ISSN 0084-6198

Algol Bulletin no. 48

AUGUST 1982

CONTENTS

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

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

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

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

Back numbers, when available, will be sent at $4 (or £1.80) each. However, it is regretted that only AB32, AB34, AB35, AB36, AB38-43 and AB45 onwards are currently available. The Editor would be willing to arrange for a Xerox copy of any individual paper to be made for anyone who undertook to pay for the cost of Xeroxing.

AB48.0 EDITOR'S NOTES.

Firstly, let me apologise for the long gap between the date on the last issue (August 1981) and the date when it was actually mailed (late December). We had problems in getting a decent reproduction of the Report and other documents contained in the microfiche. One result of this was that the symposium to mark the retirement of Professor van Wijngaarden was over by the time you received the notice of it. However the proceedings of that symposium are now available, as announced elsewhere in this issue.

Cambridge Conference

Some time ago, you received a Call for Papers for another ALGOL 68 Conference to be held in Cambridge in December of this year. Unfortunately, there was not sufficient response to enable the organisers to proceed. It is therefore regretted that the Conference has been cancelled.

ALGOL 68 on the VAX

Most of you will also have received a questionaire from SPL International, who have been commissioned to study the market potential for a possible implementation of ALGOL 68RS (the Malvern dialect already available on ICL 2900 equipment) on the VAX. If the response is good, they might even go ahead and implement it. So please will everybody with even a marginal interest in such a project please respond. It is only by vigorously responding to such initiatives that we shall ever get ALGOL 68 more widely implemented.

Inflation

The price of the ALGOL Bulletin has remained at $10 per 3 issues since January 1980. However, even with the smaller size (which saves on postage as well as on printing) it is only just breaking even - and we really ought to be building up a small reserve. The last three issues were very thin on pages (for reasons which I keep complaining to you about), but I would really like to budget on an average issue of 50 pages (which is still thin by some previous standards).

Therefore, most of you will see from the subscription notice enclosed with this issue (although some people's subscriptions are phased differently) that the price has gone up to $11 per three issues, the sterling price being £6. To those of you who pay in dollars, the increase may seem trivial. Not so to those who pay in sterling - which all goes to show how unfair and unpredictable exchange rates are. My printing bill has to be paid in sterling, and the rates quoted above do in fact allow for a little adverse movement of the exchange rates in the future.

AB48.1 Announcements.

AB48.1.1 Barry J. Mailloux: 1939 - 1982.

Bad news from Canada. Even though we all knew Barry James Mailloux would not live much longer, it still came as a shock to hear he died.

Exactly when Barry joined the Mathematical Centre in Amsterdam I do not recall, but somewhere in 1966 our plotter started to be much more useful: Barry had brought the wonderful world of Computer Graphics to Amsterdam. When I was grabbed by Aad van Wijngaarden in the middle of 1967 to become the junior Editor of the ALGOL 68 Report, I came to share a cubicle with him. He looked obviously and outspokenly North American, wearing a shoelace as a tie, colourful trousers and jackets and speaking explosively in a curious extended subset of Dutch.

Barry's responsibilities in the making of the Report were many: He carefully guarded its style and orthography, alerting us to Webster, Roget's Thesaurus and the

distinction between skewed and straight commas. He kept in his head a comprehensive overview over the whole document as it was updated daily and wholly rewritten every few months without the benefit of any automatic support for editing, cross-referencing and formatting. But his main task was to ensure the implementability.

For every aspect of the language, singly and in orthogonal combinations, Barry would dream up a number of possible implementations, be it syntactic (e.g.' assuring parseability of the two level grammar) or semantic (considering the feasibility and efficiency of various implementation models).

For computer scientists raised in the ALGOL tradition, both Barry and I had an unusual experience in real-life FORTRAN programming under early IBM operating systems, so that we shared a great concern for the transput of the new algorithmic language.

The mornings we would spend brainstorming, discussing and writing. In the afternoon, Aad would smilingly invite us into his room or even, occasionally, to his splendid house in Amsteiveen. The three of us (or four, whenever John Peck managed to join us in person) always carried our annotated copy of the latest intermediate draft. Social conversastion was in Dutch, but English was our working language even though Barry (and John) were fluent enough in the vernacular.

Tirrhenia in mid 1968 was my first WG 2.1 meeting, and also the place where I lost my virginal awe for the great Computer Scientists. The discussions were very intense but often not so deeply technical since the grasp of detail of the Working Group members was by necessity not so thorough as that of the authors. In the midst of the melee, Barry served as a bridge between the Senior Editor and the rest of the Woirking Group, pouring water on the troubled flames by working day and by drinking night, integrating each constructive proposal from the Working Group into the fabric of the Report. In this situation, Barry was at his best: keeping the visionaries and the realists together.

In the months that followed, the Editors grew even closer together, working through stacks of letters from people reacting to MR 93 and later drafts, many of whom we later came to know very well, sharing a vision and expressing it in the terse prose of MR 95, MR 99 and MR 100, living through the meeting in North Berwick and the Pyrrhic victory for ALGOL 68 in Munich in December 1968, where not even Barry's sense of balance could keep the ALGOL group together.

In the following years, Barry returned to Canada and concerned himself with teaching and implementation. The FLACC compiler was one of the results. We were endlessly embroiled in the Revision of the Report.

For the last 14 years Barry was suffering from a brain tumour. Repeatedly he was treated and recovered, but his health and his pleasure in life were steadily diminishing. Barry has deserved the rest he has now found. I am proud and thankful to him for the years we have worked together for ALGOL 68.

C. H. A. Koster

Postscript.

Although Barry's main contribution to ALGOL 68 was in the original Report, he did also play a substantial role in the Revision, being responsible for keeping the texts as they were prepared (this time in machine-readable form) and eventually for their typesetting.

The Editors of the Revision spent three weeks with him at Edmonton in the summer of 1974, where many happy hours were spent arguing over such matters as the correct form for the negative of the subjunctive in English, and the proper fount in which to represent "nil".

After that, he began to take an interest in machine architecture, especially as it could be realised by microprogramming, and finally, with the advent of X-ray scanners and their increasingly frequent application to his own head, he embarked upon research into computerized tomography. As he said, "If you can't beat them, ... ".

C. H. Lindsey

AB48.1.2 New International Journal on COMPUTERS & STANDARDS.

North-Holland Publishers (New York and Amsterdam) has launched a new international journal, COMPUTERS & STANDARDS: THE INTERNATIONAL JOURNAL.

Editor-in-Chief will be John Berg, known internationally for his work in this area.

Mr. Berg has stated that COMPUTERS & STANDARDS will provide a long-needed, independent forum for the vendors, the business community, academia, and the standards professionals. The journal seeks to provide, on a world-wide basis, fair and equal treatment for all views consistent with an orderly and constructive discussion of computer standards issues.

Interested professionals may request a FREE copy of the premier issue (on official letterhead) from: Judy Marcure, North-Holland Publishinbg Company, P.O. Box 103, 1000 AC Amsterdam, The Netherlands.

AB48.1.3 Proceedings of Van Wijngaarden Symposium.

The proceedings of the Symposium held, from Oct. 26-29 1981, to mark the retirement of Professor A. van Wijngaarden have been published by North Holland Publishing Company under the title "Algorithmic Languages" (Eds J.W. de Bakker and J.C. van Vliet). A full list of the papers given at the symposium and now published in these proceedings can be found in AB47.1.1.

AB48.1.4 Numerical Computation and Programming Languages.

The Proceedings of the IFIP TC2 Working Conference on "The Relationship between Numerical Computation and Programming Languages" (Ed. John K. Reid), held at Boulder, Colorado from Aug. 3-7 1981, have been published by North Holland Publishing Company.

Of particular interest to readers of this Bulletin may be the paper by C.G. van der Laan (Rijksuniversiteit Groningen) entitled "Programming in ALGOL 68 (as a host) and the Usage of FORTRAN (program libraries)", of which the following is the Abstract:

> A technique is described whereby a collection of FORTRAN subprograms can be made available to users of other programming languages, notable ALGOL 68. This is illustrated with some examples from Forsythe-Malcolm-Moler.

AB48.1.5

Thesis presented by Michel SIMONET - July, 3rd 1981 - University of Grenoble

W-GRAMMARS AND FIRST-ORDER LOGIC FOR THE DEFINITION AND IMPLEMENTATION OF LANGUAGES

Abstract

W-grammars are a powerful tool for the definition of languages. Their general form cannot reasonably be implemented. Moreover, their expressive power gives rise to risks of abusive use. A proposition is made to restrict them in a way which can be implemented in first-order logic.

