ISSN 0084-6198

# Algol Bulletin no. 42

MAY 1978

CONTENTS

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

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

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

Facilities for the reproduction and distribution of the Bulletin have been provided by Professor Dr. Ir. W. L. van der Poel, Technische Hogeschool, Delft, The Netherlands. Mailing in N. America is handled by the AFIPS office in New York.

The ALGOL BULLETIN is published approximately three times per year, at a subscription of $7 per three issues, payable in advance. Orders and remittances (made payable to IFIP) should be sent to the Editor. Payment may be made in any currency (a list of acceptable approximations in the major currencies will be sent on request), but it is the responsibility of each sender to ensure that cheques etc. are endorsed, where necessary, to conform to the currency requirements of his own country. Subscribers in countries from which the export of currency is absolutely forbidden are asked to contact the Editor, since it is not the policy of IFIP that any person should be debarred from receiving the ALGOL BULLETIN for such a reason.

The Editor of the ALGOL BULLETIN is:

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

Back numbers, when available, will be sent at $3 each. However, it is regretted that only AB32, AB34, AB35, AB38. AB39 and AB41 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.

## AB42.0 EDITOR'S NOTES

The Working group met at St Edmund's Hall, Oxford in December of last year. The main work of the WG is now proceeding in the direction of trying to understand the sort of languages one would need for the specification of computations, without too much pressure from the problems of implementation. A certain professor teaching ALGOL once remarked that, before constructing the program to perform some operation, he would first describe his intended algorithm "in abstracto". At the end of the class, one of the students came to him and asked for further details of this language "ABSTRACTO". Well, the WG is trying to decide what ABSTRACTO would be like, and some of the papers in this issue of AB are working papers of the Oxford meeting which tried to answer that question.

Discussion of ALGOL 68 matters mainly took place in meetings of the Support Sub-committee, and of its Task Force on Transput. It is now the established policy to publish "commentaries" dealing with the various bugs and other problems that have arisen in connection with the Revised Report, rather than to try and publish errata piecemeal. The first two of these commentaries appear in this issue (AB42.3.1) and it is expected that another batch, dealing mainly with transput, will appear in the next issue. Conventional errata will still be prepared for misprints and other such trivia without technical significance, and I can provide a list of these to interested parties on request (see AB42.1.2).

Other matters discussed included a proposal for a Modules and Seperate Compilation extension to the language (this should be finalized at the next meeting - see AB42.1.1 in this issue) and the possibility of publishing an implementation model of the Transput which implementors could then implement more or less as it stands without having first to understand all the finer nuances of the official definition (which, whilst being "correct" - some bugs excepted - never pretended to be an efficient means of implementation). J. C. van Vliet of the Mathematisch Centrum, Amsterdam is working on this alternative model.

Finally, it was announced during the meeting that John Peck, the Working Group Chairman, would be resigning, having completed the customary three year stint. The new Chairman is to be:

Robert B. K. Dewar,
New York University, Courant Institute of Mathematical Sciences,
251 Mercer Street, New York, N.Y. 10012, U.S.A.

Likewise, Bob Uzgalis, Convenor of the ALGOL 68 Support Sub-committee, is retiring, and is to be replaced by:

S. G. van der Meulen,
Vakgroep Informatica, Rijksuniversiteit Utrecht,
Budapestlaan 6, Utrecht 2506, The Netherlands.

WG2.1 is, so far as I know, the only Working Group to publish its own Bulletin. I am therefore particularly pleased to be able to offer space in this issue (AB42.3.2) to an official pronouncement by WG2.5 (the Working Group for Numerical Software). Although this is primarily aimed at the FORTRAN fraternity, it should be of concern to designers of all programming languages.

In the last issue I published a plea for Algorithms, and the Factorial Program in this issue (AB42.4.6) appears to be the result. As you will see, it is of the specialized nature that I called for, having nothing to do with any method Numerical Analysts might use for that purpose.

## AB42.1 Announcements

### AB42.1.1 Modules and Separate Compilation

An extension to ALGOL 68 for Modules (after the style of AB37.4.1) and Separate Compilation is under consideration by the Working Group's Sub-committee on ALGOL 68 Support, and is expected to be adopted at its next meeting in August 1978. In the meantime, implementors and others interested can obtain a copy of the latest draft by writing to the ALGOL Bulletin Editor. The final proposals, if adopted, will be published in the next issue of the AB.

### AB41.1.2 Errata to the Revised Report

A list of Errata to the Springer Edition of the Revised Report was published in AB41.5.2 (these were all corrected in the SIGPLAN Edition). Since then, various other misprints have come to light. It was decided, however, at the meeting of the Working Group's Sub-committee on ALGOL 68 Support in December 1977, not to issue any further Errata pertaining to minor misprints and clerical errors in the Report. Rather, the ALGOL Bulletin Editor will maintain a cumulative list of all the Errata which have been accepted by the Support Sub-committee, and a copy may be obtained on request. In particular, anyone contemplating making a new printing or translation of the Report should obtain, and elaborate, this list.

It should be noted that none of the items on this list makes any change to the language or resolves any technical problem. For these matters, the Support Sub-committee has decided not to modify the Report itself, but rather to issue Commentaries on specific problems, for the guidance of implementors and others. The first set of such commentaries is published in this issue (AB42.3.1) and it is anticipated that a much larger set, covering in particular the Transput section of the Report, will be published after the next meeting of the Sub-committee.

### AB41.1.3 ALGOL 68 Compiler for the DEC PDP11 Computers

A one-pass compiler for ALGOL 68 on the PDP 11 Computer operating under both UNIX and RSX-11M Operating Systems is now available for distribution.

The Compiler itself was written and developed at Liverpool University (U.K.) and Carnegie-Mellon university. The operating-system interfaces were provided by the University of Manitoba (Canada). The system has been successfully used in a number of programming language courses and as a general programming and research tool.

The source language is an extended version of ALGOL 68S (the official IFIP subset of ALGOL 68). Thus, the major restrictions on the full ALGOL 68 language are the lack of formatted transput, flexible multiples (though strings are provided) and some one-pass implied restrictions on structured and multiple types and on the coercions (implicit type changes) allowed. Features provided which go beyond full ALGOL 68 include parallel processing using "eventual variables" and an interface to Macro 11 procedures.

The system requires no special facilities beyond those available on the simplest PDP 11s. The version currently being distributed requires a space of 32K words (in addition to the operating system) to run. A second version incorporating a loader (thus requiring considerably less space) is in preparation. Those who order the existing system will receive the new version at no additional cost; this version should be available in late

Spring 1978.

Documentation on both source language and installation instructions for either or both operating systems will accompany the distribution tapes.

Distribution details available from:
ALGOL 68 Distribution manager,
Department of Computer Science,
University of Manitoba,
Winnipeg, Manitoba,
CANADA, R3T 2N2.

### AB42.1.4 Conference Proceedings: Vth III Meeting

The Proceedings of the 5th International Conference on the Implementation and Design of Algorithmic Languages, held in Guidel (France), May 16-18, 1977, edited by J. Andre and J. P. Banatre, are now available at 100 Francs per copy.

To order, write to
IRIA-SEFI-Diffusion,
B.P. 105,
F-78150 LE CHESNAY,
France
and include a cheque for 100 French francs (payable to "Agent Comptable de l'IRIA").

### AB42.1.5 TORRIX

TORRIX - A Programming System for Operations on Vectors and Matrices over Arbitrary Fields and of Variable Size, by S. G. van der Meulen and M. Veldhorst, is being published as Mathematical Centre Tracts No. 86. Volume 1 (TORRIX Basis) is available now, and may be obtained from:
The mathematical Centre,
2e Boerhaavestraat 49,
Amsterdam O,
The Netherlands.
Volume 2 (TORRIX Complex, Triangular matrices, Sparse matrices, etc.) will be available later in the year.

TORRIX comprises a complete library-prelude for ALGOL 68 for the efficient handling of vectors and matrices. Volume 1 comprises three chapters describing respectively the Mathematical Foundation, Language Considerations, and a Users Guide (with the complete text of the prelude).

### AB42.1.6 Textbook: Programming and Problem Solving in ALGOL 68

This new textbook, by Professor Andrew J. T. Colin of Strathclyde University, is published by Macmillan in both hardback (ISBN 0-333-21716-0) and paperback (ISBN 0-333-23115-5) editions. The paperback edition costs four pounds and fifty pence. It is intended primarily to teach the art of programming, using ALGOL 68 as the specific language.

