compiled correctly, because it is not possible to convert 0.1 to the corresponding interval value during compilation.

A circumvention is rounding the bounds down respectively up to the nearest representable machine number. This guarantees that

the correct value is included in the interval.

The rounding operations can be easily realized in ALGOL 68, as shown later. In our system these operations are implicitly performed for the mixed mode operations between intval and real objects and for the operators widen and comp (compose) applied to real objects. We can explicitly use the operations by the operators up and down. (See appendix)

- (iv) No appropriate transput routines for intervals.

Because an intval object is composed of two floating-point numbers and because the transput modes simplin and simplout are not extendable by the user, the standard transput routines "read" and "print" transmit two real numbers without any special conversion for the lower and upper bounds. For output it would be more readable if intval numbers were printed with special delimiting characters like square brackets:

e.g. [+ 1.998, + 2.004]

In correspondence to that the intval numbers should be marked with brackets for input, too:

input data: [1.0], [-10.0, + 10.0]

The formatted transput also needs a special "format pattern" for intval objects. To overcome that we shall introduce special routines like "inintval" and "outintval".

- (v) A minor deficiency is that the identifiers for the standard functions like sqrt, sin, cos etc. have to be changed to intvalsqrt, intvalsin, intvalcos etc. for interval arguments.
- (vi) In addition, some extensions of the arithmetic in ALGOL 68 would be helpful and would facilitate the implementation of interval systems:

- a concept of exception handling (invalid operation, underflow, overflow),
- a better way to define the precision of floating-point numbers,
- special values (invalid numbers, infinities).

The other implementation-problems are dependent on the hardware. At the moment the floating-point arithmetic on different machines is so varying that it is impossible to develop a fully portable interval package. The

problems of numerical computations are explained in detail by KULISCH [4]. To permit the implementation of both rigorous and tight bounds of the intervals it is necessary to realize the operations of the directed rounding towards $\pm \infty$. These rounding operations are also proposed in the IEEE floating-point standard [10] in addition to the usual rounding operations like "rounding to the nearest" and "rounding to zero" (truncation).

On the other hand, the interval arithmetic seems so important to us that the computer manufacturers should realize the interval operations directly by hardware. The costs for such arithmetic units would be low, considering the present hardware prices.

5. IMPLEMENTATION

In our implementation for CDC CYBER machines we have simulated the optimal directed rounding by taking advantage of the fact that on the CYBER machine the result of a floating-point operation is available in double precision. The implementation of the arithmetic interval operations like $+$, $-$, $*$, $/$, $**$ is based on the routines of WIPPERMANN [3] developed for the TRIPLEX arithmetic, for the standard functions like `intvalsin`, `intvalexp` etc. we took the proposals of HERZBERGER [7].

An advantage of the CDC CYBER ALGOL 68 compiler is the possibility to define inline operators, so that we can directly define which machine instructions have to be generated by the code generator. The usage of inline operators is a very powerful tool for efficient code generation and a way to define operations not expressible in ALGOL 68.

On the other hand, this implementation is machine dependent. Another possibility would be to take the algorithms of ALEFELD & HERZBERGER [1] to implement the basic machine interval arithmetic operations. They are depending on the kind of rounding used and on the behaviour in the neighbourhood of zero. This implementation would be fully transportable, but the resulting interval bounds are not the best possible approximations.

The algorithms used are based on the operators up and down. The transportability is guaranteed by the fact that they can be expressed in ALGOL 68: e.g.

up of a real positive number x is defined by

op up = (real x) real: (1 + smallreal) * x

6. EXTENSIONS

6.1 Complex-valued interval arithmetic

Since it is possible in ALGOL 68 to extend existing preludes by definitions of new data types, a complex error arithmetic can easily be implemented as extension of our INTVAL system. It was done by using the circular arithmetic of GARGANTINI & HENRICI [11]. A complex interval in that arithmetic is represented by a circle, i.e. its midpoint and its radius.

Concerning the precision, the user can select between three preludes for this arithmetic. The first prelude (standard) contains the center (xm, ym) in double precision and the radius in single precision, the second one uses only single precision, the third one contains both variables in double precision.

The first (standard) prelude is the one regarded in the next section.

For the implementation of the circular arithmetic a real interval arithmetic in double precision was developed. The corresponding data type of an interval of that kind is defined as

```
mode longintval = struct(long real a, b)
```

In this case, simple rounding procedures (up, down) must be used because of hard-ware properties.

The complex interval describes a disk. Its data type is

```
mode disk = struct(long real xm, ym, real rad)
```

with the center (xm, ym) and the radius rad.

The definitions of the arithmetic operators are known from [11]. The result Z of an arithmetic operation between two disks K_1 and K_2 is performed in the following way: the centers of K_1 and K_2 are regarded as (point) long real intervals and the appropriate long interval operations are applied. The center of Z is then composed of the midpoints of the two intervals and their widths are added to the radius of Z. The implementation of the complex standard functions is easily done, because the programs are known from [12].

6.2 TORRIX-INTVAL

An extension of the system in a different respect is to complement it by vector and matrix operations.

As a basis we can use TORRIX, a programming system for operations on vectors and matrices over arbitrary fields, developed by VAN DER MEULEN & VELDHORST [8]. TORRIX can be implemented as a library prelude in ALGOL 68. In the standard system the underlying scalar system is set to the mode real.

Our next project will be to develop a TORRIX-INTVAL system. The transformation can easily be done by defining

```
mode scal = intval
```

and then recompiling the TORRIX routines with the INTVAL system.

7. EXAMPLES

The first example is intended to demonstrate how easily a program can be transformed from a real to an intval version.

real version:

```
begin
  comment
    pi, as computed by the archimedean algorithm: approximation of the
    perimeter of a circle by inscribed and circumscribed n-cornered
    regular polygons comment
  int n, k:= 0;
  real perout, perin, piout, piin, s, side;
  print((newpage, " RESULTS:", newline, " ", 8 * "*", newline, newline,
    8 * " ", "N", 4 * " ", "PI - PIIN", 4 * " ", "PI - PIOUT",
    newline, 8 * " ", 30 * "=", newline, newline));
  real r = 3.0; real r2 = 2 * r, rr = r ** 2;
  side:= r * sqrt(3.0);
  to 15 do
    k += 1;
    n:= 3 * 2 ** k;
    s:= side * 0.5;
    side:= sqrt(2 * rr - r2 * sqrt(rr - s ** 2));
```

```

perin:= n * side;
perout:= perin / (sqrt(1 - (side / r2) ** 2));
piin:= perin / r2;
piout:= perout / r2;
print((4 * " ", whole(n, -6), 3 * " ", float((pi - piin), 10, 3, 3),
      4 * " ", float((pi - piout), 10, 3, 3), newline))
od
end

```

RESULTS:

N	PI - PIIN	PI - PIOUT
=====		
3072	+5.442E -7	-1.099E -6
6144	+1.221E -7	-2.886E -7
12288	-2.968E -8	-1.324E -7
24576	-2.573E -7	-2.830E -7
49152	-1.472E -6	-1.478E -6

This program has been an exercise for students. They were surprised to notice that the algorithm is not convergent to "pi". The cause for this is obviously the propagation of rounding errors.

The transformation to the intval version is accomplished by replacing the modes of the variables and applying the appropriate standard function. Furthermore the widen operator has to be inserted.