In the first part, after an introduction to W-grammars, the author presents a survey of the studies made in this area : by Sintzoff, Hesse, Wegner, Deussen, Dembinsky, Maluszinsky, as well as formalisms similar to or derived from W-grammars : attribute systems from Knuth, affix grammars from Koster, Extended Affix Grammars from Watt, Bracketed two-level grammars from Deussen and conjugation grammars from Kramer and Schmidt. It is followed by a presentation of first-order logic, the PROLOG language and metamorphosis grammars, and the formalism of ramifications [tree-like structures, as defined by Pair] used in the third part for the definition of RW-grammars.

In the second part, three experiences of implementation in PROLOG of languages defined by a W-grammar are presented. The first one is a transcription in PROLOG of the W-grammar of ASPLE, A Simple Programming Language, already used for comparing methods of definition of languages. The second one is a subset of Algol 68 and the third one is the grammar of types in a high level language.

In the third part, a new class of W-grammars is defined : RW-grammars, whose metanotions are ramifications (trees, terms) instead of chains. These RW-grammars are equivalent to Horn Clauses (clauses of logic having at most one positive litteral) whose variables take values in domains specified by regular bi-grammars (grammars for trees). These clauses may be implemented very easily in PROLOG, and a proposition is made to introduce domains for the variables in this language in order to increase its expressive power as well as to ensure a safer programming.

AB48.1.6 Book Review : Draft Proposal for the *B* Programming Language.

by Lambert Meertens.
Mathematical Centre, Postbus 4079, 1009 AB Amsterdam.
ISBN 90 6196 238 2.
Price: HFl 11.55.

B is a language designed by Lambert Meertens, with help from Leo Geurts and further input from Robert Dewar, to be a solution to the following equation:

B : BASIC = PASCAL : FORTRAN

Its original target was undoubtedly the ecological niche which BASIC seems to have found for itself in small home and school micros. However, it has now moved a little bit upmarket, and describes itself as a "simple language for use on personal computers". It is intended to be embedded in its own *B* system (if there happens to be a larger operating system hiding in the background, that fact should, so far as possible, be invisible to the user). The command language of the *B* system should be *B* itself. Files in the *B* system are just *B* variables which, being created at the system level, have a permanent existence. An integral editor will prevent entering anything but correct *B* syntax, which will appear in a canonical pretty-printed layout on the screen (which means, for example, that change of indentation level is significant in the language, eliminating the need for begin, end, fi and the like).

Rather than give you a quick rundown of what is in the language itself, I have obtained permission to reproduce the "Quick Reference" section of the Proposal, and it appears as the next article in this Bulletin. I hope you like its style. The rest of the document is written in a more conventional and formal manner. In the main, this follows the style of the Revised ALGOL 68 Report with a 2-level Van Wijngaarden grammar (but not incorporating all of the type checking as yet). However, just as the language is simpler, so is its description less formidable. The most difficult part (which I must confess I have not yet fully understood) is the strong typing which it is claimed (in spite of the absence of declarations) can mostly be checked at compile time.

The present state of the project is that the language is now defined, but by no means frozen. Trial implementations are now in order (and prospective implementers are welcomed). The experience gained will help to improve the language for its final "official" definition.

C.H. Lindsey

AB48.4.1

QUICK REFERENCE to B, by L.G.L.T. Meertens

Numbers are exact or approximate. You get an exact number even if you use *3.14* or *22/7*. You get an approximate number if you use *E* for the ten power, or if you use the ~ function (pronounced "about"). For example, *~1000 = E3*, and *~0.005 = 0.5E−2*. You may also write *~(a+b)* etc.
Warning: an approximate number is *never* equal to an exact number. If you want to test if you may divide by *x*, and if you are not very sure that *x* is exact, it is not safe to use the test *x <> 0* (which is shorthand for *(x < 0 OR x > 0)*). You should use *~x <> ~0*.
If functions like +, −, *, / and ** work on exact numbers, the result is also computed exactly, except if the exponent *n* in *x**n* is a fraction. (A formula like $a*x**2+b*x+c$ stands for what is usually written as ax^2+bx+c: your computer cannot stand dancing lines and requires that you write * whenever you mean multiplication, even in cases like *2*x*.) Arithmetic on approximate numbers gives approximate results (which, for many purposes, are precise enough, and often are computed much faster). Functions like *root*, *sin* and *log* always give an approximate result. (So *root 4 <> 2* and *log 1 <> 0*). More details are given at the functions below.

Texts consist of characters and are written like *'Jack and Jill'* or *"Jack and Jill"*. (The characters meant are not Jack and Jill, but the "*J*", "*a*", etc. You may use any printing character and the space.) Which of the forms you use, the one with single quotes or the one with double quotes, makes no difference to your computer. Never confuse the number *747* with the text *'747'*. Whereas *747 = 3*249*, *'747'* is quite another text than the text *'3*249'*, and *'3'*'249'* is not even a text; to your computer it is meaningless. The number *747* can be used to do arithmetic; to your computer it does not consist of characters and it is written that way only because the dominant earthian species has twice five wriggly appendices sprouting from its upper tentacles and finds this clumsy notation convenient, and because you are (presumably) a member of that species and your computer tries to please you. The text *'747'*, on the other hand, cannot be used in arithmetic, and if you nevertheless try to do so, your computer will warn you. It really is three characters in a row. The so-called quotes on the outside do not really count. They only serve to make clear where the text begins and ends. If you say prayers, it does not mean that you say "prayers". But if you say "prayers", you don't say the quotes, do you? You can find out the length of a text with the function #. For example, *#'toe' = 3*. If you use *'* before and after your text, you can only use it inside if you double it thus: *''*. Your computer knows that you really mean it only once: *#'p''q' = 3*. The rules for *"* are similar.
But if you use the other quote sign inside than the one you use on the outside, you should not double it. So write either: *'He said: "don''t!"'* or: *"He said: ""don't!"""*.

Inside texts, you can use weirdos (which are known as conversions) of the form *`e`*. Your computer computes the value and replaces the conversion by a suitable text. For example, if *i = 239* and *j = 4649*, then *''i` * `j` = `i*j`'' = '239 * 4649 = 1111111'*. Within the conversions the need to double the outside quotes inside has disappeared: *''`#'toe'`'' = '3'*. (Don't look too long at it if you don't want to strain your eyes.) On the other hand, if you use a single ` as character in a text, you have to double it.
You can join two texts thus: *'now'^'here' = 'nowhere'*, and you can repeat a text as many times as you want: *'ox'^^3 = 'ox'^'ox'^'ox' = 'oxoxox'* (just like *x**3 = x*x*x*). You can take texts apart thus: *'lamplight'@4 = 'plight'* (since the "*p*" is the fourth character) and *'scarface'|5 = 'scarf'*.
You may combine @ and |: *'Benedictine'@4|5 = 'edictine'|5 = 'edict'*, and *'Benedictine'|8@4 = 'Benedict'@4 = 'edict'*.
Forms with @ and | may be used as targets:
if *t* has as content *'Benedictine'*, and you tell your computer to

```
PUT 'zedr' IN t@4|5
```

it puts *'Benzedrine'* in *t*; if *t* is *'participle'* and you tell your computer to

```
PUT '' IN t|8@7
```

it puts *'particle'* in *t*; and if *t* is *'creation'* and you tell your computer to

```
PUT 'm' IN t@4|0
```

or to

```
PUT 'm' IN t|3@4
```

it puts *'cremation'* in *t*.

Compounds are a bunch of values grouped together. For example, if you want to keep track of which books you have lent when to whom of your friends, you may tell your computer to

```
PUT 'N&P', 'Mote' IN book
PUT 84, 3, 17 IN date
INSERT book, date, 'bearded gnome' IN books'lent
```

and your computer inserts *(('N&P', 'Mote'), (84, 3, 17), 'bearded gnome')* in the list of lent books it keeps for you. (Better ask him his name next time, though.)
You can obtain the fields (as they are called) by putting the compound in a compound target. In the example, your computer would obey

PUT book IN author, title

by putting *'N&P'* in *author* and *'Mote'* in *title*.
The following is a neat trick to swap the contents of two targets:

PUT a, b IN b, a

This tells the computer to make the compound *(a, b)* and to decompose it into *(b, a)*.