In fact, it only describes that part of ALGOL 68 which one would expect to teach in a first-year undergraduate course, so that there is no mention of unions, operator definitions, GOTOs, the heap, or any advanced transput. It is particularly noteworthy for the vast number of examples in the text, illustrating all the common algorithms which students should have at their fingertips. There are also copious excercises at the end of each chapter,

with worked solutions to the more interesting ones.

## AB42.2 Letter to the Editor

### AB42.2.1 Implementation on TESLA 200

Dear Dr. Lindsey,

I take the liberty to complete the list of ALGOL 68 Compilers that was published in AB41.4.6, p. 71-73. the last but one item of the list should have the form:

| | |
|---|---|
| Language | A68 subset |
| Authors | J. Nadrchal et al. |
| Organization | Czechoslovak Academy of Sciences, Prague |
| Start Date | 1970 |
| Finish Date | 1977 |
| Written in | ML/1 macrogenerator, APS assembler |
| Runs on | TESLA 200 |
| Remarks | No heap, union, flex<br>4 passes. |

Yours sincerely,

J. Nadrchal

Czechoslovak Academy of Sciences
Institute of Solid State Physics
162 53 Praha 6, Cukrovarnicka 10
Czechoslovakia

## AB42.3.1 Commentaries on the Revised Report

The following commentaries are issued by the Sub-committee on ALGOL 68 Support, a standing sub-committee of IFIP WG2.1. They deal with problems which have been raised in connection with the Revised Report on the Algorithmic Language ALGOL 68, and mostly take the form of advice to implementors as to what action they should take in connection with those problems. These commentaries are not to be construed as modifications to the text of the Revised Report.

Note that commentaries are not being published on trivial misprints. Those concerned about such misprints (and especially those preparing new printings of the Report) should apply to the Editor of the ALGOL Bulletin for the latest list of agreed Errata.

1) Interruption of loops.

Although the semantics of 3.5.2 suggest that a count of the number of iterations of a loop should be kept even when the for-part and the intervals of a loop-clause are EMPTY, it is clearly unnecessary for an implementation actually to implement the count in this case, and it would therefore be unreasonable for an implementation to interrupt (2.1.4.3.h) the elaboration simply because such a count had overflowed. Thus, the elaboration of WHILE TRUE DO SKIP OD would be expected to continue beyond maxint iterations and would not be terminated unless some othe action of the operator or operating system intervened.

On the other hand, if a for-part or a FROBYT-part is present in a loop-clause an iteration count must be kept and will be subject to the arithmetic limitations of the hardware. If this count should overflow, therefore, it is reasonable for the implementation to interrupt the elaboration under the provisions of 2.1.4.3.h. For example, in:

```
FOR i FROM maxint-3 TO maxint DO print(i) OD
```

the implementation may attempt to compute the quantity (maxint+1) (as is indeed suggested by 3.5.2.Step 4), and it will then be quite justified in interrupting.

2) Plus operator on strings.

The + operator for STRINGs declared in 10.2.3.10.i works with strings whose descriptors are exactly flat (2.1.3.4.c), e.g.:

```
LOC [1:0] CHAR + "abc"    # yields "abc" #
```

but has undefined semantics if a descriptor is "super flat", e.g.:

```
LOC [1:-1] CHAR + "abc"   # should have yielded "abc" also #
```

This is an error in the Report, and implementations should accept all such STRINGs and yield the same result as if LOC [1:0] CHAR had been provided.

## AB42.3.2 Parameterisation of the Environment for Transportable Numerical Software

Editor B. Ford
Prepared and agreed by the IFIP Working Group on Numerical Software (WG 2.5)

### History

An early draft of this note was used as a discussion document during the first meeting of the IFIP Working Group on Numerical Software (WG 2.5) in Oxford in January 1975. The meeting requested a precise specification of the purpose of the note and suggested a number of other improvements which led to a second draft. Written comment led to further changes. A third draft was discussed during a workshop on transportable numerical software in the Applied Mathematics Division of the Argonne National Laboratory in August 1975. A fourth draft was written in January 1976 and distributed widely for comment and criticism. Discussion at the NSF/ERDA workshop on portability of numerical software and at the second meeting of the IFIP WG 2.5 (both in June 1976), together with correspondence from other parties led to the preparation of the present document. Although some of the comment was contradictory, it is our belief that this final document represents a consensus view.

### Objectives

The development of numerical software entails three distinct steps:

(i) the design of the algorithm

(ii) its realization as a documented source-language subroutine or program

and

(iii) the testing of the compiled code on a given configuration (i.e. a machine together with its operating system, compiler and available libraries) and its detailed documentation for that configuration.

While the final code must depend on the configuration, it is very desirable for the source-language version to be transportable (i.e. needing only a small number of mechanical changes prior to compilation on a specific configuration) and for the algorithm to be adaptable (i.e. expressed in terms that permit efficient and accurate computation on any configuration that is used later). Our first aim is to suggest parameters that allow the algorithm designer to make his work adaptable in this sense. For example, he may need the "relative precision" for a root-finding algorithm or "page size" for an algorithm designed to solve linear equations efficiently on a machine with virtual memory.

Our second aim is to provide a larger set of parameters that can be used in the transportable source-language (step (ii)) in the expectation that actual values will be readily available in the eventual run (step (iii)). The set has to be larger since it should include such items as the standard input and standard output units, neither of which are likely to be of concern to the algorithm designer. The exact way that these values will be made available is left open; some possibilities are as function calls, as values in a name COMMON BLOCK and as flags for a preprocessor. In all cases we hope that the actual names we suggest are used.

Our suggested parameters are listed in the table below. In addition to a definition we give a characteristic name for the algorithm writer and an explicit name for inclusion in the source code. These explicit names conform to FORTRAN conventions.

Parameter Selection

The first group of parameters, which we have called the arithmetic set, are intended to provide necessary information about the arithmetic hardware. It should be noted that they are not intended to be sufficient to describe the operation of the arithmetic in full detail, nor is it expected that all will be wanted in any one program. One group of numerical analysts feels that the detail provided by the overflow and underflow threshold parameters is unnecessary and that the range parameter is what is really needed; another group wants this detail. Since range cannot be computed from overflow and underflow thresholds, we decided that all three parameters should be included. Similarly we expect that in many applications the relative precision parameter will be used in preference to the mantissa length but a requirement for the mantissa has been expressed strongly by a number of numerical analysts.

The next set consists of basic input-output parameters likely to be required frequently.

The third set contains miscellaneous parameters. As long as the majority of FORTRAN compilers are designed to meet the ANSI X3.9-1966 FORTRAN standard there is a continuing requirement for a parameter for the maximum number of characters that can always be stored in an INTEGER storage unit. The number of decimal digits accepted by the compiler is needed for preprocessors generating specific code for a configuration. The page size may be required to ensure that an algorithm performs efficiently on a virtual memory machine. We suggest that this parameter is given the value of 1 for non-paging machines.

Representation and Arithmetic Operations

We use the systems of INTEGER, REAL and DOUBLE PRECISION numbers as described in the ANSI X3.9-1966 FORTRAN standard. For the system of INTEGER numbers we consider the arithmetic operation a $\square$ b, where $\square$ belongs to {+,-,x}, and the monadic operation -a. The computed and stored result can be computed exactly. For the systems of REAL and DOUBLE PRECISION numbers we consider the arithmetic operation a o b, where o belongs to {+,-,x,/}, and the monadic operation -a, to be performed correctly if the computed and stored result can be expressed exactly as a $(1+E')$ o b $(1+E'')$ and $-a(1+E''')$ respectively, where $|E'|$, $|E''|$ and $|E'''|$ are at most comparable to relative precision.