Intval version:

```

begin
  comment
    pi, as computed by the archimedean algorithm: approximation of the
    perimeter of a circle by inscribed and circumscribed n - cornered
    regular polygons comment
  int n, k:= 0;
  intval perout, perin, piout, piin, s, side;
  intval solnew, solold:= maxintval;
  intval r = widen 3.0; intval r2 = 2 * r, rr = r ** 2;
  side:= r * intvlsqrt(widen 3.0);
  while
    k += 1;

```

```

n:= 3 * 2 ** k;
s:= side * 0.5;
side:= intvlsqrt(2 * rr - r2 * intvlsqrt(rr - s ** 2));
perin:= n * side;
perout:= perin / (intvlsqrt(1 - (side / r2) ** 2));
piin:= perin / r2;
piout:= perout / r2;
solnew:= solold intsct (piin u piout);
solold ne solnew
do
  solold:= solnew
od;
k:= 3 * 2 ** (k - 1);
print((newline, "RESULT:", newline, " ", 7 * "*", newline, newline,
" N = ", whole(k, -6), newline,
" BEST INTERVAL = [", inf of solnew, " ", " ", sup of solnew, "]",
newline, newline))
end

```

```

RESULT:
*****

```

```

N = 6144

```

```

BEST INTERVAL = [+3.1415924745664E +0 , +3.1415929990960E +0]

```

As opposed to the real version, it is now quite simple to find a stopping criterion.

The second example shows how to combine real and interval algorithms:

```

begin
# NEWTON - ALGORITHM:  $X_{N+1} = X_N - F(XN) / \text{DERIV}(XN)$  #
proc f = (real x) real: (real y = x * x; x - (1 - y) / (3 + y));
proc if = (intval x) intval: (intval y = x ** 2; x - (1 - y) / (3 + y));
proc deriv = (real x) real: (real y = 3 + x * x; 1 + 8 * x / (y * y));
proc ideriv = (intval x) intval: (intval y = 3 + x ** 2; 1 + 8 * x / y ** 2);
real x, x1;
real x0 = 0.0, eps = 1.0e-7, int max = 20;
print((" RESULTS:", newline, " ", 8 * "*", newline, newline,
" STARTING VALUE X0 = ", fixed(x0, 5, 2),
" EPS =", float(eps, 8, 1, 3), newline, newline,

```

```

" STEP", 10*" ", "X", 25 * " ", "F(X)", newline, " ", 52 * "-",
newline));
x:= x0;
for m while
  x1:= x;
  real d1 = deriv(x1); if abs d1 < eps then stop fi;
  x:= x1 - f(x1) / d1;
  print((" ", whole(m, -2), 6 * " ", x, 3 * " ", f(x), newline));
  abs (x - x1) > eps and m < max
do skip od;

#LAST STEP WITH INTERVALS#
intval ix:= (x, x); intval id1 = ideriv(ix);
ix -=: if(ix) / id1;
print((newline, "INTERVAL STEP: ", newline,
      "IX   = [", inf ix, ",", sup ix, "]", newline,
      "F(IX) = [", inf if(ix), ",", sup if(ix), "]",
      newline, newline))

end

RESULTS:
*****
STARTING VALUE X0 = +0.00          EPS =+1.0E -7

STEP          X                      F(X)
-----
 1      +3.333333333333333E -1      +4.7619047619047E -2
 2      +2.960000000000000E -1      +5.0211425254965E -4
 3      +2.9559779074107E -1      +6.0181692518313E -8
 4      +2.9559774252209E -1      +1.7763568394003E -15

INTERVAL STEP:#
IX   = [+2.9559774252208E -1, +2.9559774252209E -1]
F(IX) = [-3.5527136788005E -15, +1.7763568394003E -15]

```

The intention of this combination of algorithms is to get an approximate solution in real arithmetic and to determine or improve it's exactness by interval arithmetic steps, until tight bounds for the exact solution are found. In this way, execution time can be reduced as opposed to an algorithm

using only interval operations, while the interval arithmetic precision is preserved.

8. CONCLUDING REMARKS

Summarizing our experience we can state that we have found in ALGOL 68 a tool well suited for the implementation of an efficient and easy to use interval programming system without great expenditure. Nevertheless, in order to be more helpful for the user, it should be accompanied by a program library containing interval algorithms for all important applications.

APPENDIX

Operators and Procedures implemented in the Interval Package

Arithmetic operators

name	mode of arguments	priority	result mode
+	(<u>intval</u> , <u>intval</u>)	6	<u>intval</u>
	(<u>real</u> , <u>intval</u>)		
	(<u>intval</u> , <u>real</u>)		
	(<u>int</u> , <u>intval</u>)		
-	(<u>intval</u> , <u>intval</u>)	10	<u>intval</u>
	like +		
*, /	like +	7	
** <u>,</u> <u>up</u>	(<u>intval</u> , <u>int</u>)	8	
+:=, <u>plusab</u>	(<u>ref intval</u> , <u>intval</u>)	1	<u>ref intval</u>
	(<u>ref intval</u> , <u>real</u>)		
	(<u>ref intval</u> , <u>int</u>)		
-:=, <u>minusab</u>	like +:=	1	
*:=, <u>timesab</u>	like +:=	1	
/:=, <u>divab</u>	like +:=	1	

Relational operators

name	mode of arguments	priority	result mode	meaning
<, lt	(<u>intval</u> , <u>intval</u>)	5	<u>bool</u>	[a1, a2] < [b1, b2]
	(<u>intval</u> , <u>real</u>)	5		a2 < b1
	(<u>real</u> , <u>intval</u>)	5		
>, gt	like <	5		[a1, a2] > [b1, b2] a1 > b2
/=, ne	like <	4		
=, eq	like <	4		

Interval standard-functions

name	mode of argument	result mode
intvalsln	<u>intval</u>	<u>intval</u>
intvalcos		
intvalsqrt		
intvalexp		
intvalln		
intvalarctan		
intvalacrsin		
intvalarccos		

Special interval operators

name	mode of arguments	prio.	result mode	meaning
sup	(<u>intval</u>)	10	<u>real</u>	right endpoint of the interval
inf	(<u>intval</u>)	10	<u>real</u>	left endpoint of the interval
mid	(<u>intval</u>)	10	<u>real</u>	midpoint: (<u>sup</u> X - <u>inf</u> X) / 2, rounded up
width	(<u>intval</u>)	10	<u>real</u>	<u>sup</u> X - <u>inf</u> X, rounded up
halfwidth	(<u>intval</u>)	10	<u>real</u>	(<u>sup</u> X - <u>inf</u> X) / 2, rounded up
bad	(<u>intval</u>)	10	<u>bool</u>	<u>inf</u> X > <u>sup</u> X
ok	(<u>intval</u>)	10	<u>bool</u>	<u>inf</u> X ≤ <u>sup</u> X
elem	(<u>real</u> , <u>intval</u>)	7	<u>bool</u>	element of X
subset	(<u>intval</u> , <u>intval</u>)	7	<u>bool</u>	X contained in Y
mag	(<u>intval</u>)	10	<u>real</u>	magnitude: <u>sup</u> (<u>abs</u> (x))
mig	(<u>intval</u>)	10	<u>real</u>	magnitude: <u>inf</u> (<u>abs</u> (x))
round	(<u>intval</u>)	10	<u>int</u>	midpoint, rounded up
entier	(<u>intval</u>)	10	<u>int</u>	entier(midpoint)
abs	(<u>intval</u>)	10	<u>intval</u>	[<u>min</u> <u>abs</u> x, <u>max</u> <u>abs</u> x]
intsct	(<u>intval</u> , <u>intval</u>)	9	<u>intval</u>	set-theoretic intersection of X and Y
disjct	(<u>intval</u> , <u>intval</u>)	9	<u>bool</u>	disjunction
u	(<u>intval</u> , <u>intval</u>)	8	<u>intval</u>	union of X and Y
up	<u>real</u>	10	<u>real</u>	rounds up to the nearest larger machine number
down	<u>real</u>	10	<u>real</u>	rounds down to the nearest smaller machine number
point	<u>real</u>	10	<u>intval</u>	[x, x]
widen	<u>real</u>	10	<u>intval</u>	[<u>down</u> x, <u>up</u> x]
widen	<u>int</u>	10	<u>intval</u>	[x, x]
comp	(<u>real</u> , <u>real</u>)	9	<u>intval</u>	compose: if the operands are real, then <u>up</u> , <u>down</u> is applied
	(<u>int</u> , <u>int</u>)			
	(<u>real</u> , <u>int</u>)			
	(<u>int</u> , <u>real</u>)			

ACKNOWLEDGEMENT

We wish to express our thanks to Dipl.-Math. J. Dehnhardt from the University of Hannover for his helpful comments on our concepts, and to G. Mensching, who did part of the programming and testing of the system.