Lists are like lists you make to do shopping: if you and a friend of yours each make a list, and your list is

tooth paste
shampoo
cucumbers
yoghurt
muffins
birthday present for linda

and your friend has

birthday present for linda
shampoo
tooth paste
muffins
cucumbers
yoghurt

and you compare lists, you will exclaim: why, we have *exactly* the same list. Similarly, your computer considers {*t; s; c; y; m; b*} and {*b; s; t; m; c; y*} as the *same* list. In fact, it always sorts the entries in a list from low to high; if you tell your computer to

```
PUT {5; 7; 3; 2} IN a
INSERT 4 IN a
WRITE a
```

you will see {*2; 3; 4; 5; 7*} written. The same entry may occur several times in a list. If you tell your computer to


```
PUT {} IN letters
FOR c IN 'mississippi':
    INSERT c IN letters
WRITE letters
```

it writes back {*'i'; 'i'; 'i'; 'i'; 'm'; 'p'; 'p'; 's'; 's'; 's'; 's'*}.
You may insert all kinds of values in a list, but for each list they must all be the same type of value (all numbers, or all texts, etc.). You may use {*1..n*} as shorthand for {*1; 2; ... ; n−1; n*} and similarly {*'a'..'z'*}.

Tables are somewhat like dictionaries. A short English-Dutch dictionary (not sufficient to maintain a conversation) might be

aardvark:	aardvarken
apartheid:	apartheid
furlough:	verlof
of:	van
or:	of
van:	bestelwagen
yacht:	jacht

Table entries, like entries in a dictionary, consist of two parts. The first part is called the *key*, and the second the *associate*. All keys must be the same type of value, and similarly for associates. A table may be written thus: {*['I']: 1; ['V']: 5; ['X']: 10*}.
If this table has been put in a target *roman*, then *roman['X'] = 10*.
Your computer keeps the tables sorted by key. If you next tell your computer to

PUT 100 IN roman['C']

then *roman* will contain {*['C']: 100; ['I']: 1; ['V']: 5; ['X']: 10*}. You can find out what the keys are with the function *keys*; in the example, *keys roman* = {*'C'; 'I'; 'V'; 'X'*}.

PREDEFINED COMMANDS

HOW'TO c: commands
tells your computer how to execute *your* command c. It must not be used inside other commands.

YIELD f: commands
tells your computer what value it must yield for *your* formula f when it is computed. It must not be used inside other commands.

TEST p*:* commands
tells your computer whether *your* proposition p should succeed or fail when it is tested. It must not be used inside other commands.

CHECK test
checks if the test succeeds, in which case nothing happens, but aborts if the test fails.

WRITE e
writes the value of e on the screen. It gives new lines for any /-signs before and after e.

READ t *EG* e
asks an expression from you to put in t. The e tells your computer what type of expression to ask for (number, text, etc.).

PUT e *IN* t
puts the value of e in t.

DRAW t
draws a random number (from ~0 up to ~1) and puts it in t.

CHOOSE t *FROM* l
chooses at random an element from the text, list or table l and puts it in t. (The element is not removed from l.)

SET'RANDOM e
sets the random generator, using the value of e.

REMOVE e *FROM* l
removes the value of e from the list held in l. The value must occur in that list. It is removed only once.

INSERT e *IN* l
inserts the value of e in the list held in l.

DELETE t
deletes the target t. This is used mostly to delete entries from tables or to kill permanent targets.

QUIT
quits from a *HOW'TO* or refinement.

RETURN e
returns the value of e from a *YIELD* or refinement for further computation.

REPORT test
reports from a *TEST* or refinement whether the test succeeds or fails.

SUCCEED
reports success from a *TEST* or refinement.

FAIL
reports failure from a *TEST* or refinement.

IF test*:* commands
executes the commands if the test succeeds.

SELECT:
test*:* commands
.
.
.
test*:* commands
selects the first test to succeed and executes the commands after that test. At least one test must succeed. To make sure, the last test may be *ELSE*, which catches if all other tests fail.

WHILE test*:* commands
executes the commands if the test succeeds, and keeps repeating this while the test keeps succeeding. If it fails the very first time around, the commands are not executed at all.

FOR t *IN* e*:* commands
executes the commands for t ranging over the successive characters of e if e is a text, entries of e if e is a list, and associates of e if e is a table.

ALLOW t
allows the use of the permanent t inside a *HOW'TO-*, *YIELD-* or *TEST*-body. It must occur there at the head.

PREDEFINED FUNCTIONS AND PREDICATES

Functions on numbers

~x
returns an approximate number, as close as possible in arithmetic magnitude to *x*.

x+y
returns the sum of *x* and *y*. The result is exact if both operands are exact.

+x
returns the value of *x*.

x−y
returns the difference of *x* and *y*. The result is exact if both operands are exact.

−x
returns minus the value of *x*. The result is exact if the operand is exact.

*x*y*
returns the product of *x* and *y*. The result is exact if both operands are exact.

x/y
returns the quotient of *x* and *y*. The value of *y must not* be zero (i.e., *~y <> ~0*). The result is exact if both operands are exact.

*x**y*
returns *x* to the power *y*. The result is exact if *x* is exact and *y* is an integer. If *x* is negative (i.e., *~x < ~0*), *y must* be an integer or an exact number with an odd denominator. If *x* is zero, *y must not* be negative. If *y* is zero, the result is one (exact or approximate).

n root x
returns the same as *x**(1/n)*.

root x
returns the same as *2 root x*.

abs x
returns the absolute value of *x*. The result is exact if the operand is exact.

sign x
returns an exact number from *{−1..+1}* with the same sign as *x* (where, e.g., *sign ~0 = sign −~0 = 0*).

floor x
returns the largest integer not exceeding *x* in arithmetic magnitude (so, even if perhaps *3 > ~3*, *floor ~3* still returns *3*).

ceiling x
returns the same as *− floor −x*.

n round x
returns the same as *(10**−n)*floor(x*10**n+.5)*. For example *4 round pi = 3.1416*. The value of *n must* be an integer. It may be negative: *(−2) round 666 = 700*.

round x
returns the same as *0 round x*.

a mod n
returns the same as *a−n*floor(a/n)*. (Both operands may be approximate, and *n* may be negative, but not zero.)

*/*x*
returns the smallest positive integer *q* such that *q*x* is an integer. The value of *x must* be an exact number.

x/x*
returns the same integer as *(/*x)*x*. So, if *x* is exact, *x = (*/x)/(/*x)*.

pi
returns approximately *3.1415926535...* .

sin x
returns an approximate number by applying the sine function to *x*.

cos x
returns an approximate number by applying the cosine function to *x*.

tan x
returns the same as *(sin x) / (cos x)*.

x atan y
returns an approximate number *phi*, in the range from (about) $-pi$ to $+pi$, such that x is approximated by $r * cos\ phi$ and y by $r * sin\ phi$, where $r = root(x*x+y*y)$. The operands *must not* both be zero.

atan x
returns the same as *1 atan x*.

e
returns approximately *2.7182818284...* .

exp x
returns approximately the same as *e**x*.

log x
returns an approximate number by applying the natural logarithm function (with base *e*) to *x*. The value of *x must* be positive.

b log x
returns the same as *(log x) / (log b)*.

(There should also be a collection of simple matrix functions.)

Functions on texts

t^u
returns the text consisting of *t* and *u* joined. For example, *'now'^'here' = 'nowhere'*.

t^^n
returns the text consisting of *n* copies of *t* joined together. For example, *'Fi! '^^3 = 'Fi! Fi! Fi! '*. The value of *n must* be an integer that is not negative.

x<<n
converts *x* to a text (see 5.1.2.2.b) and adds space characters to the right until the length is *n*. For example, *123<<6 = '123 '*. In no case is the text truncated; if *n* is too small, the likely effect is that your beautiful lay out is spoiled. The value of *n must* be an integer.

x><n
converts *x* to a text and adds space characters to the right and to the left, in turn, until the length is *n*. For example, *123><6 = ' 123 '*. In no case is the text truncated. The value of *n must* be an integer.

x>>n
converts *x* to a text and adds space characters to the left until the length is *n*. For example, *123>>6 = ' 123'*. In no case is the text truncated. The value of *n must* be an integer.

Functions and predicates on texts, lists and tables

keys t
requires a table as operand, and returns a list of all keys in the table. For example, *keys {[1]: 1; [4]: 2; [9]: 3} = {1; 4; 9}*.