| Characteristic Name | Definition | Explicit Name: INTEGER | REAL | DOUBLE PRECISION |
|---|---|---|---|---|
| ARITHMETIC SET | | | | |
| Radix | Base of the floating-point number system. | - | SRADIX | DRADIX |
| Mantissa Length | Number of base-RADIX digits in the mantissa of a stored floating-point number (including, for example, the implicit digit when the first bit of the mantissa of a normalized floating-point number is not stored). | - | SDIGIT | DDIGIT |
| Relative Precision | The smallest number x such that 1.0-x < 1.0 < 1.0+x where 1.0-x and 1.0+x are the stored values of the computed results. | - | SRELPR | DRELPR |
| Overflow Threshold | The largest integer i such that all integers in the range [-i,i] belong to the system of INTEGER numbers. | IOVFLO | - | - |
| | The largest number x such that both x and -x belong to the system of REAL (DOUBLE PRECISION) numbers. | - | SOVFLO | DOVFLO |
| Underflow Threshold | The smallest positive real number x such that both x and -x are representable as elements of the system of REAL (DOUBLE PRECISION) numbers. | - | SUNFLO | DUNFLO |
| Symmetric Range | The largest integer i such that the arithmetic operations □ are exactly performed for all integers a, b satisfying $\|a\|, \|b\| \leq i$ provided that the exact mathematical results of a □ b does not exceed i in absolute value. | IRANGE | - | - |
| | The largest real number x such that the arithmetic operations o are correctly performed for all elements a, b of the system of REAL (DOUBLE PRECISION) numbers, provided that a, b and the exact mathematical result of a o b do not have an absolute value outside the range [1/x, x]. | - | SRANGE | DRANGE |

| Characteristic Name | Definition | Explicit Name |
|---|---|---|
| INPUT/OUTPUT SET | | |
| Standard input unit | The logical unit number for the standard input unit of the system. | NIN |
| Standard output unit | The logical unit number for the standard output unit of the system. | NOUT |
| Standard error message unit | The logical unit number for the standard error message unit of the system. | NERR |
| Number of characters | The maximum number of characters, including blanks, which can be read from a single record on logical unit NIN. | NCNIN |
| Number of characters | The maximum number of characters, including blanks and carriage controls, which can be output to a single record on logical unit NOUT. | NCNOUT |
| MISCELLANEOUS SET | | |
| Number of characters per word | The maximum number of characters that can always be stored in an INTEGER storage unit. | NCHAR |
| Page size | The size of storage defined for use within the paging algorithm of a virtual storage system, in INTEGER storage units. On non-paging machines it has the value 1. | NIPAGE |
| Number of decimal digits | The largest number of decimal digits allowed and converted by a given compiler when compiling constants. | INTEGER: NIDEC; REAL: NSDEC; DOUBLE PRECISION: NDDEC |

# AB42.4.1 Trouble Spots in the Standard Hardware Representation for ALGOL 68

Wilfred J. Hansen
University of Illinois at Urbana-Champaign
June 1977

A number of implementors have written to me with various comments about the SHR, usually complaining that some feature is surprising. This note tries to collate these comments and--where appropriate--discuss why the SHR is the way it is and what action a compiler might take. (For a notation I use "|x|" for hardware representations and "/ x /" for the corresponding ALGOL 68 symbols.)

First I want to mention the treatment of errors. No concept of degree of erroneousness was recognized in the Report on the SHR, but I feel that a number of usages, while questionable, ought to be accepted with no more than a warning message. It is better to get a warning than some of the other indications of difficulty that might arise (e.g., "/ e3 / cannot be identified" is less clear than "point ambiguous in |4.E3|").

## Problems with all regimes

|.X_Y| ⇒ / x y /, with a warning about the incorrect underscore. I suggest that the point strop ought to have more importance than the underline for determing intended meaning.

|4.xy|, where |xy| cannot continue a real denotation. The appropriate treatment depends on the nature of |x| and |y|. If |x| is |E| and |y| is a digit or a sign, then a warning is imperative. It cannot be rejected as illegal, however, because |4.E0| ⇒ / 4 e0 / may be part of a program where / e0 / is a TAD. In all other cases, the interpretation / 4 xy / without a message is appropriate. Indeed, if there have been no operator definitions, conversion of a possible real to an integer is always satisfactory. Where real denotations may precede bold words, they can always be widened from integral.

|"" " " "" ""| ⇒ / " " /, i.e., a character denotation. Some object that this is silly, and I agree, but we never bothered to outlaw it. A warning for complex character denotations would not be inappropriate.

|.GO TO| ⇒ / go to /. Probably a warning isn't even needed (thanks to D. Taupin).

Problems with UPPER stropping

|.aB| ⇒ /a b /. This treatment seems to be a logical consequence of the decision that /reOFz/ is to be a selection. The "case disjunction" is treated everywhere as a break between tokens. Otherwise, we felt, a token scanner would have to have a special exception for when the case changed and the first token was stropped. Note that |.a3B3| goes to /a3 b3 /.

|a_B| ⇒ /a b/. The SHR dictates that this is illegal even though |aB| and |a_ B| are allowed. Though the illegality is not very defensible, it came about from following the principle that "bold cannot be adjacent to underline." Compilers can certainly accept it without great harm.

|AT1| ⇒ /at1 /. There is an existing compiler that treats this as /at 1/, but this clearly does not satisfy the Report itself because there is no way to write the /at1 / symbol. There must be bold digits, and with upper stropping they can only be recognized by the fact that they follow bold letters.

|.pr| ⇒ /pr/. The objection to this is that |PR| is available so why is the stropping permitted. Indeed if there is only one case, it is upper case so there is still no problem getting /pr/. What we did was to follow the principle that explicit stropping always works. This helps make the three regimes closer, so a single scanner can be used. It is a tad inconsistent to allow this, while forbidding intimidation with underscores, but that is what we did. In our scanner design it was easier to allow the point strop at all places than to forbid it in some.

Problems with REServed stropping

|.a3B3| ⇒ / a3b3 /. That this is inconsistent with UPPER stropping is truly unfortunate, but otherwise this treatment is very reasonable. The alternative seems to be to forbid |reOFz|.

| int I; read(i)| ⇒ / int i; read ( i ) /. i.e., case is ignored except in UPPER. Distinguishing identifiers by case was thought to be too prone to possible error. It is already true that variations in spacing are not significant, i.e., that |as in| is the same as |a sin|.

|+3END| ⇒ /+3 end /. This is illegal, but compilers can certainly accept it if their implementors so desire. That it is forbidden is probably an oversight.

AB42.4.2 ALGOL 68 Hardware Representation Recommendations

Wilfred J. Hansen

University of Illinois at Urbana-Champaign

During my work on the Report on the Standard Hardware Representation for Revised ALGOL 68, I encountered a number of ideas which--though meritorious--could not be agreed on by the Subcommittee on ALGOL 68 Support. I have collected a few of these here in the hopes that where implementors choose to provide such a feature they will all provide it in the same way.

1. TITLE's

A valuable facility for identifying listings is to have a title at the top of each page. Therefore, the pragmat

PR TITLE string-denotation PR

should be the last line on its page. The string_denotation should appear in the header of succeeding pages. The provision that it start a new page is followed in a number of implementations of other languages and works well in practice. (A number of people suggested this, and I have forgotten who was first.)

2. 48-Characters Representation

At least two popular languages require only 48 characters and some devices provide only those. Usually they are

letters digits space ' $ ( ) * + , - . / =

Of the 23 special characters which are worthy, 10 are available in the 48 set, six can be avoided by using alternate constructs with bold words (lt, at, if, mod, co, etc.), and three can be represented with alternate characters from the 48 set:

" → '  [ → (  ] → )

Colon and semicolon can be replaced with diphthongs as suggested in the original Report:

: → ..  ; → .,  := → .=

Though only the first is still mentioned in the Revised Report, all three are non-ambiguous. The third is a special abbreviation for the very common assignment operation, but "..=" should also be acceptable. To resolve the ambiguity inside strings, it should be the case that upb "...," yields four. There is the additional problem that "3.,4" might be either an error or a series; especially curious is

(i|3.,7.,2|5.).

Presumably a compiler will have to interpret these as semicolons so the expression is an *if* rather than a *case*.

The remaining two worthy characters, underscore and apostrophe, have no reasonable representation in the 48 set. Thus, there can be no string escapes and reserved words cannot be intimidated to plain in the RES stropping regime.

### 3. *up* and *down*

For some reason the Report provides both *up* and *shl* for shift-left-operator and both *down* and *shr* for shift-right-operator. But they are not synonymous because only *up* and *down* can operate on semaphores and because *up* can signify exponentiation while *shl* cannot.