REFERENCES

- [1] ALEFELD, G. & J. HERZBERGER, *Einführung in die Intervallrechnung*, 1974.
- [2] MOORE, R.E., *Interval analysis*, 1966.
- [3] WIPPERMANN, H.-W., *Realisierung einer Intervall-Arithmetik in einem ALGOL 60 System*, *Elektronische Rechenanlagen* 9 (1967), H. 5, p. 224-233.
- [4] KULISCH, U., *Grundlagen des Numerischen Rechnens*, Reihe Informatik/19, 1976.
- [5] YOHE, J.M., *Software for interval arithmetic: A reasonably portable package*, *ACM Trans. on Math. Software*, vol. 5, no. 1, March 1979.
- [6] CRARY, F.D., *A versatile precompiler for nonstandard arithmetics*, *ACM Trans. on math. software*, vol. 2, no. 2, June 1979.
- [7] HERZBERGER, J., *Intervallmässige Auswertung von Standardfunktionen in ALGOL 60*, *Computing* 5, 1970, p. 377-384.
- [8] VAN DER MEULEN, S.G. & M. VELDHORST, *TORRIX vol. 1*, Mathematical Centre Tracts 86, Amsterdam 1978.
- [9] CURNOW, H.J. & B.A. WICHMANN, *A synthetic benchmark*, *Computer Journal*, 1976, vol. 19, no. 1, p. 43-49.
- [10] COONEN, J. et.al. *Proposed IEEE floating-point standard*, *ACM Signum Newsletter*, Special issue, October 1979.
- [11] GARGANTINI, I. & P. HENRICI, *Circular arithmetic and the determination of polynomial zeros*, *Numer. Mathematik* 18 (1972), p. 305-320.
- [12] BÖRSKEN, N.C., *Komplexe Kreis-Standard-Funktionen*, *Freiberger Intervall-Berichte* 78/2.

TEACHING WITH ALGOL 68, IN MANCHESTER

C.H. LINDSEY

ABSTRACT

If approached with care, ALGOL 68 provides an excellent vehicle for teaching the art of programming. Students should be encouraged to think about the visual shape of the language, of their programs and of their data, and suitable visual aids for all of these must be provided. Copious sample programs should be exhibited, to illustrate all the paradigms which are the programmer's stock-in-trade.

There are some specific features of ALGOL 68 which have traditionally been regarded as "difficult", for example, variable-declarations, names, subnames, formal- and actual-declarers, and scope. The answer to these problems lies in a careful choice of the order in which language features are introduced, the particular options which are recommended for use, and the models of behaviour with which they are explained. There are also some features of the language which ought not to be explained at all. The paper contains specific proposals in all these areas.

1. INTRODUCTION

ALGOL 68 is an excellent vehicle for teaching programming because it contains nearly all the tools which a student should know about, and it provides them in a way which encourages a well-structured style of programming. Even if he never uses ALGOL 68 in the outside world, he will still have learnt much, and be a better programmer as a result of it.

Programming is an art. And teaching the art of programming is yet another art. And the chief difficulty of that art lies in understanding why the students seem to be so stupid. If the teacher can get inside the student's mind, he will find many things that surprise him - for example that the uninitiated student does not naturally think recursively, or even orthogonally, and that for these reasons he would far rather learn BASIC whose concepts, in the short term, are so deceptively simple. Again, the concepts of an orthogonal language are so neatly interrelated that there is no starting point whereat teaching may begin, and writing even the simplest program involves, in principle, nearly every concept in the language. Nevertheless, the rate at which students can absorb new concepts is limited, and really rather low, and concepts must therefore somehow be packaged up into small doses.

In teaching programming, there are essentially three things to be taught:

- 1) Program structure
or how to view programs as hierarchies of refinements.
- 2) Paradigms
or how to turn real-world problems into program structures.
- 3) Programming languages
or how to turn program structures into programs.

Inevitably, teaching the chosen programming language is going to take up most of the actual time; nevertheless, it is the least important of the three. I shall therefore consider these topics in the order given but, as inevitably, it will be number 3 that will occupy the bulk of this paper.

2. PROGRAM STRUCTURE

It came as a distinct surprise to me to discover, some years ago, that computer scientists come in two varieties - the "Verbalizers" and the "Visualizers".

The verbalizers solve every problem by inventing a name for it, and invent an implementation of the name later. So, they write their programs with great numbers of procedures, whose bodies are mostly calls on other procedures.

The visualizers, on the other hand, delight in drawing little boxes and depicting relationships or movements between them. Their programs consist of very long procedures, indented almost out of sight into the right-hand margin.

To find largest element of array a

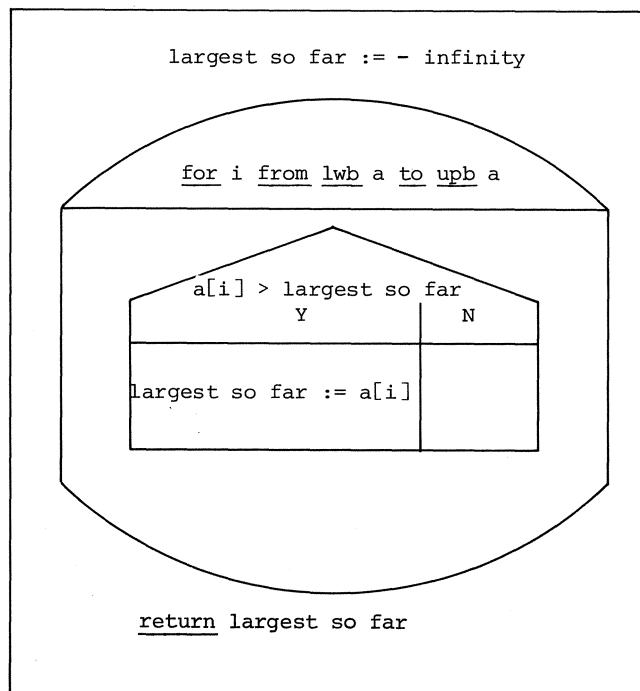


Fig. 1 Example of a Structure Chart.

I am an unashamed visualizer. Hierarchical program structures need to be diagrammed. They can of course be drawn as tree structures, but I prefer a contour model which I call "Structure Charts" [4]. Fig. 1 shows an example of such a chart. There are different shapes of box for the three customary program compositions, and each corresponds to a particular type of ENCLOSED-clause in ALGOL 68. Fig. 2 shows this correspondence for the choice-box. Observe that the conditional-clause comes complete with indentation. Indentation is not optional. The students are taught that this is how a conditional-clause is written, and it is only later that they discover that the compiler doesn't really care. The visual impact of a well-indented program can tell you as much about how it is meant to work as the actual keywords used. In case the students contrive to get their indentation wrong, they are provided with an indenting program.