#t
accepts texts, lists and tables. For a text operand, its length is returned, and for a list or table operand, the number of entries is returned (where duplicates in lists are counted).

e#t
accepts texts, lists and tables for the right operand.
For a text operand, the first operand *must* be a character, and the number of times the character occurs in the text is returned. For example, *'i'#'mississippi' = 4*.
For a list operand, the number of entries is returned that is equal to the first operand (which *must* have the same type as the list entries.) For example, *3 # {1; 3; 3; 4} = 2*.
For a table operand, the number of *associates* is returned that is equal to the first operand (which *must* have the same type as the associates in the table.) For example, *3 # {[1]: 3; [2]: 4; [3]: 3} = 2*.

e in t
accepts texts, lists and tables for the right operand. It succeeds if *e#t > 0* succeeds.

e not'in t
is the same as *(NOT e in t)*.

min t
accepts texts, lists and tables. For a text operand, its smallest (in the ASCII order) character is returned, for a list operand, its smallest entry is returned, and for a table operand, its smallest *associate* is returned. For example, *min 'syrupy' = 'p'*, *min {1; 3; 3; 4} = 1*, and *min {[1]: 3; [2]: 4; [3]: 3} = 3*. The text, list or table *must not* be empty.

e min t
accepts texts, lists and tables for the right operand.
For a text operand, the first operand *must* be a character, and the smallest character in the text *exceeding* that character is returned. For example, *'i' min 'mississippi' = 'm'.*
For a list operand, the smallest entry is returned exceeding the first operand (which *must* have the same type as the list entries.) For example, *3 min {1; 3; 3; 4} = 4.*
For a table operand, the smallest associate is returned exceeding the first operand (which *must* have the same type as the associates in the table.) For example, *3 min {[1]: 3; [2]: 4; [3]: 3} = 4.*
There *must* be a character, list entry or table associate exceeding the first operand.

max t and *e max t*
are like *min*, except that they return the largest element, and in the dyadic case the largest element that is less than the first operand. For example, *'m' max 'mississippi' = 'i'.*

n th'of t
requires an integer in *{1..#t}* for the left operand, and accepts texts, lists and tables for the right operand. It returns the *n*'th character, list entry or associate. In fact, *n th'of t*, for a text *t*, is written as easily *t@n|1*. For a table, it is the same as *t[n th'of (keys t)]*, which is something different from *t[n]*, unless, of course, *keys t = {1..#t}*. For a list, *1 th'of t* is *min t*.

Modular Compilation Systems for High Level Programming Languages

by I.F. Currie and N.E. Peeling

(Royal Signals and Radar Establishment, Malvern)

Introduction

This paper will try to draw some conclusions from the experience gained by the different implementations of modular compilation in Algol 68. This does not mean that it is written only for the Algol 68 community. It is also produced for those working on new high level languages, notably Ada, in the hope that they may avoid some of the problems that have befallen the implementors of Algol 68.

The proposed standard for Ada [1] seems to indicate that many of the lessons painfully learnt by Algol 68 implementors have not been passed on to the designers of Ada. Although one of the design goals of Ada was that the language should offer "support for separate compilation of program units in a way that provides the same degree of checking as within the unit", the proposed standard describes a system for modular compilation that is more dangerous than any such system that has been implemented in Algol 68.

What is modular compilation?

A modular compilation system allows a large program to be subdivided into a number of smaller modules which can be submitted for compilation separately.

By making the separate modules as self-contained as possible they can be used in the production of more than just the one program. To this end each module has a specification which defines which parts are visible outside itself. This allows the possible interactions between modules to be checked. The specification defines a module to the outside world so that a module can be altered and recompiled without affecting any other modules provided that its specification remains the same. It can be seen that if a separate compilation system admits a natural subdivision of a programming task it will provide a useful means of dividing a large problem into manageable sized portions as well as minimising the amount of recompilation necessary during development.

What is a natural subdivision of a programming task?

Programming is often described in terms of the "top down" and the "bottom up" approaches. The top down approach starts with a high level description that breaks the problem down into a number of steps which are only specified in general terms. Each of these steps is then tackled in a similar manner by breaking it down into even smaller steps. At each level more detail is introduced until a complete solution has been generated.

The bottom up approach starts by defining the lowest level of primitives first (for example defining the data structures and basic procedures for manipulating them). These primitives are then used to produce a more powerful set of facilties which can then in turn produce yet more powerful ones until the problem can be easily solved using the facilities that have been built up. The building of a subroutine library is a naturally bottom up activity with each new level producing routines of greater sophistication but less wide ranging applicability.

The top down approach tends to be used to solve a specific problem while the bottom up approach is particularly suited to the provision of a set of utilities that can be used selectively to help solve a wide range of problems.

Modular compilation systems need to provide two different types of module to cater for the top down and the bottom up approaches. All separate compilation systems produced for Algol 68 have catered for the bottom up approach (after all this is the sort of facility offered by most FORTRAN compilers), but only a few have provided a type of module suitable for top down usage. In real life, a problem will tend to be solved by a mixture of the two approaches, so where two different types of module are provided it is usually possible to combine them in a natural manner.

Some systems draw a distinction between modules and compilation units because the visibility rules for modules do not have to be linked to a particular compilation mechanism. For the sake of simplicity we will treat them as one and the same.

We will only be considering bottom up modularisation because it is the more important of the two types.

In its simplest form bottom up programming provides the facility to separately compile some (possibly restricted) piece of program, to which has been added a means of publishing identifiers declared in it for use by other modules or programs ("keep" and "pub" constructions have been used for this purpose). Kept identifiers are made available to other modules or programs if a simple directive is included in their text ("with", "use" and "access" have all been used for this sort of construction).

Simple procedure modules

The simplest and safest piece of program allowed as the unit of separate compilation is a single procedure declaration (we will refer to these as simple procedure modules). Simple procedure modules correspond to the units of separate compilation in most FORTRAN systems. Large libraries have been written in FORTRAN so it is reasonable to ask why high level languages such as Algol 68 and Ada need a different type of module. The answer is that if Ada and Algol 68 are satisfied just to copy the facilities that can be provided by FORTRAN libraries then there is no reason why simple procedure modules should not suffice. Writers of Algol 68 and Ada can however make great use of the data structuring provided by such languages and may well wish to declare and initialise data structures and make them available in the library.

A separate compilation system based on simple procedure modules has been implemented in Algol 68 (the CDC system). It provides a little more than this by having a single module (called a prelude) in which objects other than procedures can be declared and initialised. The prelude is automatically obeyed before all programs that use the library. The CDC system does present certain practical problems. If data space is to be created by the library it can only be generated locally in a procedure, or locally in the prelude which is then global to all users of the library, or

by using a global generator which will use the heap with its associated overheads. The CDC systems can also give rise to very large preludes for very large libraries. For these reasons the simple procedure modules are unpopular with the producers of large libraries (eg the NAG library).

Procedures in Algol 68 (and their equivalents in Ada) are restricted in that they cannot be produced dynamically (in particular a procedure cannot produce a new meaningful procedure as its result). This restriction is imposed to allow efficient "stack based" implementations of the languages. If this restriction is removed, procedures become increasingly attractive as the basic unit for separate compilation. A module could most naturally be treated as a procedure delivering keeps as its result (hopefully using a nice structuring facility that allows easy access to the different fields). The module's parameters would either be procedures (unevaluated modules) or keeps (evaluated modules). This approach is still not a complete solution because you cannot say "only evaluate this module if it has not been evaluated by some previous module". It is because modular compilation systems are trying to get the effect of dynamically produced procedures, without abandoning "stack based" implementations, that leads to all the complexities that are assumed to be inherent in separate compilation systems.

Modules requiring elaboration

Simple procedure modules consist of compiled code that is obeyed whenever the procedure is called. If separately compiled modules do more than just declare procedures, for example declare variables, the module may also contain code that is obeyed before the using program (the obeying of this code is referred to as the elaboration of the module). If more than one such module is being used, it becomes necessary to know if the order of elaboration of the modules is important. It may also be important to know how many times each module is elaborated.

An obvious extension to the simple procedure module is to make the unit of compilation as unrestricted as possible. To this end many systems allow any legal sequence of statements (with some expression of the module's external specification) to be compiled separately. This is the only unit of separate compilation in Algol 68-R [2], it is one of the units in the proposed standard for Algol 68 [3], and it is also one of the units in the proposed standard for Ada.

This is obviously a much more flexible unit than the procedure declaration. Unlike the procedure declaration these modules may require elaboration and a simple example will suffice to show that the order of this elaboration may have to be known if the result is not to be ambiguous.

In our examples modules will be headed by the word module followed by the name of the module. An (optional) use list of module names may be included after the module name; this use list will provide access to all the identifiers published by the modules named in it and will also cause their elaboration if required. The modules may publish identifiers in a keep list at the foot of the text.

```
MODULE aa =
        (INT i := 1)
KEEP i

MODULE bb USE aa =
        (....; i := i+1; ....)
KEEP ....

MODULE cc USE aa =
        ( ....; i := 2; ....)
KEEP ....
```

This example shows that there is an obvious partial order within a library of modules because module aa must be compiled before either bb or cc. It is thus easy to say that aa must be elaborated first, but any program that uses both bb and cc must know the order of their elaboration. For example, if the order of elaboration is aa, bb, cc the variable i refers to 2, but if the order of elaboration had been aa, cc, bb the variable i would have refered to 3.