I cannot understand the need for this tangle of relationships, especially when *shl*, *shr*, and ** are available and are--to me at least--more mnemonic. Therfore, I suggest that, even if implemented, *up* and *down* should be restricted to their operations on semaphores. That is, their other uses should be down-played to the extent of being mentioned in some lowly place like a single small footnote.

### 4. String Escapes

The Standard Hardware Representation requires that a string escape--if it is allowed--must begin with an apostrophe, but make no further specification. There were two reasons: firstly, it was felt to be too machine dependent, and secondly, the authors of the Report on the SHR couldn't agree as to whether prefix or circumfix apostrophes should be used. My own feeling is that the Cambridge scheme is excellent and machine independent. It extends to providing two cases if it is modified to the following form:

```
''      → apostrophe
'/      → new line
'%      → new page
'+      → carriage return (no line feed)
',      → tab
'-      → backspace
'␣      → ␣ (where ␣ represents a space)
'int␣   → {the number of spaces specified by "int"}
'(list of character codes separated by commas)
        → {the indicated characters}
```

Note that '/ and '+ have the functions that slash and plus have when used as ASCII carriage control characters. '% seems a logical extension and the rest seem to be pleasantly mnemonic. I would allow the character codes to be specified as integers, bits-denotations, or by mnemonic name (DC1, VT, ...), but this can be left for implementation definition.

By changing the Cambridge convention to use all special characters or digits following the apostrophe, we achieve the desirable goal of a representation for upper and lower case strings. We simply specify that a letter alone is the lower-case version of the letter and

'letter → upper-case of letter.

On one case devices, all output strings will print in a single case, but the system will be ready for two case devices. (Apostrophe for lower case is currently used on at least the DEC-10.)

5. of

I included a long discussion of the of problem in the Proceedings of the 1975 International Conference on ALGOL 58, Oklahoma State University, pp. 335-337. Essentially, there must be a special-character representation of of because the operation binds so much more closely even than monadic operators and yet the bold of is at least four characters long (unless the crowded case disjunction is used as in "reOFz").

As we worked on the Standard, it became apparent that the best alternative for of is the character that is worthy and yet has no meaning outside strings: apostrophe. Part of the function "value of" in the formula manipulation example of R11.10 then becomes.

```
BEGIN REF FUNCTION f = function_name'ef;
    value'bound_var'f := value_of(parameter'f);
    value_of(body'f)
END
```

Since the "operand" to the left of the of must be a selector, and since a selector must be followed by of, there is little purpose to having a more obtrusive symbol.

For stropping regimes that still use apostrophes, the best proposal that has appeared is the "./" that I proposed at Oklahoma State. Only six of the 25 implementors who voted agreed that it was first choice, but no other symobl had as many votes. In the absence of a clearly superior symbol, I suggest that "./" be adopted as the non-bold of:

```
BEGIN REF FUNCTION f = function name./ef;
    value./bound var./f := value of(parameter./f);
    value of(body./f)
END
```

Among other advantages, "./" is entirely in lower case positions on most keyboards, just as are the letters which are the characters usually surrounding it. Please see the Oklahoma proceedings for further discussion of the merits of these two representations.

AB42.4.3 Corrections to and Discussion of "A Token Recognizer for the Standard Hardware Representation of ALGOL 68"

R. Bell
Algol Bulletin 41.4.5 1977 pp 47 - 70

**Corrections** evoked by a letter to the author by Dr V.V. Brol, and otherwise.

**Pages** 48, 49, 52, 62.

```
# integer-denotation ⇒ integral-denotation #
```

(Occurs only in pragmatic sections and comments)

**Page** 51↓23 after C insert a semicolon

(between declarations of 'chartype' and 'uletter' etc)

**Page** 51↑5

```
# uletter OR sletter ⇒ uletter(c) OR sletter(c) #
```

**Page** 55 (In PROC tagscanner)

```
line ↑15    ....  IF    NOT eol
     ↑14              AND char = underscore
     ↑13          THEN  get next character (char)
  insert                                      ;
    "                   IF upperstrop AND uletter(char)
    "                   THEN C emit a warning message
    "                          (space or point required
    "                           between tag ending with underscore
    "                           and upper case letter
    "                           starting bold word)   C
    "                   FI
  end of insert
line ↑12          FI ;  ....
```

(See HR 3.5.2. Such an error causes no ambiguity in token recognition, therefore a warning rather than a fault message is suggested. One might ask why this requirement is imposed: is it perhaps only to simplify translations into other stropping regimes? )

**Page** 62↑20 (In declaration BOOL expart )

```
#aletterproc AND char = "E"
 ⇒
 BOOL  aletter = aletterproc ;
 aletter AND char = "E"        #
```

('aletterproc' must be elaborated first, to replace "e" by "E" if necessary, before 'char = "E" ' is elaborated).

**End of corrections**

**Further responses to Dr Brol**

1

Brol: "Why does tag processing continue when an incorrect underscore appears (e.g. 'a__b') ?
HR obviously forbids this".

Reply:
See page 55, PROC break in tag .

(a) Consider first POINT or UPPER stropping
(page 55, 'break in tag' called in PROC tagscanner )

I understand that you would analyse ⫤a__b⫥ thus :-

a_ ⇒ ⟨taggle⟩

_ incorrect exit from PROC tagscanner

⇒ ⟨tag⟩ equivalent to ⫤a⫥

b ⇒ ⟨taggle⟩

⇒ ⟨tag⟩ ⫤b⫥

Whereas I analyse it as

a_ ⇒ ⟨taggle⟩

_ unwanted discarded, with a warning

b ⇒ ⟨taggle⟩

⇒ ⟨tag⟩ equivalent to ⫤ab⫥

Now the concatenation ⟨tag⟩⟨tag⟩ is nowhere syntactically correct in Algol 68.
I think that a programmer is most likely to put ⫤a__b⫥ where he means to put ⫤a_b⫥ or ⫤a_ b⫥ , both of which are equivalent to ⫤ab⫥ . In such cases my analysis preserves the programmer's probable intention, which I think is preferable.

(b) Under RES stropping
(page 59, 'break in tag' called at line 16)

If for example the programmer intends
⫤for i_ to n⫥
but by mistake delivers ⫤for i__to n⫥
then
my way will still give the same result as if the extra underscore had not crept in;
what your way would give would depend on how you recover on detecting the error.

(c) I suggest a warning rather than a fault message because this is only a weak error.

2

Brol: "For the procedure 'get next character',
(a) Why does this procedure need the parameter (actual parameter of every call is 'char') ?
(b) It is more convenient and effective in many cases when - this procedure yield CHAR,
- the event routine 'on line end' make 'char := eol' where 'eol' is some available character."

Reply:

(a) I agree that for my own purposes I need not have the REF CHAR parameter; but if anyone else wants to add to my work, for example routines to deal with format texts, he may find it helpful that 'get next character' has this parameter.

(b) To make 'get next character' PROC CHAR instead of PROC (REF CHAR) VOID will help in some places, but I think there are just as many places where it would make me put 'char := get next character' instead of 'get next character(char)'.

To make 'eol' some'available character' is not good. The difficulty is to choose an 'available character'. This must be implementation-dependent, even device-dependent. One would obviously choose a character not used in any Algol 68 symbol, and this character could not then be allowed as a string-item. On implementations with small character sets this could make difficulties. This trick could wreck the portability of the token scanner.

Brol: "Why do you use so many generators and never use variable declarations ?"

Reply: I agree that it is pedantic. It is also because I normally have to use the Algol 68-R compiler, which is older than Revised Algol 68 and does not allow me to use the construct 'LOC INT i' instead of the construct 'REF INT i = LOC INT'. I could simply use 'INT i' but dislike this, especially when teaching.

Brol: "Have you a method of design of a format scanner?"

Reply: I have not thought about this at all.

Use of the word procedures

I specify that for each different "word" there is to be a unique procedure of mode PROC(WORDPARAMS)VOID, and that the token recognizer is to deliver a STRUCT(STRING repstring, PROC(WORDPARAMS)VOID wordproc) value including the appropriate procedure for every word it extracts from the program text. (All tags deliver the procedure 'tagproc', all bold-tags deliver 'boldtagproc', and so on).

I have not defined the routines to be ascribed to these word procedures.

Suppose we intend to use the token recognizer to feed a (mainly) top-down syntax analyser.