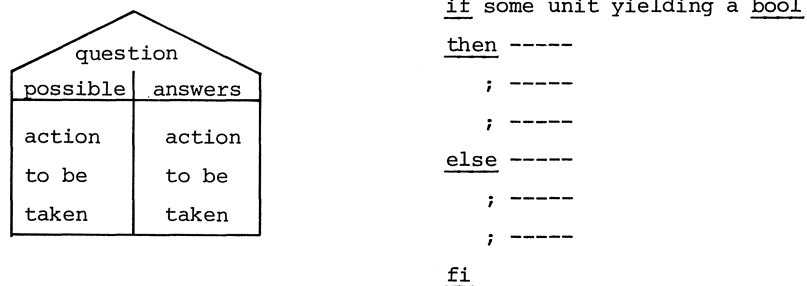


Fig.2 Choice Box and corresponding ALGOL 68 text.

Observe also that, at least in the case of beginners unbesmirched by other programming languages, students are taught to put their go-on-symbols at the beginning of a line. Apart from making it easier to edit in an extra statement at the end of a serial-clause (a much more common requirement than to insert an extra phrase at the beginning), this has significant didactic advantage. The indentation structure is emphasised, the start and the extent of each new statement is clearly flagged (hence students are less likely to try to talk about "then-statements" and "else-statements"), and the habit is cultivated of not writing a go-on-symbol until one is ready to write a unit to be gone-on to. It also helps to emphasise the idea of serial composition if these symbols are pronounced as "go-on", rather than as "semicolon", each time they are written. Fig.3 shows the program from Fig.1 laid out in this way.

```

proc find largest = ([ ] real a) real:
  ( loc real largest so far := - max real
  ; for i from lwb a to upb a
    do real element = a[i]
      ; if element > largest so far
        then largest so far := element
      fi
    od
  ; largest so far
  )

```

Fig. 3 Program layout with go-on-symbols at the start of the line

Program structure is closely related to Syntax. Students must certainly be shown some model of the Syntax in order to drive home its recursive and orthogonal nature (although persuading them actually to consult the model when constructing programs is another matter). Obviously, the 2-level Van Wijngaarden Grammar is too much to show on day one (it may be a fine thing to teach later). At this point, the verbalizers introduce some variant of BNF, but I prefer a visual representation such as those given by WATT [8] or LINDSEY [5,6], of which Fig.4 is an example. I find it particularly convenient to have a large wall chart and to refer to it frequently. It is an advantage if the chosen model can indicate some at least of the second level of the grammar, such as the strengths of the contexts and the modes yielded or expected by the various constructs.

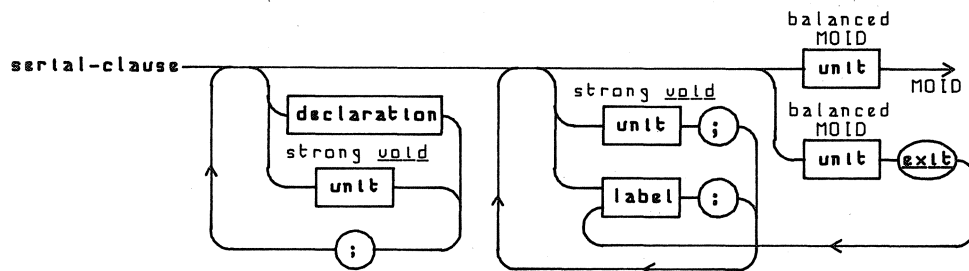


Fig. 4. Example of notation for Syntax Charts.

3. PARADIGMS

Those of us who have been writing programs for many years find ourselves writing the same piece of code in different contexts so many times, that we are apt to imagine that our particular "paradigm" is the obvious way to do the given job. To the student who has just met such amazingly new concepts as the array and the loop-clause for the first time, nothing is obvious - not even the paradigm for finding the largest element of an array given in Fig.1. In his Turing Award lecture, FLOYD [3] made this point very strongly. A repertoire of basic paradigms must be taught. It is too much to expect the student to discover them all for himself (and there is also the risk that he will discover bad ways of doing things instead).

Of course, teaching Paradigms and teaching the language constructs with which to implement them can go hand in hand. The book by ANDREW COLIN [2] adopts this technique with great success. The important thing is that students should see as many realistic programs as possible developed before their very eyes preferably, in the early stages, with a computer online to the lecture room with TV monitors connected to the VDU.

The paradigms to be taught include both activities (searching arrays, summing series, looking up tables) and data structures (lists, queues, trees of various sorts). Most paradigms apply equally to all programming languages, but there are some which are specific to ALGOL 68, of which examples are given in Section 4.3 below.

4.SOME SPECIFIC ALGOL 68 PROBLEMS

Every language has its weak points which, unless taught very carefully, will cause difficulty to the students. In ALGOL 68 the points to watch are:

- variable-declarations
 - (with which are associated assignments)
- identity-declarations
 - (and when to introduce them)
- names
 - (with which are associated dereferencing and generators)
- subnames
 - (also known as the "bend")

multiple values
 (or should they be called "arrays")
 scope
 (and how not to confuse it with "reach").

4.1. Values and ascription

It is well recognised that teaching variable-declarations is the foremost of these. At one extreme lies the school of thought [1] which exhibits the mysterious

ref real x = loc real;

almost on day one, whilst at the other extreme are those who try not to mention ref at all. I prefer to leave the whole subject for a while and to talk about a concept that is especially important to ALGOL 68 and which is already familiar to the student, namely the concept of "values", which result from the elaboration of "expressions". Fortunately, the notation for formulas and denotations in ALGOL 68 is so like that of conventional algebra that its detailed syntax need not be taught (at least not in the first instance). Next, the student should be shown how to ascribe values to identifiers in identity-declarations, and already he knows enough to write simple, but meaningful, programs (to solve quadratic equations with real roots, for example). Now he can be taught conditional-clauses (as constructs that yield values), and likewise case-clauses (because they are so similar). Quite complex programs can now be exhibited (to compute the date of Easter, for example).

4.2. Variables and assignations

So far, there has been no mention of variables, nor of assignations, nor of data input. Early programming languages made the mistake of supposing that these were indeed the fundamental concepts upon which all computing must inevitably be based. They are not. On the contrary, they are dangerous, complex, misleading and, above all, addictive. (Consider, for example, the complications of the usual axiomatic definition of assignation, and the difficulties of correctness proofs when variables are passed by reference). They should not be used unless there is good reason to do so (which, of course, there often is).

It is my experience that the programs that students write are all modeled on the first program that they ever saw. I used to show, as their very first example, something like the following.

```
# program to read two numbers and to print their sum and
  difference #
( loc int x, y
  # here point out carefully that x and y exist, but that their
    values are "undefined" at this point #
; read((x,y)
  # now they have sensible values #
; print((x+y, x-y))
)
```

Result: every time they declare a variable (even for local use inside a procedure), they follow it with a 'read' "just make sure that it has a defined value". Moral: if you want them to use identity-declarations and expression-oriented programming in appropriate situations, teach them that style of programming first. And if, as a byproduct, they get the idea that variables and assignments are an "advanced" feature of the language (and therefore "difficult"), well, they might just be right. Many textbooks make the mistake of leaving identity-declarations until nearly the end, which is the surest way to give students a bad impression of them.

Nevertheless, the time for teaching variables must soon come. The idea to instill is that space in which to keep variables is a valuable commodity, and that it can only be "manufactured" to order. The best way is to insist that the loc symbol is included in every variable-declaration (indeed, my compiler issues a long and wordy warning if ever you leave it out). Thus, variable-declarations are clearly distinguished from identity-declarations, you teach that "loc int" means "please manufacture space to hold an int variable", and you illustrate the variable-declaration

```
loc int tom := 99
```

by drawing the picture in fig.5, carefully pointing out that 'tom' knows "where" the variable is kept, and that he is very fortunate and privileged to be the sole repository of this valuable piece of information. Although the diagram obviously contains room to show the ref int value that is really accessed by 'tom' enough is enough for now, and that part of the

story can wait for another day. It suffices that the correct foundations have been laid. Now assignments can be taught (with emphasis on "what" value is assigned to "where"). And observe that the initialized form of the variable-declaration was taught first, because it is the one that should normally be used. It is easy to explain later that the initialization is optional and, perhaps at the same time, to introduce 'read'.