We have so far assumed that there was only one elaboration of each module so that there was only the one copy of the variable i, which allowed the modules to communicate via the common reference. If any module that used i had its own copy (ie a module is elaborated as many times as it is used) such communication would have been impossible. It should be obvious that the number of copies of each module will affect any decisions made to define the nature of any communication between modules using common references (communication during the elaboration is often referred to as a side-effect). Is such communication a defect or a facility? It is easy to construct examples where you want communication and equally easy to construct examples where you do not (consider a module that defines a procedure that produces elements of a pseudo-random sequence - do you want modules to use the same random sequence, or do they each require their own?).

No one has managed to devise a system that allows the user to choose as many copies as required, which would be the best solution to the problem. If a copy of a module is taken every time it is used, the implementation is liable to become slow and use a lot of space. For this reason most systems take as few copies as is possible given the information known at compile time, usually just a single copy of each module. Because the proposed standard for Algol 68 allows the use of its access clauses within the body of the module text rather than the more usual approach of having a use list at the top of the module, it is not always possible to determine at compile time if a copy of a module already exists; in such cases a separate copy must be taken which can cause the most appalling confusions (a good reason for keeping the use list at the top of the module).

Comparison of the different systems

Given that a decision has been made to permit only one copy of each module, what can be done about the possibility of communication between modules using common references.

We will examine the relative advantages and disadvantages of the Algol 68-R system, the proposed standard for Algol 68, the modules system for the RS Algol 68 compiler [4] and the proposed standard for Ada.

The Algol 68-R system is somewhat of an anachronism because it was the production of this system that showed up many of these problems in the first place. It is still worth examining because it has probably been more heavily used than any other system and the problems users have had with it were an important factor in the production of the RS module system.

The Algol 68-R module system

Algol 68-R modules can only be used if they are incorporated in a library (called an album), and they are date stamped when they are put into the library. The date stamping gives a total ordering within the library. When a program is run that uses modules from the library, the total set of modules required is obtained and they are elaborated once only in order of their date stamps (oldest first). It is obvious that if this system is used to build up a library from scratch it will, of necessity, obey the partial order. If however a module is changed we must decide if any modules that use it must be recompiled. We have already said that a module can be changed without affecting any other modules provided that its external specification remains unaltered. We will now consider what constitutes the specification of an Algol 68-R module. To allow the complete interface checking that will be necessary to implement a safe system it follows that if the contents of the keep list are changed (or the modes of the elements in it) then any module that uses any of the identifiers that have changed must be recompiled. For efficiency reasons it is usual to specify that any change in the keep list of a module will require the recompilation of all modules that use it. This means that an exact description of the keep list is part of the external specification of a module. Regrettably this is not sufficient for the external specification in Algol 68-R; the use list must also be included. The necessity of including the use list will be demonstrated by an example.

Consider compiling the following modules in the given order:

```
MODULE aaa = ....

MODULE bbb USE aaa = ....

MODULE ccc USE bbb = ....
```

If we recompile aaa and change its keep list we change its specification. The old version of module aaa has to be removed from the library before being replaced by the new. Module aaa now has a new date stamp. We have to recompile bbb because it uses aaa but can we amend it (amending means that the specification of the new module is the same as the old version so that the code referred to by the module bbb can be replaced by the new code and no module that uses bbb will have to be recompiled)? The answer is no because the partial order tells us that we must elaborate aaa then bbb, while unfortunately the date stamping tells us to elaborate bbb before aaa, which is nonsense, and so we have to remove bbb from the library and then

put it in again with a new date stamp. This means that the use list of bbb complete with date stamps has formed part of the external specification. The module ccc must also be recompiled because bbb has a more recent date stamp. The result of adding the use list to the specification means that all modules that use aaa must be recompiled plus all the modules that use the recompiled modules no matter how indirectly. It will come as no surprise to know that the users of the Algol 68-R system have to rebuild their libraries quite frequently.

The Algol 68-R system imposes an external total ordering on the modules and all the recompilation problems arise because the total ordering can at times seriously contradict the necessary partial order. If a total ordering could be found that did not allow these gross anomalies with the partial order the system could probably allow a more liberal regime for amending a module.

The external total ordering does have some advantages over the proposed standard for Algol 68 which uses the syntax of the use list to give the total ordering. We will see that in the proposed standard for Algol 68 any changes in the order of the use list can alter the total ordering while in Algol 68-R this does not happen. This does not mean that the external total ordering totally freezes the side-effects as we will now demonstrate.

Consider a module

```
MODULE aaaa =
          (INT i := 1)
KEEP i
```

Imagine one person compiles a module bbbb that uses the fact that i is initialised to 1.

```
MODULE bbbb USE aaaa =
          (....; INT j := i; ....)
KEEP ....
```

Then imagine another person compiles a modules cccc that alters i

```
MODULE cccc USE aaaa =
          (....; i := 2; ....)
KEEP ....
```

If the external total order is aaaa, cccc, bbbb then any module (or program) that uses just bbbb will get the effect that the author of bbbb intended but if both bbbb and cccc are used it is likely that bbbb will not have the effect the author intended because j will refer to 2 after its assignment instead of referring to 1 as the author expected. It seems likely that the author of bbbb will feel he is being punished for the sins of the author of cccc.

The NAG Algol 68 library has been implemented in Algol 68-R and the only serious complaint is that the library is not tolerant of changes and so needs rebuilding too often. Experience with more naive users has shown that unexpected side effects are also a serious problem.

The proposed standard for Algol 68

The modules required are elaborated by a recursive procedure working from left to right in the use lists. If a module in a use list itself has a use list then the recursive procedure calls itself on this new use list. If a module has no use list or its use list has been exhausted it will be elaborated (provided that a copy is not known to exist).

This has the effect that all the use lists must be known if the total order is to be determined and that altering the order of a use list can sometimes alter the total order. Unexpected side-effects can only be avoided by knowledge, self-discipline or luck.

This system has not yet been implemented, so feed-back from users is unavailable. In the authors' opinion it is overly complex, possesses an unlovable syntax and is much too difficult for anyone who is not an Algol 68 lawyer to understand. It is however better than the proposed Ada system because given the complete text of all the modules involved all side-effects can be defined.

The modular compilation system on the RS Algol 68 compiler

The RS module system wished to avoid the shortcomings of simple procedure modules but without then encountering the problems with side effects that have beset other systems. It was decided to extend simple procedure modules so that any declarations could be made in a module (for this reason they are called DECS modules). It was also found that other statements could be included without danger of side-effects. A DECS module can be any sequence of legal steps provided that two restrictions are enforced at compile-time. Firstly, the outer level of a module must not use any reference kept in another module. Since procedures are just code that is obeyed whenever they are called it possible to make a routine text free of restrictions if a second restriction, that procedures kept in other modules are not called at the outer level, is imposed. It is essential that routine texts be free of any restrictions so that procedures called by programs that use the modules can communicate via non-locals that are also kept in the modules. These restrictions are sufficient to prevent any side-effects during elaboration, but unfortunately they prohibit the writing of some perfectly safe constructions. For example a procedure that does not alter any kept reference cannot be called in the outer level of another module even though it cannot cause a side-effect.

The designers of the module system of the RS Algol 68 compiler felt that communication during elaboration is so confusing that it is unreasonable to expect users to have to learn enough about the problem to be sure of always avoiding it, or to know enough about the module system to use the communication safely. The aim was to get the same effect (on communication) as multiple copying of modules but without the attendant overheads.

Two RS Algol 68 systems are already in use and so far the compile-time restrictions have not been found unduly difficult to work within.

The proposed standard for Ada

Ada allows the separate compilation of a number of different compilation units. These include simple procedure modules (subprograms in Ada parlance) and unrestricted pieces of program (packages).

Ada only allows for one copy of each module and definitely regards communication during elaboration as a defect because the proposed standard says ([1], section 10.5) that if the order of elaboration is important then the program is erroneous, which means ([1], section 1.6) that if the program is executed then its results are ambiguous. This is a totally unsatisfactory solution to the problem because it will be very difficult to check if a program is erroneous. The authors feel that such ambiguities can best be avoided by compile-time restrictions, or failing that, the algorithm for the elaboration should be given.

The authors did not find this mechanism particularly easy to understand.

Conclusions

The authors' opinions of the relative merits of the various modular compilation systems have already been given. We would however like to add one more general comment. If, for efficency reasons, only one copy of each module is allowed then we feel that a minimal set of restrictions should be applied to get the same effect (on communication) as multiple copies. The restrictions in RS modules are one such set. The inability to call procedures in modules because there is no way of telling if they alter references non-locally is however quite a serious restriction. If a language contained a separate construct to describe a procedure that cannot alter non-locals then this sort of function could be called at the outermost level of a module. It is also possible that such a function could be made completely free of side-effects by prohibiting the use of reference parameters. A "side-effect free" function has a lot to be said for it in its own right because it would accurately reflect the mathematical idea of a function transforming one set of values into another set.