It is well known that the syntax rules used in RR to define Algol 68 lead to an inefficient syntax analyser if applied directly, so let us presuppose some equivalent but more applicable set of rules. From these rules a "syntax-graph" can be conceived. This graph may be represented explicitly by a linked-list-structure, or implicitly in the structure of the syntax analyser algorithm.

Every syntactically correct (particular-)program-text has an equivalent finite production-tree, and every such production-tree corresponds to some path through the syntax-graph. From successive words in the program text (extracted by the token recognizer) the syntax analyser computes the path and hence the production-tree. That is to say, that having reached some vertex in the graph, the analyser will use the next word from the program text to decide which of the possible onward-leading edges to choose, and so will move on to another vertex.

A unique (and hence identifying) set of attributes can be ascribed to each vertex; let this set of attributes of a vertex be called a "context". Hence at each step in the pathfinding two arguments are known, the current context and the current word, from which the next context is computed. (I think we can build error-recovery into the syntax-graph).

Now suppose that the syntax analyser is to be written in Algol 68: the programming device that I have in mind is as follows.

To represent a set of attributes it is surely natural to use a structured value. I think that a variety of modes of structures will be needed to represent all the various contexts. In fact I shall go to the extreme and arrange that for every distinct context I shall have a distinctive mode of structure (then each mode-indication will be as good as a context identifier). It follows that the mode of all contexts must be the union of all these modes of structures.

Now understand why I declare MODE WORDPARAMS = UNION( ? ? ?) : this is my mode for the contexts, which will all be structured values.

An afterthought now comes to me, that the mode for the word procedures had better be PROC(REF WORDPARAMS)VOID , which is a trifling alteration from the point of view of the work done so far, but will make things more convenient for future work on a syntax analyser.

Use of word procedures (continued)

A step in the pathfinding may be programmed in roughly this way :-

```
get word (next word # a REF WORD variable # ) ;

   # Now we do not have to enquire what class of word
     has been extracted
   #

(wordproc OF next word) (context # a REF WORDPARAMS variable # ) ;

   # Calls the word procedure, whatever it is.
     The word procedure will probably assign a new value
     to 'context': a structured value, but not of the
     same mode as the superseded structured value.
   #
```

The unit of the routine of every word procedure will be a conformity clause:-

```
PROC tagproc = (REF WORDPARAMS context) VOID :

     CASE
        ... ;
        context

       IN

        C One case part for each mode representing a context
          in which a tag is syntactically acceptable
          #recall that every context has a different mode# C

       OUT

        C Error recovery routine for all those contexts in
          which a tag is wrong C

     ESAC
```

(By no means original, see for instance Informal Introduction, 2nd Ed., Section 1.6.2).

What is happening is that the hidden mode-representation which is assigned to a reference-to-UNITED variable is being made to work harder, by indicating context values.

AB42.4.4

# ALGOL 68 and your Friendly Neighbourhood Operating System.

C.H.Lindsey.
University of Manchester.

## 1. Introduction.

The purpose of this paper is to consider the interface between the ALGOL 68 transput system, as defined by the Revised Report, and the features provided by real-life operating systems which are not always particularly friendly when seen from an ALGOL 68 point of view. I start from the definition of the Transput as given in R10.3, and I try to identify the features of that Transput with which actual operating systems might have difficulty. My purpose is to list the possible courses of action open to an implementor in each such situation, to indicate their relative merits and, in particular, to establish their legality, i.e. whether they imply an actual violation of the Report. It is only when we have established the precise extent to which the present Report copes with these various situations that we can begin to discuss whether the Report itself is inadequate. I have tried to include all the implementation difficulties which have been reported recently (but bear in mind that I am restricting myself to operating system interface problems).

## 2. The Gremlins.

A proper understanding of the role of the Gremlins is essential to any understanding of the relationship between an operating system and the Transput defined by the Report.

{For the benefit of readers not familiar with RAF slang as developed during the last war, it should be explained that "gremlins" are mythical creatures of unfriendly and anti-social disposition (somewhat related to hobgoblins, trolls, and other such) which were first discovered during the 1940s inhabiting the (somewhat unreliable) radar equipment of that time. They have been variously accused of causing component failures, dry joints, spurious signals and even, on occasions, of interchanging connections and causing short circuits. I have no doubt that there are still preserved, at the Royal Radar Establishment, Malvern, Top Secret documents which the public may never be allowed to see, describing the first discovery of these malignant creatures.}

In the Report, the Gremlins are a semaphore which, when upped, releases a call of "undefined" (R10.4.2.a). This call is contained in a loop which is elaborated continuously in parallel with the users' programs. It is important to realise that, in spite of the very broad wording of R10.3.1.4.a, this call of "undefined" is very limited in the damage it can do. Because it is elaborated in an environ very close to the primal environ, it can only see objects declared in the standard-prelude, the librarary-prelude or the system-prelude. Therefore it can neither see nor interfere with objects declared within any user-task (i.e. it has no access to anything which is kept upon the stack of any particular-program) and it is therefore unable to cause arbitrary disruptions to the user's intentions.

What then can the Gremlins legitimately do? The only variables declared within the standard-prelude are "chainbfile" and "lockedbfile", but it may be presumed that the Gremlins are aware of the processes going on in the other system-tasks (R10.4.2) and are able to communicate with those processes by means of variables declared in the system-prelude. Therefore, they are able to tinker with the chain of bfiles, adding and removing links,

and they may arrange to keep a private copy of the chain so that they can detect when alterations to the official chain have been made by "open", "establish", "close", etc. Thus they may be presumed to be aware of all books within the system, whether opened or not, and so they are able to modify the contents of books (even opened ones - but that would be rather unfair) and they can alter the physical size of books and move the position of the logical file end.

## 3. Friendly and unfriendly operating systems.

The purpose of the Gremlins is to model those features of operating systems that are beyond the control of the language designer. One can say that the intention (or rather the hope) of the Editors of the Report is embodied in that Transput system which would be defined by the Report if the call of "undefined" in the Gremlins never performed any action at all. I will define a "friendly" operating system to be one which would permit an exact implementation of that intention.

Alas! Actual operating systems are never so friendly (and implementors of ALGOL 68 are not the only ones to have found them so). An "unfriendly" operating system, therefore, is one in which it has to be supposed that the Gremlins have taken various specific actions in various specific situations. The more outrageous the actions the Gremlins must be supposed to undertake, the more unfriendly that operating system can be said to be.

However, since the Gremlins are indeed defined, even the most unfriendly operating system need not be in violation of the Report. Nevertheless, implementors should be encouraged to be "reasonable" in deciding how to circumvent the problems posed by their operating systems. That is to say, they should only take refuge behind some action of the gremlins when some feature of their operating system genuinely prevents them from doing otherwise. We know that real operating systems are indeed unfriendly, and that is what the Gremlins are there for, but nevertheless the Gremlin-free implementation is the ideal which implementors should try to approach. The same principle of reasonableness should be applied in all the other parts of the transput definition which apparently give freedom for outrageous interpretations.

## 4. Sequential files.

I take it for granted that all operating systems provide a (more or less friendly) mechanism for storing files of textual information, for retrieving them on demand, and for inputing them from and outputing them to external devices. There will also be means for restricting access to files by unauthorized users. Although an ALGOL 68 implementor might be able to implement a closer approximation to the transput defined by the Report by ignoring his operating system conventions and constructing files in some private format, this course of action would hardly be appreciated by the majority of his customers. In the case of random access books (with "set possible"), which are a peculiarly ALGOL 68 phenomenon which may not fit in well with the operating system, private formats may be justified, but not so in the case of sequential files. I shall discuss random access books in a later section. In the meantime, please assume that my remarks apply to sequential access books only, and that I am concerned only with attempts to implement them in conjunction with the file system provided by the operating system.

I shall periodically give examples from particular operating systems, usually those with which I am most familiar namely GEORGE3 (ICL 1900 series), NOS (CDC machines) and RSX11 (PDP11). In the case of RSX11, what I

will really be talking about is its Record Manager, known as FCS (File Control Services).

It is clear that an ALGOL 68 source text on its own is insufficient to cause a system to execute a complex program. There will be a necessity for Control Cards, or even for more complex sequences of statements in some Job Control Language, in order to control, in particular, the connection of books and physical devices to the program. In what follows, I presume that such external commands are indeed provided as may be appropriate for the particular operating system, and I shall refer to all such external sources of knowledge simply as "JCL".