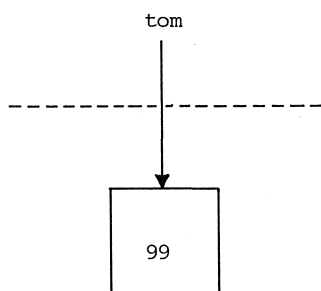


Fig.5 Van der Meulen diagram for loc int tom := 99.

And, in spite of all your efforts, your students will hereforth still use uninitialized variable-declarations, assignments and conditional-statements for everything, and will write

```
if b=c then a := true else a := false fi
```

when what they really meant was

```
a := b=c
```

Such is life!

4.3. Names

Eventually, however, "names" must be taught. The first problem is one of terminology. The Report gives us the term "name", but the man-in-the-street is undoubtedly going to confuse this with "identifier" especially as, in most situations, they turn out to be almost the same thing. Other languages have used the terms "pointer", "access" and "reference" for the same concept. I think I would have preferred "reference", but I have not dared to put my preference into practice as yet.

How the concept is introduced depends upon what the students need to know. For beginners, especially where it is not intended to teach the whole of ALGOL 68, it is best to teach ref in the first place as a special kind of formal-parameter, just like the var parameter of PASCAL.

I like to teach that calling a procedure is like giving a job to a specialist subcontractor. You give him some values and he returns a result. For a ref parameter, I say that you give him a long piece of ribbon pointing to your variable (thus allowing him to share that valuable piece of information formerly known only to 'tom'), which gives him the right to alter your variable as well as to inspect it. Later, it can be explained how this piece of ribbon can also be ascribed to suitable identifiers or assigned to suitable variables.

For more mature students who are studying the full language, it may be better to introduce the class of ref modes at an earlier stage, showing how to construct ref mode variables which refer to other variables already declared. This leads on to generators and the construction of lists, trees and the like (and

ref real x = loc real

appears as an interesting oddity which helps to tie the various concepts together). The wisdom of including loc in every variable-declaration is now apparent, because it is possible to say that "loc int" always means just the same as "ref int", except that it has the additional side effect of creating space.

At this point, one must teach the concept of what the authors of the original Report called the "bend" (because that is what it drove them round) and which, in the Revised Report, is called a "subname". This is one of the most important features of ALGOL 68, and at the same time the most difficult to teach. It is the idea that, when you select (slice) from the name referring to a structure (an array), what you get is a subname referring to the selected field (element(s)). At this stage, a visual model is essential, and the appropriate model is the Van der Meulen diagram [5] or [7].

Consider the program fragment in Fig.6. Fig.7 shows the Van der Meulen diagram corresponding to the unit `ptr := next of ptr`. It is my practice to divide the diagram with a dotted line. Everything below this line is some sort of variable which has been specifically brought into existence by use of the word loc (or maybe heap). The shapes above the line represent the yields of the external objects written within them. If the external object is an identifier, it continues to access the same value within its reach. Other external objects are transient, and may

Fig.6 also illustrates four specific paradigms, peculiar to ALGOL 68, which the students should know about.

- # 1 # Whenever a new struct is invented, it is wise to declare a private version of nil to refer to it. This avoids the traps inherent in using nil in identity-relations.
- # 2 # This is the so called "3 ref trick" (because the mode of 'ptr' has three refs in it). It should be used whenever a list or a tree is to be searched with the intention of inserting a new node at the found location. Perhaps it is unfair to refer to such an important technique as a "trick", but there is nothing like a good name to encourage students to remember an idea.
- # 3 # This brief-conditional-clause, with one part permanently yielding false, is the standard way to avoid testing the non-existent link beyond the end of the chain.
- # 4 # The standard way to insert a new node in a list or tree.

4.4. Arrays

We have another problem of terminology in the case of "multiple values". Here, however, the rest of the world is unanimous in describing them as "arrays", and there is nothing to be gained by trying to resist. My chief difficulty with arrays has been to persuade students to distinguish between formal- and actual-declarers. I explained to them ad nauseum that an "actual"-declarer is used when they want to obtain "actual" space for an "actual" object in the "actual" core store of their "actual" computer. Finally, I modified my compiler so as to parse fully both types of declarer in all contexts. This enables it then to emit very specific error messages when they get it wrong.

4.5. Scope

Another confusion of terminology arises with "scope". Unfortunately, in all other languages, "scope" denotes a purely static concept which, in ALGOL 68, should be referred to as "reach".

Contrariwise, in ALGOL 68, "scope" is a dynamic problem with which the

other languages do not have to cope. It can only be convincingly explained to students in terms of the run-time stack which they are to presume their implementor will keep. And one has to explain to them that other languages use the same word differently, and apologise for ALGOL 68 having got it wrong.

5. SOME SPECIFIC ALGOL 68 NON-PROBLEMS

The above are some problems that I have encountered in teaching ALGOL 68. Here now are some non-problems.

The Report and 2-level grammars are non-problems, because one does not teach them (at least, not until they fully understand the language).

Balancing is a non-problem because the normal user will probably make use of it without being in the least aware that he is doing anything unusual. Implementors and language designers need to know about it, but not users.

Mode equivalence is another non-problem, for the same reason.

The sublanguage ALGOL 68S is a non-problem. Although my students use an ALGOL 68S compiler for their exercises, I teach them the full language, and simply refrain from setting problems that require the missing features. Hardly any of them notices.

6. COMPILERS

The teaching of ALGOL 68 must be supported by the provision of a friendly compiler. A friendly compiler is as valuable as an extra teacher in the class. At compile time, it should give error messages that are closely related to the error that the student actually made, couched in terms with which he is familiar. Implementors should be aware that the average user is likely to be unfamiliar with all but a small percentage of the many paranoitions defined in the Report.

"missing in-part in chooser-choice-using-boolean-bold-clause", although correct, is unlikely to be as readily understood as

"no 'units after then in conditional-clause".

Also, the parser should recover quickly after all errors.

At runtime, checking should be exceedingly thorough. In addition to obvious matters like array-bound checks and scope violations, all misuses of uninitialized variables and of nil should be cleanly caught. The reporting of the error should be followed by a clear printout of the stack, giving each identifier in use together with its mode and value - which should include the fields of each structure and some at least (say the first 3 and the last 3) of the elements of each array.

7. CONCLUSION

Many people, whilst appreciating its powers, are afraid of teaching ALGOL 68 because it is "too complicated". Indeed there are difficulties; they are bigger than molehills but they are certainly not mountains. And in any case, the way to deal with mountains is to go around them. I hope that I have been able, in this paper, to show some of the ways around.

REFERENCES

- [1] BRAILSFORD, D.F. & A.N. WALKER, *Introductory ALGOL 68 Programming*, Ellis Horwood Ltd., 1979.
- [2] COLIN, A.J.T., *Programming and Problem-solving in ALGOL 68*, Macmillan, 1978.
- [3] FLOYD, R.W., 1978 ACM Turing Award Lecture: *The Paradigms of Programming*, Comm. ACM 22, 8 (Aug. 1979), p455.
- [4] LINDSEY, C.H., *Structure Charts - A Structured Alternative to Flowcharts*, SIGPLAN Notices, 12, 11 (Nov. 1977), p36.
- [5] LINDSEY, C.H. & S.G. VAN DER MEULEN, *Informal Introduction to ALGOL 68*, Revised Edition, Revised reprint 1980, North Holland, 1980.
- [6] LINDSEY, C.H., ALGOL 68 Syntax Chart on microfiche, ALGOL Bulletin, AB46.5.1.
- [7] STILLER G., *ALGOL 68 - Begriffe und Ausdrucksmittel*, BSB B.G. Teubner Verlagsgesellschaft, Leipzig, 1974.
- [8] PECK, J.E.L., M. SINTZOFF & J.M. WATT, *ALGOL 68 Syntax Chart*, ALGOL Bulletin, AB37.4.7.