Acknowledgments

The authors would like to thank Miss Susan Bond, Dr J.M.Foster and Dr D.P.Jenkins for all their advice and help.

References

[1] Reference Manual for the Ada Programming Language - Proposed Standard Document. United States Department of Defense, (1980).

[2] Woodward,P.M. and Bond,S.G., "Algol 68-R Users Guide". Her Majesty's Stationery Office, (1975).

[3] Lindsey,C.H. and Boom,H.J., A Modules and Separate Compilation Facility for Algol 68, Algol Bulletin, 43, (1978).

[4] Bond,S.G. and Woodward,P.M., Introduction to the 'RS' portable compiler, RRE Technical Note, 802, (1977).

AB48.4.3

The NAG ALGOL 68 Library

by G.S. Hodgson (NAG Central Office, Oxford)

1. Numerical Algorithms Group (NAG)

The Numerical Algorithms Group (NAG) has been in existence as a co-operative inter-university venture since 1970.

Its aim is to assist users of University computers by the development of a balanced general-purpose numerical library, which is well documented and validated. Each "contributor" to the Library is assigned a specific area of Numerical Analysis in which he has responsibility for the selection, collection, collation, testing and documentation of suitable routines. Test programs are also provided by the appropriate contributor. All software is retested by a person other than the contributor. NAG provides a Library Service by the provision and support of the compiled version of the NAG Library and its accompanying documentation. A copy of the source text of the Library is also available for inspection at each installation.

The original versions of the Library are available in Algol 60 and in FORTRAN. In 1973 it was decided to proceed with an Algol 68 version, and this version has also been produced as a co-operative effort between a number of University and Research centres both in this country and in the Netherlands.

2. Implementations

The Algol 68 Library has been "implemented" on several different ranges of computers. The table below lists the implementations currently available and those in progress.

COMPUTER SYSTEM	COMPILER	LIBRARY MARK NOW	NEXT	DUE	COMMENTS
CDC 7600/CYBER	CDC	2	3	-	
IBM 360/370/AMDAHL	FLACC	2	3	-	No Library Mechanism
ICL 1900*	68R	3	4	-	NON - 1906A/S
ICL 1906A/S	68R	3	4	-	
ICL 2900(B)	68RS	-	3	DEC 82	VME/B
TELEFUNKEN TR440	68C	2	3	-	

In order to make the NAG Library available on the different compilers, it has been necessary to restrict the features of Algol 68 which are used. The major restrictions are summarised below:

a) Defining occurrences of identifiers, operators and mode indications must precede their applied occurrences.
b) Avoid heap generators.
c) No flexible arrays (the mode STRING is a flexible array).
d) Avoid unions.
e) No fields of a structure may be of mode ROW (reference to ROW is allowed).
f) The mode ROW of ROW is not allowed.
g) The symbol GOTO may not be omitted in jumps.
h) A mode indication may not be the same as any operator.
i) No parallel clauses or semaphores.
j) No vacuums.
k) Use only "worthy" characters in operator tokens.

The restriction on heap generators has been relaxed so that array generating packages such as Torrix can be included. However, we have been careful only to include heap generators in a very restricted number of array generating procedures. A distinction has been made in the use of such procedures, that is:

a) use of the heap in a stack oriented manner (level 1),

and

b) use of the heap with a garbage collector (level 2).

In this way, we believe that it may be possible to provide level 1 facilities on a compiling system which does not provide a built in garbage collector.

No use has been made of unions at Mark 3, although it is intended to use unions in some procedure specifications at Mark 4. This use will be restricted only to the formal parameters of a limited number of routines which are directly called by the user. It is not intended to manipulate unions within the bodies of routines.

With these restrictions it has been possible to translate between dialects in an automated manner, with one exception. It is sometimes necessary to make very limited use of machine coding. For efficiency, it is desirable to be able to plant "inline" code. The way this is done varies between the different compilers. The CDC compiler provides the most flexible system - inline operators can be defined; in 68R the code sections have to be included explicitly; in 68RS some inline operators are provided but new ones cannot be defined. Hence some tailoring of the source code is necessary.

Each new implementation requires the running of test programs and the checking of their results - about 3 man months work. But there is a further and more serious delay in introducing further implementations. Dialect conversion is automatic, but it has not been found possible to automate the conversions necessary because of the diverse library mechanisms.

The existing mechanisms are incompatible:

a) CDC - a top down mechanism defining a prelude with automatic inclusion of precompiled NAG routines at each applied occurrence. There is no protection between items within the prelude.

b) FLACC - no library mechanism is currently available, the operating system has to be extended to provide primitive source text inclusion of NAG routines.

c) 68R - a bottom up mechanism with protection of modules from one another and from the particular program.

d) 68RS - both a top down and bottom up mechanism, however the bottom up mechanism has restrictions to exclude modules with side effects. This excludes a very limited number of NAG facilities.

e) 68C - a combination of top down and bottom up facilities. However the bottom up facilities are very primitive and the user is responsible for including a dummy specification for each library routine used in the particular program.

With these diverse mechanisms, which all differ from the standard mechanism recommended by Lindsey & Boom [AB 43.pp 19-53], further restrictions are required to make it possible to adapt (though not by automatic means) the library source code to the different mechanisms. We summarise the major restrictions:

i) Definition Modules must be side-effect free. That is:

a) No procedures or user-defined operators may be called, except within routine texts.

b) No labels may be declared in the outermost level.

c) No public item which is a reference (or a structure, array or union containing a reference) may be used in a definition module other than the one in which it was declared, except within a routine text.

With such restrictions, the order of invocation of such definition modules is irrelevant to the user.

2) All ACCESSing of modules must be suitable for accessing in the form:

ACCESS A ACCESS B ... .

This means that clashes of identifiers will be resolved in a range structured manner - the identifiers PUBlished in B will take precedence over those of A. Contributors must therefore take care that items are not publicly declared twice (possibly with different specifications) without good cause, since no check can be made that an unfortunate order of accessing of modules does not result in a change of indentification (and hence meaning) of such items.

3) All accessing of modules must be done at the head of a definition module.

This ensures that only one invocation of a module can occur; a contributor is not allowed to generate multiple invocations by for example ACCESSing a module within a routine text (see Lindsey and Boom AB 43 pp. 22, 23).

4) The standard mechanism only permits PUBlic ACCESSing of a module to publish all the PUBlicly declared items - selective publications using the keeplist mechanism of 68R and 68RS is not allowed.

5) All objects defined at the outermost level of a module (i.e. not local to a routine text) must follow standard NAG naming conventions to avoid conflicting names (all such objects are compiled together in the CDC prelude).

These restrictions only make it possible to adapt the library source code for the different mechanisms, the syntactical form of the library required for the different mechanisms varies considerably. Hence three different source versions of the NAG Library exist - CDC, 68R and 68RS. Other implementations have to be begun from whichever of these implementations provides the closest starting point.

3. Scope of the Library

The Mark 3 Algol 68 Library provides, with very limited exceptions, facilities equivalent to those of the Mark 5 FORTRAN Library, with additional material in some chapters based on Marks 6, 7 or 8 of the FORTRAN Library. Also included are some facilities not available in other language versions of the Library (e.g. multiple length integer and rational arithmetic packages and the vector and matrix operations package - Torrix).

4. Acknowledgement

NAG is most grateful to the Royal Signals and Radar Establishment, Malvern for a significant contribution to the funding of co-ordination for, and contribution to, the Mark 3 Algol 68 NAG Library. Our thanks are also due to contributors in university and research centres for their continuing voluntary efforts.

Summary of the contents of the NAG Algol 68 Library, Mark 3

The NAG Library is organised into chapters, each devoted to a branch of numerical computation. Each chapter has a one- or three-character name and a title, based on the ACM modified SHARE Classification Index. The chapters in the Mark 3 Library are:

A02 - COMPLEX ARITHMETIC
A04 - EXTENDED ARITHMETIC
C02 - ZEROS OF POLYNOMIALS
C05 - ROOTS OF ONE OR MORE TRANSCENDENTAL EQUATIONS
C06 - SUMMATION OF SERIES
D01 - QUADRATURE
D02 - ORDINARY DIFFERENTIAL EQUATIONS
D04 - NUMERICAL DIFFERENTIATION
D05 - INTEGRAL EQUATIONS
E01 - INTERPOLATION
E02 - CURVE AND SURFACE FITTING
E04 - MINIMISING OR MAXIMISING A FUNCTION
F01 - MATRIX OPERATIONS, INCLUDING INVERSION
F02 - EIGENVALUES AND EIGENVECTORS
F03 - DETERMINANTS
F04 - SIMULTANEOUS LINEAR EQUATIONS
F05 - ORTHOGONALISATION
G01 - SIMPLE CALCULATIONS ON STATISTICAL DATA
G02 - CORRELATION AND REGRESSION ANALYSIS
G05 - RANDOM NUMBER GENERATORS
H - OPERATIONS RESEARCH
M01 - SORTING
P01 - ERROR TRAPPING
S - APPROXIMATIONS OF SPECIAL FUNCTIONS
T01 - VECTOR AND MATRIX OPERATIONS, TORRIX
X02 - MACHINE CONSTANTS
X03 - INNERPRODUCTS

Each routine name has six characters. The first three denote the chapter or subchapter and the sixth and last character is 'E' in the standard precision version of the Algol 68 Library e.g. D02ADE.