## 5. An analysis of unfriendly features.

### 5.1. Opening of files.

Opening causes a file to be attached to a book via a channel. The Report envisages that the book may be a file entrusted to the safe keeping of the operating system and retrievable on demand, or it may be an online physical device which generates (or disposes of) the book line by line in real time, or it may even be that "set up in nuclear physics". The Report provides no explicit mechanism, apart from the "idf" parameters of "open" and "establish", for indicating which of these situations is intended in each particular case, and in the case of "create" and the files "standin" and "standout", opened outside the user's control, even the "idf" parameter is not available. Clearly, therefore, information in JCL is expected to be provided. Similarly, in the case of books output from the program, the Report makes no provision for disposal instructions (to cause the book to be listed on a lineprinter upon completion of the program, for example) and here again JCL is obviously needed.

The features provided by various operating systems to make these various attachments and disposals and the JCL whereby they are accomplished are exceedingly diverse. It is generally best that an ALGOL 68 implementor should try to fit in with the conventions of his operating system. Thus the JCL he expects should be in a form similar to that required for other languages under that system. There remains the question of what interpretations he can reasonably place on his "idf" parameters.

The Report, in the procedures "idf ok" and "match", deliberately allows considerable latitude here. It is sometimes the case that the customer for a program is in the best position to know which particular books should be selected for a particular run, and he should then provide JCL accordingly. On other occasions it is the writer of the program who knows best, either because the program is always intended to operate with the same books, or because the title of the book is to be computed by the program, or because an indeterminate number of books is to be opened in succession. In these cases, the information should be provided in "idf" parameters, from which it is clear that it should be possible to bring about, by writing suitable "idf"s, anything that could have been brought about by JCL. In other words, the contents of "idf" strings should look rather like fragments of JCL. Even disposal instructions could be included if operating system conventions allow.

Here are some examples of what might be reasonable:

```
GEORGE3
      establish(
         ":MBACSL.FILE NAME-2(+1/AL68)(ALLCHAR, TRAPGO(:MBACT, READ))",
         chan, p, 1, c)
```

(which, being interpreted, means "establish a file with filename "FILE

NAME-2" under user name ":MBACSL" with generation number one more than the latest existing generation number of "FILE NAME-2" and with language code "AL68". The file is to be in the character code which permits the full ASCII set of 128 characters, and permission is to be given to user ":MBACT" to read it. Hardly what one would expect in a portable program, but all good familiar stuff to one who is used to GEORGE3.)

RSX11

establish("DK1:[112,1]FILENAME2.A68;7", chan, p, 1, c)

which specifies a physical device, a user name, a filename, an extension and a generation number.

OS

OS distinguishes between "dsnames", by which files may be permanently identified, and "ddnames" with which dsnames, or maybe physical devices, may be temporarily associated by JCL statements. Either may, on occasions, need to be used within "idf"s. FLACC distinguishes them by preceding dsnames with an "*". Indeed, I see no reason why one should not go further and permit complete DDs within an "idf".

Although the Report makes provision for many channels in an implementation, it will be natural for users to expect to use "stand in channel" and "stand out channel" for all normal transput. Therefore these channels are likely to encompass the features of all other channels, files opened via them having their properties ("possible"s, etc) determined by JCL. Cases where specialized channels might be included in the implementation are where it is to be specified that Carriage Control information is to be output, or expected on input, and a "read only channel" for which "put" would be most definitely not possible (assuming that this was not already a property of "stand in channel").

All the opening procedures, as well as many others, start by testing whether the file is already opened. Clearly, a newly declared and not yet initialized file should appear to the system as not opened, even though this requirement is only hinted at rather than explicitly prescribed in the Report.

## 5.2. Close, Scratch and Lock.

The intention of the Report is that,
(i) after "close", an attempt to re-open the same book should succeed,
(ii) after "scratch", such an attempt should never succeed,
(iii) after "lock", such an attempt should only succeed after permission has been given by some operating system action.

However, since all three of these procedures up the Gremlins, the implementor may legally make any of them have any of those three effects.

What ought the implementor to do in practice? Clearly, he should implement the intention of the Report so far as he is able, but there may well be circumstances which prevent this. Here are some examples:

A. The book is in fact an online device (e.g. a line printer), or a spooling system leading to such a device, which keeps no permanent record of the text sent to it. Clearly, the effect of "close" will be the same as the effect of "scratch".

B. The operating system, being unfriendly, does not provide any means for user programs to delete permanent files. The effect of "scratch" will be the same as that of "close".

C. The particular user does not have permission to delete the file. The effect of "scratch" will be the same as that of "close" (a diagnostic message in addition might be appreciated).

D. The operating system has no way of implementing "lock". Therefore, it must implement "lock" like "close". Note that, although

it is often said that "lock" is unimplementable on most operating systems, it seems to me that most systems have some means of preventing unauthorized reading/writing of files, and a reasonable implementation of "lock" would be to remove the particular user's own permission to read (or at least to overwrite) it. Presumably it would then remain unreadable by him until he gave himself permission to use it again. If the book is an online magnetic tape, "lock" should certainly cause the tape to be demounted (even if "close" didn't), and it might also modify the Volume Label so as to prevent re-writing prior to some expiry date.

Note that none of the suggestions given above implies any violation of the Report. It has also been suggested that "scratch", in addition to deleting the book, should also override any disposal instructions (for example, by aborting any listing on a line printer that had been called for). Again, this is legal, and even reasonable.

## 5.3. Lines.

Books to be read may have been input from external media (by some system task), or they may have been created by a program written in some other language, or they may have been created by another (or even the same) ALGOL 68 program. In the first two cases, they contain what they contain, and the ALGOL 68 system can only presume that this coincides with what they were meant to contain. In the third case, however, the Report makes it clear that, in a friendly system, what is read back is meant to be exactly that which was previously written out - every last character of it, with line and page boundaries in their correct places.

Unfriendly operating systems may contain the following features:

A. Lines may be padded with blanks to the next multiple of the word length (as sometimes in GEORGE3).

B. Trailing spaces may be removed from the line entirely, or down to a multiple of the word length (e.g. card input in GEORGE3, the SORT utility in NOS).

C. Empty lines cannot exist (MTS), or they subsequently get deleted from the book (the SORT utility in NOS).

D. Certain characters may have peculiar effects (e.g. colons in Z-type records in NOS).

Note that B is nothing to do with the effect prescribed when the book is "compressible". The Report clearly distinguishes between writing a lot of spaces (which are not compressed away) and not writing at all (the line is compressed to the last character actually written). It is expected that implementors will generally make sequential channels compressible except where they are specifically writing to fixed width devices such as card punches.

Apart from storing books in a private format, with length counters with each line (a practice not to be recommended), there is not much an implementor can do about these things. Suppose, then, that he entrusts each line to his operating system as it is completed. In a sequential book, he cannot then return to read that line without first closing and reopening, or else resetting. If he closes, then the gremlins are upped and they may be deemed to have modified the text of the book to be that which the operating system had actually stored. Thus there is no violation. If he resets, then the gremlins are not upped and there is a clear violation. Of course, he is not obliged to permit resetting of such books, but I feel it would be better to do it wrongly than not to do it at all. It could well be argued that there should have been an up gremlins before the final fi of 10.3.1.6.j ("reset").

5.4. Pages.

Page boundaries are primarily of interest in connection with line printers which have facilities for rapid paper movement to the head of the next form. The question at issue is whether books which are ultimately intended for printing can be held in the meantime in a form which preserves the page boundary information. If this is not possible at all, then it is perfectly legal for the implementor to decide that all his books shall consist of exactly one page of some large number of lines (indeed, this is what Cambridge have done). In Report terminology, he merely has to decree that "max pos" of each of his channels should always return a "p" field of 1.

To implement pages properly, it is only necessary to mark the positions of the page boundaries in the book. One of the following methods is usually available:

A. Explicit Carriage Control (CC) characters (a la Fortran) at the start of each line. These usually provide for advance of 0, 1, or 2 lines or paper throw before printing the line. (NOS, OS, RSX11). Unlike Fortran and some (erroneous) implementations of COBOL, it is clear that the ALGOL 68 programmer is supposed to be totally unaware of their existence. Unfortunately, these systems also permit books without CC characters (indeed this is the normal situation for input files and for files not intended for printing), but only OS and RSX11 keep a suitable bit in the directory to indicate whether the file is a CC one or not.