LIST OF ADDRESSES OF AUTHORS

1. S.G. VAN DER MEULEN
Department of Computer Science
University of Utrecht
Princetonplein 5
3584 CC Utrecht
The Netherlands
2. J. ANDRÉ
IRISA/INRIA-Laboratoire de Rennes
35042 Rennes Cedex
France
- J. BARRÉ
IRISA-Université de Rennes
35042 Rennes Cedex
France
3. P.R. EGGERT
Department of Computer Science
University of California
Santa Barbara, CA 93106
USA
- R.C. UZGALIS
Computer Science Department
University of California
Los Angeles, CA 90024
4. G. STILLER
Department of Information Processing
Technical University Dresden
Mommssenstrasse 13
8027 Dresden
GDR
5. H. LOEPER
H.-J. JÄKEL
H. PIETSCH
Department of Information Processing
Technical University Dresden
Mommssenstrasse 13
8027 Dresden
GDR
6. K. WRIGHT
Data General Corporation
Route 9
Westboro, Ma 01580
USA
7. L.G.L.T. MEERTENS
Mathematical Centre
Kruislaan 413
1098 SJ Amsterdam
The Netherlands
8. G.J. FINNIE
M.C. THOMAS
South West Universities
Regional Computer Centre
Claverton Down
Bath, BA2 7AY
UK
9. V.J. RAYWARD-SMITH
School of Computing Studies and Ac-
countancy
University of East Anglia
Norwich, NR4 7TJ
UK

10. V. LINNEMANN

Dwostrasse 169 A
2870 Delmenhorst
BRD

11. S.D. BUTLAND

Computing Laboratory
University of Bradford
Bradford West Yorkshire BD7 1DP
UK

12. G. GÜNTHER
G. MARQUARDT

Regionales Rechenzentrum für
Niedersachsen bei der Universität
Hannover
Wunstorfer Strasse 14
3000 Hannover 91
BRD

13. C.H. LINDSEY

Department of Computer Science
University of Manchester
Manchester M13 9PL
UK

TITLES IN THE SERIES MATHEMATICAL CENTRE TRACTS

(An asterisk before the MCT number indicates that the tract is under preparation).

A leaflet containing an order form and abstracts of all publications mentioned below is available at the Mathematisch Centrum, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. Orders should be sent to the same address.

-
- MCT 1 T. VAN DER WALT, *Fixed and almost fixed points*, 1963.
ISBN 90 6196 002 9.
- MCT 2 A.R. BLOEMENA, *Sampling from a graph*, 1964. ISBN 90 6196 003 7.
- MCT 3 G. DE LEVE, *Generalized Markovian decision processes, part I: Model and method*, 1964. ISBN 90 6196 004 5.
- MCT 4 G. DE LEVE, *Generalized Markovian decision processes, part II: Probabilistic background*, 1964. ISBN 90 6196 005 3.
- MCT 5 G. DE LEVE, H.C. TIJMS & P.J. WEEDA, *Generalized Markovian decision processes, Applications*, 1970. ISBN 90 6196 051 7.
- MCT 6 M.A. MAURICE, *Compact ordered spaces*, 1964. ISBN 90 6196 006 1.
- MCT 7 W.R. VAN ZWET, *Convex transformations of random variables*, 1964.
ISBN 90 6196 007 X.
- MCT 8 J.A. ZONNEVELD, *Automatic numerical integration*, 1964.
ISBN 90 6196 008 8.
- MCT 9 P.C. BAAYEN, *Universal morphisms*, 1964. ISBN 90 6196 009 6.
- MCT 10 E.M. DE JAGER, *Applications of distributions in mathematical physics*, 1964. ISBN 90 6196 010 X.
- MCT 11 A.B. PAALMAN-DE MIRANDA, *Topological semigroups*, 1964.
ISBN 90 6196 011 8.
- MCT 12 J.A.Th.M. VAN BERCKEL, H. BRANDT CORSTIUS, R.J. MOKKEN & A. VAN WIJNGAARDEN, *Formal properties of newspaper Dutch*, 1965.
ISBN 90 6196 013 4.
- MCT 13 H.A. LAUWERIER, *Asymptotic expansions*, 1966, out of print; replaced by MCT 54.
- MCT 14 H.A. LAUWERIER, *Calculus of variations in mathematical physics*, 1966. ISBN 90 6196 020 7.
- MCT 15 R. DOORNBOS, *Slippage tests*, 1966. ISBN 90 6196 021 5.
- MCT 16 J.W. DE BAKKER, *Formal definition of programming languages with an application to the definition of ALGOL 60*, 1967.
ISBN 90 6196 022 3.

- MCT 17 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 1*, 1968.
ISBN 90 6196 025 8.
- MCT 18 R.P. VAN DE RIET, *Formula manipulation in ALGOL 60, part 2*, 1968.
ISBN 90 6196 038 X.
- MCT 19 J. VAN DER SLOT, *Some properties related to compactness*, 1968.
ISBN 90 6196 026 6.
- MCT 20 P.J. VAN DER HOUWEN, *Finite difference methods for solving partial differential equations*, 1968. ISBN 90 6196 027 4.
- MCT 21 E. WATTEL, *The compactness operator in set theory and topology*, 1968.
ISBN 90 6196 028 2.
- MCT 22 T.J. DEKKER, *ALGOL 60 procedures in numerical algebra, part 1*, 1968.
ISBN 90 6196 029 0.
- MCT 23 T.J. DEKKER & W. HOFFMANN, *ALGOL 60 procedures in numerical algebra, part 2*, 1968. ISBN 90 6196 030 4.
- MCT 24 J.W. DE BAKKER, *Recursive procedures*, 1971. ISBN 90 6196 060 6.
- MCT 25 E.R. PAËRL, *Representations of the Lorentz group and projective geometry*, 1969. ISBN 90 6196 039 8.
- MCT 26 EUROPEAN MEETING 1968, *Selected statistical papers, part I*, 1968.
ISBN 90 6196 031 2.
- MCT 27 EUROPEAN MEETING 1968, *Selected statistical papers, part II*, 1969.
ISBN 90 6196 040 1.
- MCT 28 J. OOSTERHOFF, *Combination of one-sided statistical tests*, 1969.
ISBN 90 6196 041 X.
- MCT 29 J. VERHOEFF, *Error detecting decimal codes*, 1969. ISBN 90 6196 042 8.
- MCT 30 H. BRANDT CORSTIUS, *Exercises in computational linguistics*, 1970.
ISBN 90 6196 052 5.
- MCT 31 W. MOLENAAR, *Approximations to the Poisson, binomial and hypergeometric distribution functions*, 1970. ISBN 90 6196 053 3.
- MCT 32 L. DE HAAN, *On regular variation and its application to the weak convergence of sample extremes*, 1970. ISBN 90 6196 054 1.
- MCT 33 F.W. STEUTEL, *Preservation of infinite divisibility under mixing and related topics*, 1970. ISBN 90 6196 061 4.
- MCT 34 I. JUHÁSZ, A. VERBEEK & N.S. KROONENBERG, *Cardinal functions in topology*, 1971. ISBN 90 6196 062 2.
- MCT 35 M.H. VAN EMDEN, *An analysis of complexity*, 1971. ISBN 90 6196 063 0.
- MCT 36 J. GRASMAN, *On the birth of boundary layers*, 1971. ISBN 90 6196 064 9.
- MCT 37 J.W. DE BAKKER, G.A. BLAAUW, A.J.W. DULJVESTIJN, E.W. DIJKSTRA, P.J. VAN DER HOUWEN, G.A.M. KAMSTEEG-KEMPER, F.E.J. KRUSEMAN ARETZ, W.L. VAN DER POEL, J.P. SCHAAP-KRUSEMAN, M.V. WILKES & G. ZOUTENDIJK, *MC-25 Informatica Symposium 1971*.
ISBN 90 6196 065 7.