This document lists the routines in the NAG Mark 3 Algol 68 Library, chapter by chapter. Routines which were introduced into the Library at Mark 3 are marked with an asterisk(*). There are 298 routines accessible to users, of which 11 will be withdrawn at Mark 4 and are not included in this list, they have been superseded by improved routines which are already in the Library.

This document is designed only to give an indication of the contents of the Library. For detailed guidance on the choice of a suitable routine, please refer to the Chapter Introductions in the NAG Algol 68 Library Manual. Each routine is completely specified by a routine document in the Library Manual.

For further information about the NAG Library and the NAG Library Service, please contact the Library Service Co-ordinator at:

Numerical Algorithms Group Limited,
NAG Central Office,
Mayfield House,
256 Banbury Road,
Oxford OX2 7DE,
United Kingdom.

Tel: National 0865 511245
International +44 865 511245
Telex: 83354 NAG UK G

North American readers may find it more convenient to contact:

The Company Secretary,
Numerical Algorithms Group (USA) Inc,
1250 Grace Court,
Downers Grove,
Illinois 60516,
USA.

Tel: (312) 971 2337

A02 - COMPLEX ARITHMETIC
- Square root of a complex number — A02AAB*

A04 - EXTENDED ARITHMETIC
- Operators for multiple-length arithmetic:
  - integer arithmetic — A04AAB*
  - rational arithmetic — A04ABB*

C02 - ZEROS OF POLYNOMIALS
- All zeros of a polynomial, by Grant and Hitchin's method:
  - complex coefficients — C02ADB*
  - real coefficients — C02AEB*

C05 - ROOTS OF ONE OR MORE TRANSCENDENTAL EQUATIONS
- Zero of a continuous function of one variable:
  - by linear interpolation, extrapolation and bisection — C05AAB
  - by hyperbolic interpolation — C05ABB
  - by bisection — C05ACB
- Solution of a system of N non-linear equations in N variables (see also Chapter E04):
  - using function values only — C05NAB*
  - using first derivatives — C05PAB*

C06 - SUMMATION OF SERIES
- Finite Fourier transforms, by Cooley-Tukey algorithm:
  - 2^m real data values — C06AAB
  - 2^m complex data values — C06ABB*
  - arbitrary number of complex data values, within a multi-variable transform — C06ADB*
- Circular convolution of two real vectors of period 2^m — C06ACB*

D01 - QUADRATURE
- Gaussian quadrature with a specified number of points:
  - one-dimensional integral — D01BAB
  - complex contour or line integral — D01BBB
  - multi-dimensional integral over product region — D01FBB
- Integral of a function defined by data values only, by Gill and Miller's method — D01GAB
- Format of structures to define integration rules — D01QPB*
- Global variables for referring to integration structures — D01QRB
- Service routines for structures defining integration rules:
  - to obtain closest n-point rule — D01QAB
  - to obtain mapped set of weights and abscissae — D01QBB

D02 - ORDINARY DIFFERENTIAL EQUATIONS
- Initial value problems for a system of O.D.E.s:
  - Merson's (Runge-Kutta) method, over one step — D02AAB
    over a range — D02ABB
  - Krogh's method — D02AHB
  - Gear's method for stiff systems — D02AJB
- Boundary value problems for a system of O.D.E.s:
  - Two-point boundary value problem — D02ADB

D04 - NUMERICAL DIFFERENTIATION
- Derivatives up to order 14 of a function of a single real variable — D04AAB
- Normalised Taylor coefficients of a function of a single variable:
  - at a point in the complex plane — D04ABB
  - at a point on the real axis — D04ACB

D05 - INTEGRAL EQUATIONS
- Linear non-singular Fredholm equation of 2nd kind — D05CAB

E01 - INTERPOLATION (see also Chapter E02)
- Interpolated values:
  - one variable, data at equally spaced points, by Everett's formula — E01ABB
    data at unequally spaced points, by fitting cubic spline — E01ADB
  - two variables, data on rectangular grid, by fitting bi-cubic spline — E01ACB

E02 - CURVE AND SURFACE FITTING
- Minimax curve fit by polynomials — E02ACB
- Least squares curve fit:
  - by polynomials, arbitrary data points — E02ADB*
    arbitrary data points, polynomial factor may be specified — E02AEB*
    special data points (including interpolation) — E02AFB*
  - by cubic splines (including interpolation) — E02BAB*
- Evaluation of fitted functions:
  - polynomial in one variable, from Chebyshev series form — E02AKB*
  - cubic spline, as computed by E02BAB, function only — E02BBB*
    function and derivatives — E02BCB*
    definite integral — E02BDB*
- Differentiation and integration of fitted functions:
  - derivative of polynomial in Chebyshev series form — E02AEB*
  - integral of polynomial in Chebyshev series form — E02AJB*

E04 - MINIMISING OR MAXIMISING A FUNCTION

(a) Function of a Single Variable
- Minimum of a Function of One Variable:
  - using function values only — E04ABB*
  - using first derivative — E04BBB*

(b) Function of Several Variables
- Unconstrained minimum (easy-to-use routines):
  - using function values only, quasi-Newton algorithm — E04CGB*
  - using first derivatives, quasi-Newton algorithm — E04DEB*
    modified Newton algorithm — E04DFB*
  - using first and second derivatives, modified Newton algorithm — E04EBB*
- Minimum subject to simple bounds on the variables (easy-to-use routines):
  - using function values only, quasi-Newton algorithm — E04JAB*
  - using first derivatives, quasi-Newton algorithm — E04KAB*
    modified Newton algorithm — E04KCB*
  - using first and second derivatives, modified Newton algorithm — E04LAB*
- Minimum subject to simple bounds on the variables (comprehensive routines):
  - using function values only, quasi Newton algorithm — E04JBB*
  - using first derivatives, quasi Newton algorithm — E04KBB*
    modified Newton algorithm — E04KDB*
  - using first and second derivatives, modified Newton algorithm — E04LBB*
- Unconstrained minimum of a sum of squares:
  - using first derivatives, Marquardt's method — E04GAB*
- Service routines:
  - finite-difference intervals for estimating first derivatives — E04HBB*
  - check user's routine for calculating first derivatives of function — E04HCB*
  - check user's routine for calculating second derivatives of function — E04HDB*

F01 - MATRIX OPERATIONS, INCLUDING INVERSION
- Matrix inversion:
  - accurate inverse,
    - complex matrix — F01EDB
    - real matrix — F01EBB
    - real symmetric band matrix — F01EKB
    - real symmetric positive-definite band matrix — F01EHB
    - real symmetric positive-definite matrix — F01EFB

approximate inverse,
- complex matrix — F01ECB
- real matrix — F01EAB
- real symmetric band matrix — F01EYB
- real symmetric positive-definite band matrix — F01EGB
- real symmetric positive-definite matrix — F01EEB

Pseudo inverse of a real mxn matrix — F01BLB*

Generalised or pseudo inverse of A^TA, where A is a real mxn matrix — F01EYB*

Matrix factorisations (see also Chapter F03):
- Rank and QR factorisation of a real mxn matrix, with column pivoting — F01BKB*

Balance a matrix by diagonal similarity transformations:
- complex matrix — F01FBB*
- real matrix — F01FAB*

Reduction of matrices to condensed form by similarity transformations:
- complex matrix to upper Hessenberg form — F01UPB*
- complex Hermitian matrix to real tridiagonal form — F01VPB*
- real matrix to upper Hessenberg form — F01SPB*
- real symmetric matrix to tridiagonal form,
  - full storage mode — F01AGB
  - full storage mode, accumulating product of transformations — F01AJB
  - rowwise storage mode — F01TPB*
- real symmetric band matrix to tridiagonal form
  - diagonal storage mode — F01TSB*
  - rowwise storage mode — F01TRB*