B. Implicit Carriage Control characters (GEORGE3). Although stored for all files, the CC character is not handed over to file readers as a character, but may be inspected by special request.

C. By use of an explicit character in the text (e.g. ASCII FF).

D. By use of segmented files. This is a feature of NOS and is used by the PASCAL compiler but not, regrettably, by the CDC ALGOL 68 compiler (except for certain very specialized channels). A utility is provided to print a segmented file, complete with CC. This method is probably nearer than any other to the model of the Report.

B, C and D should provide little difficulty for the implementor; neither should A if the system can indicate whether any given book has CC characters (as in OS and RSX11). In other cases of A, the implementor will presumably provide a "printer channel" which puts out CC characters as appropriate. In that case, he ought also to provide a "CC input channel" to read such files, stripping off the CC characters and creating page end events appropriately. It is then up to the user to know which way each book was written, and to use the appropriate channel. If he uses the wrong one, his implementor will have to explain to him that the gremlins have removed all the page boundaries, or inserted extra characters at the start of each line, as the case may be (thus there is no violation). It also has to be decided whether, in such implementations, "stand out channel" has the CC property or not (certainly, "stand in channel" should not have it, except perhaps with the aid of JCL information).

The Report model admits the possibility of empty pages which contain no lines (just as it admits the possibility of lines with zero characters and books with zero pages). This is clearly very othogonal (and tidy), but may not be implementable except in cases C and D above. Clearly, the implementor has little choice but to substitute a page containing one empty line whenever a user creates such a page. If the book is closed and re-opened between writing this page and reading it, there is no violation because the Gremlins can be blamed, but in the case of "reset" there will be a violation, which again suggests that the Gremlins ought to have been upped in "reset".

If a book being written is "compressible", then pages are compressed to the number of lines actually written (personally, I would have preferred to have said that pages contain as many lines as are written to them (up to some physical limit) and that, in non-compressible books, they are made up to the maximum with empty lines). Note that there is no provision for compressing lines but not pages, or vice versa. Personally, I think this is reasonable, since I expect sequential books to be compressible in most implementations anyway.

## 5.5. Logical File Limit.

The Report permits the logical end of file (a point just beyond the furthest character historically written to the book) to be at any position within (or just outside) the physical book. Insofar as the implementor maintains a buffer for the line currently being processed, he can easily maintain a pointer indicating the position of the logical end within this buffer, or indicate in a flag that the logical end is not within the current line. It may be, however, that his operating system cannot distinguish between a logical end that is at the end of the last line actually written out, and a logical end that is presumed to be at the start of the (non-existent) line after the last. In the case of operating systems which actually store the CR/LF characters in the record (some options of RSX11), the distinction can be maintained. In other cases, it will be best for the implementor to assume that the LFE is after the last line actually written, since the operating system will presumably inform him as soon as he tries to refill his buffer from the non-existent line, which is just in time to provoke the logical file end event according to the Report. If a user writes a book and leaves the LFE at the end of a line, therefore, the Gremlins must be supposed to have moved it upon closing the file. In the case of a file that is "reset" after writing, the implementor could presumably remember the (p, l, c) of the actual logical end, but I feel it would be better to accept the slight violation involved in assuming the Gremlins to have been uped in "reset".

It is always the case that the exact position of the LFE only needs to be known when it is within the current line. In a sequential book being written to, it always is. In books being read it may be in the current line or some unknown distance ahead; the operating system must then indicate as soon as the line containing the LFE is read. Even in the case of a call of "newpage", which apparently may involve a leap beyond the LFE, the implementor will in fact read successive lines of the book until either a page marker (CC character or explicit FF) or the LFE is encountered, and then take appropriate action according to which it is.

## 5.6. Physical file limits.

The physical size of a book embodies information concerning the actual number of characters in each line, lines in each page, and pages in the book. For lines and pages up to and including the logical end of the file this information is historical - i.e. these numbers reflect the actual text as input by external means, or as created by a previous user (including the effects of compression, if any). In front of the logical end (where, presumably, no actual text is stored) there is a presumption that each line (page, the whole book) has some physical length which will be discovered only when an attempt is made to write beyond it. In the case of a newly "establish"ed book, the physical limits are defined to be those provided by the user (or the defaults for the channel in the case of "create"). Therefore the implementor should maintain 3 integers (mp, ml, mc) for each writeable book, and should provoke line, page and physical file end events accordingly. If now the book is closed and subsequently re-opened, should

the same values (mp, ml and mc) still apply? In a friendly system, yes; but this would involve storing these numbers in the directory or elsewhere (RSX 11 does make provision for storing a maximum line length in the directory, and this could be used to store mc). There is no obligation upon the implementor to do this in unfriendly systems, however, since the Gremlins can be presumed to have altered the limits (presumably to the system defaults) upon closing. In the case of a book that is "reset", there is no reason why the implementor should forget (mp, ml, mc), even if it becomes accepted that the Gremlins are to be uped in "reset".

Many operating systems put a limit on the amount of output on a given stream. Usually, this is expressed as a number of lines rather than as a number of pages. Since the default physical limits are specified by a proc pos field of the channel, and since there is no way for the user to discover the limits in force except by actually running into them, an implementor might consider provoking the physical file end event as soon as the line limit had been exceeded. Strictly, this would be a violation of the Report since, in general, this event would not occur at the exact end of a page, and "create" is written so as to imply that all pages are of the same length. There would be no violation in the case of a "create"d book that was supposed to consist of one long page. However, there is a further difficulty of implementation here. If the operating system imposes some such limit but is unwilling to inform the implementor's run-time system in advance what it is, then the run-time system may start filling the buffer for the offending line and not be aware of the offence until it comes to write it out, which is too late to provoke a physical file end event according to the Report. Nevertheless, the implementor has to do something specific here, and a physical file end event must be preferable to a program abort. Users should be warned that any characters written to the buffer up to that point will then have been lost (they can find how many by calling "char number"). A further possibility arises if the operating system cases an interrupt upon detecting the violation, but then allows transput to continue (perhaps the run-time system is allowed to specify a further limit). In this case, the output could then continue to the end of the current page (or at last for one more line), and then raise the physical file end event in good order.

Note that for an "establish"ed book, if the operating system limit is known at establish time it can be compared with p*l as given by the user, and "establish" can fail if necessary. Alternatively, if it is required that some limit be given to the operating system at this point, clearly p*l is the right limit to give.

On input, which always takes place before the logical end, the physical file end event can never be provoked. Line end and page end events (which are physical events) occur according to the historical state of the file as it was created, or written by previous users.

When a book, as previously written (and maybe compressed), is overwritten (its logical end is moved back), then the space in front of the new logical end ceases to be what had historically been written there, and the physical limits revert to some undefined values, either the (mp, ml, mc) in use beforehand (if the implementor has remembered them - which he ought to do where he can), or the default values for the channel (which is what the implementor ought to do otherwise).

## 5.7. Reading and Writing to the same book.

The Report nowhere defines that an implementor is obliged to provide channels on which "put" and "get" are both "possible" on a sequential book. However, the properties of such channels are well defined for implementors who decide to take the plunge.

On a sequential book, writing is only possible on the line which contains the LFE (more precisely, any attempt to write on a line before the LFE causes the LFE immediately to be moved back). Three cases are of interest:

A. The book contains information up to some LFE, and has perhaps been read part way. It is then "reset", and the next action is to write something. The effect is as if a new book had been "create"d, as discussed above under physical file limits.

B. The book has been read until the user has detected the logical file end event, which is presumed to be at the start of the first non-existent line. The user then starts writing. That non-existent line then becomes an existent line and output proceeds. Most operating systems should be able to manage this, even if the implementor is forced to close the book and to re-open it in some "append" mode.

C. The book has been read by the user up to some arbitrary point (or up to the LFE, but the LFE is not at the start of a line). He then attempts to write. Presumably the current line at this point is made up partly of what was read before, and partly of what he is now writing. This is well defined, but may be defficult for some operating systems which do not permit a line that has been read to be overwritten, or which do not permit overwriting except from the LFE.