- MCT 38 W.A. VERLOREN VAN THEMAAT, *Automatic analysis of Dutch compound words*, 1971. ISBN 90 6196 073 8.
- MCT 39 H. BAVINCK, *Jacobi series and approximation*, 1972. ISBN 90 6196 074 6.
- MCT 40 H.C. TIJMS, *Analysis of (s,S) inventory models*, 1972. ISBN 90 6196 075 4.
- MCT 41 A. VERBEEK, *Superextensions of topological spaces*, 1972. ISBN 90 6196 076 2.
- MCT 42 W. VERVAAT, *Success epochs in Bernoulli trials (with applications in number theory)*, 1972. ISBN 90 6196 077 0.
- MCT 43 F.H. RUYMGAART, *Asymptotic theory of rank tests for independence*, 1973. ISBN 90 6196 081 9.
- MCT 44 H. BART, *Meromorphic operator valued functions*, 1973. ISBN 90 6196 082 7.
- MCT 45 A.A. BALKEMA, *Monotone transformations and limit laws* 1973. ISBN 90 6196 083 5.
- MCT 46 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 1: The language*, 1973. ISBN 90 6196 084 3.
- MCT 47 R.P. VAN DE RIET, *ABC ALGOL, A portable language for formula manipulation systems, part 2: The compiler*, 1973. ISBN 90 6196 085 1.
- MCT 48 F.E.J. KRUSEMAN ARETZ, P.J.W. TEN HAGEN & H.L. OUDSHOORN, *An ALGOL 60 compiler in ALGOL 60, Text of the MC-compiler for the EL-X8*, 1973. ISBN 90 6196 086 X.
- MCT 49 H. KOK, *Connected orderable spaces*, 1974. ISBN 90 6196 088 6.
- MCT 50 A. VAN WIJNGAARDEN, B.J. MAILLOUX, J.E.L. PECK, C.H.A. KOSTER, M. SINTZOFF, C.H. LINDSEY, L.G.L.T. MEERTENS & R.G. FISHER (eds), *Revised report on the algorithmic language ALGOL 68*, 1976. ISBN 90 6196 089 4.
- MCT 51 A. HORDIJK, *Dynamic programming and Markov potential theory*, 1974. ISBN 90 6196 095 9.
- MCT 52 P.C. BAAYEN (ed.), *Topological structures*, 1974. ISBN 90 6196 096 7.
- MCT 53 M.J. FABER, *Metrizability in generalized ordered spaces*, 1974. ISBN 90 6196 097 5.
- MCT 54 H.A. LAUWERIER, *Asymptotic analysis, part 1*, 1974. ISBN 90 6196 098 3.
- MCT 55 M. HALL JR. & J.H. VAN LINT (eds), *Combinatorics, part 1: Theory of designs, finite geometry and coding theory*, 1974. ISBN 90 6196 099 1.
- MCT 56 M. HALL JR. & J.H. VAN LINT (eds), *Combinatorics, part 2: Graph theory, foundations, partitions and combinatorial geometry*, 1974. ISBN 90 6196 100 9.
- MCT 57 M. HALL JR. & J.H. VAN LINT (eds), *Combinatorics, part 3: Combinatorial group theory*, 1974. ISBN 90 6196 101 7.

- MCT 58 W. ALBERS, *Asymptotic expansions and the deficiency concept in statistics*, 1975. ISBN 90 6196 102 5.
- MCT 59 J.L. MIJNHEER, *Sample path properties of stable processes*, 1975. ISBN 90 6196 107 6.
- MCT 60 F. GÖBEL, *Queueing models involving buffers*, 1975. ISBN 90 6196 108 4.
- *MCT 61 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 1*, ISBN 90 6196 109 2.
- *MCT 62 P. VAN EMDE BOAS, *Abstract resource-bound classes, part 2*, ISBN 90 6196 110 6.
- MCT 63 J.W. DE BAKKER (ed.), *Foundations of computer science*, 1975. ISBN 90 6196 111 4.
- MCT 64 W.J. DE SCHIPPER, *Symmetric closed categories*, 1975. ISBN 90 6196 112 2.
- MCT 65 J. DE VRIES, *Topological transformation groups 1 A categorical approach*, 1975. ISBN 90 6196 113 0.
- MCT 66 H.G.J. PIJLS, *Locally convex algebras in spectral theory and eigenfunction expansions*, 1976. ISBN 90 6196 114 9.
- *MCT 67 H.A. LAUWERIER, *Asymptotic analysis, part 2*, ISBN 90 6196 119 X.
- MCT 68 P.P.N. DE GROEN, *Singularly perturbed differential operators of second order*, 1976. ISBN 90 6196 120 3.
- MCT 69 J.K. LENSTRA, *Sequencing by enumerative methods*, 1977. ISBN 90 6196 125 4.
- MCT 70 W.P. DE ROEVER JR., *Recursive program schemes: Semantics and proof theory*, 1976. ISBN 90 6196 127 0.
- MCT 71 J.A.E.E. VAN NUNEN, *Contracting Markov decision processes*, 1976. ISBN 90 6196 129 7.
- MCT 72 J.K.M. JANSEN, *Simple periodic and nonperiodic Lamé functions and their applications in the theory of conical waveguides*, 1977. ISBN 90 6196 130 0.
- MCT 73 D.M.R. LEIVANT, *Absoluteness of intuitionistic logic*, 1979. ISBN 90 6196 122 X.
- MCT 74 H.J.J. TE RIELE, *A theoretical and computational study of generalized aliquot sequences*, 1976. ISBN 90 6196 131 9.
- MCT 75 A.E. BROUWER, *Treelike spaces and related connected topological spaces*, 1977. ISBN 90 6196 132 7.
- MCT 76 M. REM, *Associations and the closure statement*, 1976. ISBN 90 6196 135 1.
- MCT 77 W.C.M. KALLENBERG, *Asymptotic optimality of likelihood ratio tests in exponential families*, 1977. ISBN 90 6196 134 3.
- MCT 78 E. DE JONGE & A.C.M. VAN ROOIJ, *Introduction to Riesz spaces*, 1977. ISBN 90 6196 133 5.