Backtransformation of eigenvectors from those of reduced forms (see also Chapter F02):
- real symmetric matrix, after reduction to tridiagonal form,
  - full storage mode — F01AHB

Matrix and vector arithmetic (see also Chapter T01):
- complex case,
  - matrix addition — F01D3B*
  - matrix initialisation — F01C3B
  - matrix multiplication — F01D3B*
  - matrix norm — F01C6B
  - matrix subtraction — F01D3B*
  - matrix trace — F01C6B
  - matrix transpose — F01C3B
  - unit matrix — F01C3B
  - vector addition — F01C9B*
  - vector division — F01C9B*
  - vector initialisation — F01C3B
  - vector multiplication — F01C9B*
  - vector norm — F01C6B
  - vector subtraction — F01C9B*
- integer case,
  - matrix addition — F01D1B
  - matrix initialisation — F01C1B
  - matrix multiplication — F01D1B
  - matrix norm — F01C4B
  - matrix subtraction — F01D1B
  - matrix trace — F01C4B
  - matrix transpose — F01C1B
  - unit matrix — F01C1B
  - vector addition — F01C7B
  - vector initialisation — F01C1B
  - vector multiplication — F01C7B
  - vector norm — F01C4B
  - vector subtraction — F01C7B
- real case,
  - matrix addition — F01D2B*
  - matrix initialisation — F01C2B

  - matrix multiplication — F01D2B*
  - matrix norm — F01C5B
  - matrix subtraction — F01D2B*
  - matrix trace — F01C5B
  - matrix transpose — F01C2B
  - unit matrix — F01C2B
  - vector addition — F01C8B
  - vector division — F01C8B
  - vector initialisation — F01C2B
  - vector multiplication — F01C8B
  - vector norm — F01C5B
  - vector subtraction — F01C8B

F02 - EIGENVALUES AND EIGENVECTORS

Matrix eigenvalue problems (black box routines):
- complex matrix, all eigenvalues and (optionally) eigenvectors — F02GAB*
  - selected eigenvalues $(\lambda_p,\lambda_{p+1},\ldots,\lambda_q)$ and eigenvectors — F02GCB*
  - selected eigenvalues $(\alpha \le \lambda \le \beta)$ and eigenvectors — F02GDB*
- complex Hermitian matrix, all eigenvalues and (optionally) eigenvectors — F02HAB*
- real matrix, all eigenvalues and (optionally) eigenvectors — F02EAB*
  - selected eigenvalues $(\lambda_p,\lambda_{p+1},\ldots,\lambda_q)$ and eigenvectors — F02ECB*
  - selected eigenvalues $(\alpha \le \lambda \le \beta)$ and eigenvectors — F02EDB*
- real symmetric matrix, all eigenvalues — F02AAB
  - selected eigenvalues and eigenvectors — F02ACB

Matrix eigenvalue problems (specialised routines):
- complex matrix, all eigenvalues and (optionally) eigenvectors,
  - after reduction to upper Hessenberg form by F01UPB, by LR algorithm — F02UAB*
- complex upper Hessenberg matrix,
  - selected eigenvectors, by inverse iteration — F02UCB*
- real matrix, all eigenvalues and (optionally) eigenvectors,
  - after reduction to upper Hessenberg form by F01SPB, by QR algorithm — F02SAB*
- real upper Hessenberg matrix,
  - selected eigenvectors, by inverse iteration — F02SCB*

real symmetric matrix, all eigenvalues and (optionally) eigenvectors,
- after reduction to tridiagonal form by F01AJB, by QL algorithm — F02TAB*

real symmetric tridiagonal matrix,
- selected eigenvalues $(\alpha \le \lambda \le \beta)$ and eigenvectors,
  - by bisection and inverse iteration — F02ASB

real symmetric band matrix,
- selected eigenvectors, by inverse iteration — F02TDB*

Backtransformation of eigenvectors from those of reduced forms and normalisation:
- complex matrix, after reduction to Hessenberg form — F02UPB*
- complex Hermitian matrix, after reduction to tridiagonal form — F02VPB*
- real matrix, after reduction to Hessenberg form — F02SPB*
- real symmetric matrix, after reduction to tridiagonal form,
  - rowwise storage mode — F02TPB*

Ordering of eigenvalues and (optionally) eigenvectors:
- complex matrix,
  - by moduli — F02UJB*
  - by real parts — F02UKB*
- real matrix,
  - by moduli — F02SJB*
  - by real parts — F02SKB*

F03 - DETERMINANTS

Determinant evaluation (black box routines):
- complex matrix — F03ADB
- real matrix — F03AAB
- real symmetric band matrix — F03AFB
- real symmetric positive-definite matrix — F03ABB

H - OPERATIONS RESEARCH

Linear programming problem:	
simplex algorithm, one iteration	H01ABB
contracted simplex method	H01AEB
Network problem, shortest path	H04CAB

M01 - SORTING

Sort a vector, by Singleton's implementation of Quicksort:	
real numbers, into ascending order	M01ANB
into descending order	M01APB
integers, into ascending order	M01AQB
into descending order	M01ARB
character data, into alphanumeric or reverse alphanumeric order	
or some other user-specified order	M01BAB*
Sort a vector and provide an index to the original order:	
real numbers, into ascending order	M01AJB
into descending order	M01AKB
integers, into ascending order	M01ALB
into descending order	M01AMB
Provide an index to the sorted order of a vector, leaving the vector unchanged:	
real numbers, into ascending or descending order	M01AAB*
integers, into ascending or descending order	M01ACB*
Sort the rows of a matrix on keys in an index column:	
real numbers, into ascending order	M01AEB
into descending order	M01AFB
integers, into ascending order	M01AGB
into descending order	M01AHB
Sort the rows of a matrix on keys in an index column(s):	
character data, into alphanumeric or reverse alphanumeric order	
or some other user-specified order	M01BCB*

P01 - ERROR TRAPPING

Suppress or divert error messages	NAGERF*
Mode of failure routines	NAGFAIL
Terminate program with error message and number	NAGHARD
Interrupt program with error message and number	NAGSOFT

S - APPROXIMATIONS OF SPECIAL FUNCTIONS

Arcsin(x)	S09AAB*
Arccos(x)	S09ABB*
Tanh(x)	S10AAB
Sinh(x)	S10ABB
Cosh(x)	S10ACB
Arctanh(x)	S11AAB*
Arcsinh(x)	S11ABB*
Arccosh(x)	S11ACB*
Exponential integral, $E_1(x)$	S13AAB
Sine integral, Si(x)	S13ABB
Cosine integral, Ci(x)	S13ACB
Gamma function	S14AAB
Cumulative normal distribution function, P(x)	S15ABB
Complement of cumulative normal distribution function, Q(x)	S15ACB
Error function, erf(x)	S15AEB
Complement of error function, erfc(x)	S15ADB
Bessel functions:	
$J_0(x)$	S17AEB*
$J_1(x)$	S17AFB*
$Y_0(x)$	S17ACB
$Y_1(x)$	S17ADB

Modified Bessel functions:

$I_0(x)$	S18AEB*
$I_1(x)$	S18AFB*
$K_0(x)$	S18ACB
$K_1(x)$	S18ADB

T01 - VECTOR AND MATRIX OPERATIONS, TORRIX

Torrix Basis, real systems:	
fundamental declarations and operators	T01AAB*
array generating routines	T01ABB*
array generating operations	T01ACB*
bound interrogations	T01ADB*
value interrogations	T01AEB*
new values	T01AFB*
interchanges	T01AGB*
new descriptors	T01AHB*
trimming operations	T01AKB*
summation and total extrema	T01ALB*
concrete extrema	T01AMB*
level 1 assigning additions	T01ANB*
level 1 assigning multiplications	T01APB*
array generating additions	T01AQB*
level 2 assigning additions	T01ARB*
array generating multiplications with scalar	T01ASB*
sum products	T01ATB*
array generating multiplications	T01AUB*
Torrix Extended, complex systems:	
fundamental declarations and operators	T01BAB*
array generating routines	T01BBB*
Torrix Extended, sparse systems:	
fundamental declarations	T01CAB*
array generating routines	T01CBB*

X02 - MACHINE CONSTANTS

Implementation-dependent constants and mathematical constants	X02AAB*
Implementation-dependent constants for Torrix-Basis	X02ABB*

X03 - INNERPRODUCTS

Single and extended precision inner-products:	
complex vectors	X03DBB*
real vectors	X03DAB*
rows and columns of real symmetric matrices	X03DCB*
Extended precision operations:	
complex arithmetic	X03DBB*
real arithmetic	X03DAB*

References

The NAG Algol 68 Library Manual - Mark 3 (1981), NAG Central Office, 256 Banbury Road, Oxford.