- MCT 79 M.C.A. VAN ZUIJLEN, *Empirical distributions and rank statistics*, 1977. ISBN 90 6196 145 9.
- MCT 80 P.W. HEMKER, *A numerical study of stiff two-point boundary problems*, 1977. ISBN 90 6196 146 7.
- MCT 81 K.R. APT & J.W. DE BAKKER (eds), *Foundations of computer science II*, part 1, 1976. ISBN 90 6196 140 8.
- MCT 82 K.R. APT & J.W. DE BAKKER (eds), *Foundations of computer science II*, part 2, 1976. ISBN 90 6196 141 6.
- MCT 83 L.S. BENTHEM JUTTING, *Checking Landau's "Grundlagen" in the AUTOMATH system*, 1979. ISBN 90 6196 147 5.
- MCT 84 H.L.L. BUSARD, *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?) books vii-xii*, 1977. ISBN 90 6196 148 3.
- MCT 85 J. VAN MILL, *Supercompactness and Wallman spaces*, 1977. ISBN 90 6196 151 3.
- MCT 86 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix I, A programming system for operations on vectors and matrices over arbitrary fields and of variable size*. 1978. ISBN 90 6196 152 1.
- *MCT 87 S.G. VAN DER MEULEN & M. VELDHORST, *Torrix II*, ISBN 90 6196 153 X.
- MCT 88 A. SCHRIJVER, *Matroids and linking systems*, 1977. ISBN 90 6196 154 8.
- MCT 89 J.W. DE ROEVER, *Complex Fourier transformation and analytic functionals with unbounded carriers*, 1978. ISBN 90 6196 155 6.
- *MCT 90 L.P.J. GROENEWEGEN, *Characterization of optimal strategies in dynamic games*, . ISBN 90 6196 156 4.
- MCT 91 J.M. GEYSEL, *Transcendence in fields of positive characteristic*, 1979. ISBN 90 6196 157 2.
- MCT 92 P.J. WEEDA, *Finite generalized Markov programming*, 1979. ISBN 90 6196 158 0.
- MCT 93 H.C. TIJMS & J. WESSELS (eds), *Markov decision theory*, 1977. ISBN 90 6196 160 2.
- MCT 94 A. BIJLSMA, *Simultaneous approximations in transcendental number theory*, 1978. ISBN 90 6196 162 9.
- MCT 95 K.M. VAN HEE, *Bayesian control of Markov chains*, 1978. ISBN 90 6196 163 7.
- MCT 96 P.M.B. VITÁNYI, *Lindermayer systems: Structure, languages, and growth functions*, 1980. ISBN 90 6196 164 5.
- *MCT 97 A. FEDERGRUEN, *Markovian control problems; functional equations and algorithms*, . ISBN 90 6196 165 3.
- MCT 98 R. GEEL, *Singular perturbations of hyperbolic type*, 1978. ISBN 90 6196 166 1.

- MCT 99 J.K. LENSTRA, A.H.G. RINNOOY KAN & P. VAN EMDE BOAS, *Interfaces between computer science and operations research*, 1978. ISBN 90 6196 170 X.
- MCT 100 P.C. BAAYEN, D. VAN DULST & J. OOSTERHOFF (eds), *Proceedings bicentennial congress of the Wiskundig Genootschap, part 1*, 1979. ISBN 90 6196 168 8.
- MCT 101 P.C. BAAYEN, D. VAN DULST & J. OOSTERHOFF (eds), *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2*, 1979. ISBN 90 6196 169 6.
- MCT 102 D. VAN DULST, *Reflexive and superreflexive Banach spaces*, 1978. ISBN 90 6196 171 8.
- MCT 103 K. VAN HARN, *Classifying infinitely divisible distributions by functional equations*, 1978. ISBN 90 6196 172 6.
- MCT 104 J.M. VAN WOUWE, *Go-spaces and generalizations of metrizable spaces*, 1979. ISBN 90 6196 173 4.
- *MCT 105 R. HELMERS, *Edgeworth expansions for linear combinations of order statistics*, . ISBN 90 6196 174 2.
- MCT 106 A. SCHRIJVER (ed.), *Packing and covering in combinatorics*, 1979. ISBN 90 6196 180 7.
- MCT 107 C. DEN HEIJER, *The numerical solution of nonlinear operator equations by imbedding methods*, 1979. ISBN 90 6196 175 0.
- MCT 108 J.W. DE BAKKER & J. VAN LEEUWEN (eds), *Foundations of computer science III, part 1*, 1979. ISBN 90 6196 176 9.
- MCT 109 J.W. DE BAKKER & J. VAN LEEUWEN (eds), *Foundations of computer science III, part 2*, 1979. ISBN 90 6196 177 7.
- MCT 110 J.C. VAN VLIET, *ALGOL 68 transput, part I: Historical review and discussion of the implementation model*, 1979. ISBN 90 6196 178 5.
- MCT 111 J.C. VAN VLIET, *ALGOL 68 transput, part II: An implementation model*, 1979. ISBN 90 6196 179 3.
- MCT 112 H.C.P. BERBEE, *Random walks with stationary increments and renewal theory*, 1979. ISBN 90 6196 182 3.
- MCT 113 T.A.B. SNIJDERS, *Asymptotic optimality theory for testing problems with restricted alternatives*, 1979. ISBN 90 6196 183 1.
- MCT 114 A.J.E.M. JANSSEN, *Application of the Wigner distribution to harmonic analysis of generalized stochastic processes*, 1979. ISBN 90 6196 184 X.
- MCT 115 P.C. BAAYEN & J. VAN MILL (eds), *Topological Structures II, part 1*, 1979. ISBN 90 6196 185 5.
- MCT 116 P.C. BAAYEN & J. VAN MILL (eds), *Topological Structures II, part 2*, 1979. ISBN 90 6196 186 6.
- MCT 117 P.J.M. KALLENBERG, *Branching processes with continuous state space*, 1979. ISBN 90 6196 188 2.

- MCT 118 P. GROENEROOM, *Large deviations and asymptotic efficiencies*, 1980. ISBN 90 6196 190 4.
- MCT 119 F. J. PETERS, *Sparse matrices and substructures, with a novel implementation of finite element algorithms*, 1980. ISBN 90 6196 192 0.
- MCT 120 W.P.M. DE RUYTER, *On the asymptotic analysis of large-scale ocean circulation*, 1980. ISBN 90 6196 192 9.
- MCT 121 W.H. HAEMERS, *Eigenvalue techniques in design and graph theory*, 1980. ISBN 90 6196 194 7.
- MCT 122 J.C.P. BUS, *Numerical solution of systems of nonlinear equations*, 1980. ISBN 90 6196 195 5.
- MCT 123 I. YUHÁSZ, *Cardinal functions in topology - ten years later*, 1980. ISBN 90 6196 196 3.
- MCT 124 R.D. GILL, *Censoring and stochastic integrals*, 1980. ISBN 90 6196 197 1.
- MCT 125 R. EISING, *2-D systems, an algebraic approach*, 1980. ISBN 90 6196 198 X.
- MCT 126 G. VAN DER HOEK, *Reduction methods in nonlinear programming*, 1980. ISBN 90 6196 199 8.
- MCT 127 J.W. KLOP, *Combinatory reduction systems*, 1980. ISBN 90 6196 200 5.
- MCT 128 A.J.J. TALMAN, *Variable dimension fixed point algorithms and triangulations*, 1980. ISBN 90 6196 201 3.
- MCT 129 G. VAN DER LAAN, *Simplicial fixed point algorithms*, 1980. ISBN 90 6196 202 1.
- MCT 130 P.J.W. TEN HAGEN et al., *ILP Intermediate language for pictures*, 1980. ISBN 90 6196 204 8.
- MCT 131 R.J.R. BACK, *Correctness preserving program refinements: Proof theory and applications*, 1980. ISBN 90 6196 207 2.
- MCT 132 H.M. MULDER, *The interval function of a graph*, 1980. ISBN 90 6196 208 0.
- MCT 133 C.A.J. KLAASSEN, *Statistical performance of location estimators*, 1981. ISBN 90 6196 209 9.
- MCT 134 J.C. VAN VLIET & H. WUPPER (eds), *Proceedings international conference on ALGOL 68*, 1981. ISBN 90 6196 210 2.
- MCT 135 J.A.G. GROENENDIJK, T.M.V. JANSSEN & M.J.B. STOKHOF (eds), *Formal methods in the study of language, part I*, 1981. ISBN 90 6196 213 7.
- MCT 136 J.A.G. GROENENDIJK, T.M.V. JANSSEN & M.J.B. STOKHOF (eds), *Formal methods in the study of language, part II*, 1981. ISBN 90 6196 213 7.