A is easy in most operating systems. The question at issue is whether an implementor can permit A to happen without being required to permit B and C also, (or whether he can implement A and B without C). The answer is yes he can, because "put possible" is tested before each write operation (in principle at least), and "put possible" is a procedure which, in effect, consults both the channel and the book before delivering its verdict. It is therefore quite legal for "put possible" to return true if the book has just been reset (case A) or maybe if it is at the LFE (case B), and to return false otherwise. (Although implementors can provide "put possible" procedures with the most outrageous properties, reasonable implementors will attempt to give some appearance of consistency; thus a user who checks "put possible" for himself and finds it true can reasonably expect to be permitted to write at the current position.)

Another case where reading and writing to the same book may be of importance is in the case of interactive devices such as online terminals. There are two cases to consider. If the operating system only permits transput to online terminals in units of one complete line (NOS, GEORGE3) then there is no problem. The device is regarded as two books attached via two files (presumably "standin" and "standout") and what the user sees before him is a merging of the lines of the two books (a friendly system will print them in different colours of ink). Users must ensure that they call "newline" after any line which they actually expect to see printed before them and they will have to type CR at the end of each line they input before they can expect their program to see it.

The other case is where the operating system permits transput to the terminal a character at a time so that the user can, for example, output a prompting message and have the customer type his reply on the same line. It is clear that the Report makes no pretence of being able to support such a facility. The trouble is due to the existence of "backspace" (also misuse of "set char number") which would be meaningless in such a context. J.C. van Vliet has proposed an environment enquiry "backspace possible". This is both a sublanguage feature (in the full language, backspace is supposed to be available on all channels) and a superlanguage feature. Nevertheless, if its use is restricted to channels intended for interactive and other such specialized uses, I think it is a good idea. With this feature, one can now arrange that the online terminal is one book attached to one file (with both "put" and "get" possible but not "backspace"). The current position in this book is advanced whenever the user types characters in response to a "get", as well as when the program does a "put". The LFE is presumed not to get in

the way, and what the user sees printed before him is exactly the contents of the book as defined by the Report. Seveal query/response cycles may take place within the confines of one line. Note that, in the sublanguage ALGOL 68S, there is neither "backspace" nor "set char number".

## 5.8. Files and events.

There are no truly secret modes in ALGOL 68 - not even those with letter-alephs in their field-selectors. Hence the well known joke which I published in the first edition of the Informal Introduction (page 278 - also page 262 of the Revised Edition) in which I, quite legally from a pedantic point of view, assigned a structure-display consisting mostly of skips to a file variable. Having pointed out there that I did not expect implementors to take it seriously and subsequently having done nothing about the problem when rewriting the Report, readers should surely deduce that I did not consider the problem worth mending (actually, we did mend the corresponding case of the mode format and the cure is indeed worse than the disease, since the extra letter-aleph added for the purpose does nothing to improve the clarity of that corner of the Report). Therefore, implementors should feel quite free to add additional fields to the mode file for their own (hopefully reasonable) purposes.

In particular, therefore, they may add extra "mended" fields to their file and include corresponding extra "on" procedures in their library-preludes. (It was because of this possibility, not originally present, that we were able to remove the "other error" field from the file of the Old Report.)

Therefore, if situations can arise in the implementor's operating system (discs/directories becoming full, power supply problems, etc) which he feels users should be able to catch, but which do not fit within the existing events, then the correct action is for him to create extra "on" procedures accordingly. I think this treatment would also be correct for parity errors. The Old Report suggested that "on char error" was the correct event for parities, but we removed this suggestion from the Revised Report.

## 5.9. Possibles.

The various environment enquiries are all procedures with undefined bodies (except that a few special cases are defined, such as that "get possible" must always yield true on "stand in channel"). This of course allows the most outrageous things to be done, but a reasonable implementor will try to make them behave as consistently as his operating system will allow.

In an exceedingly friendly system, each environment enquiry will always return the same value, true or false, for a given channel. In practice, this can hardly be managed. Here are some examples of reasonable exceptions:

A. The properties of "stand in channel" and "stand out channel" may well depend upon the particular books and methods of attachment as specified by the JCL. Thus if the attached book is a physical device, or a spooling system of rigid specification, it is unlikely that other than a straightforward "put" or "get" will be possible. On some systems, "put" may be possible even via "stand in channel", but this may not be so if the user in question does not have write permission for the particular book. "reset" may well be possible on these channels, but "set possible" is unlikely. "compressible" may turn out to be false if the system knows that the output is destined for a card punch (otherwise, I expect that it will always be true for sequential books).

B. It may be that certain things will only be possible if the book is in certain states. For example, a change from reading to writing may only be allowed if the book is positioned at its beginning. The procedure "put possible" would then have to test for this case and, as soon as writing had commenced, it might be that thereafter "get possible" yielded false. It might also be that "reidf" was only possible if no transput had occurred since the book was opened.

Nevertheless, I would expect implementors to play fair. If a user explicitly enquired whether some action was possible and then immediately tried to perform that action, he could reasonably feel aggrieved if he found that the system had suddenly changed its mind about the possibility.

## 5.10. Reidf.

According to the Report, the procedure "reidf" can only be called on an opened book. We inherited this feature from the Old Report but, if I had to justify it post facto, I would say that before any operation can be performed on a book, it must first be established that the book exists and that the user is entitled to access it, and maybe even that the user has gained exclusive access to it. All these functions are already performed by "open" or by "establish", so why make additional provision for them to be performed elsewhere?

However, the problem is that not all operating systems are able to rename a book when it is open. With such operating systems, the implementor has the following options:

A. "reidf" closes the file, renames it, and then reopens it. This is likely to lose the current position, it could be that a different book is obtained on reopening, and there may be difficulty in writing further characters, or in calling "backspace", on what had been the current line. All in all, it is doubtful if this method could be implemented without violating the Report.

B. As above, but "reidf possible" (which is a procedure) only yields true if the file is positioned at its beginning (or maybe at its end). This is legal (provided there are no problems with reopening the same book - remember that "reidf" does not up any gremlins) but maybe not what the user expects.

C. As above, but "reidf" is only possible if no transput has yet occurred. This may be helpful if the implementor does not actually perform the operating system "open" function until the first transput is attempted (a practice he may find useful for other reasons).

D. "reidf" causes no immediate action, but the revised "idf" is remembered and the book is renamed when the file is eventually "close"d or "lock"ed (even here there may be problems if some other user has the book open at the same time, but that snag seems to exist with all the methods). I think this method is just about legal. A possible violation could only occur if the user tried to open the book again (assuming the operating system allowed two simultaneous accesses to the same book) before the closing. If he tried to open it under its old "idf" then he might succeed, even though the Report suggests that he should fail. In fact, the Report will call "undefined" in "open", but who is to say that for the system then to give him the book which it still knows by its old "idf" is not "sensible"? If he tried to open it under its new name he would fail, even though the Report suggests that he should succeed. However, the Report relies on the procedure "match" for this purpose, and the definition of that procedure is sufficiently flexible for it to be allowed to return false for any reason that might be considered reasonable in the context of the particular operating system.

E. Of course, there is no obligation upon the implementor to make "reidf possible" at all.

I recommend alternative D, or, failing that, alternative E.

Note that the facilities which might be provided in the "idf" parameter of "reidf" are likely to be much less generous than those suggested in 5.1 above for "open" and "establish".

5.11. Conversions.

The conv feature of the transput section of the Report is one of its less satisfactory features. Any feature provided by the hardware or system software for converting between character codes is likely to work on a line of information at a time, and then only after that line has passed out of the view of the ALGOL 68 run-time system. J.C. van Vliet has suggested a restriction whereby it would only be legal to change a conv when the current position of the file was at the beginning of a line. This is indeed legal, since the conversions which the implementor may store in his library-prelude are all of the mode proc(ref book)conv and, the bodies of these procedures not being defined in the Report, the implementor may elect to include in them tests on the current position. Unfortunately, this does not solve the problem. Here is how to output a string with alternate characters in ASCII and EBCDIC:

```
file a := standout, e := standout;
make conv (a, ascii); make conv (e, ebcdic);
string s := skip;
for i to upb s do put ( (odd(i) | a | e), s[i]) od
```

Therefore, if the implementor is to make use of the code conversion facilities of his hardware or system software, he can only do so by means of disposal instructions for the book written in JCL (or maybe in some "idf" parameter).

To implement the conv features of the Report, therefore, the implementor must write conversion procedures for himself. Note that the Report specifies that conversions may fail, that failures are to be checked character by character, and that the "char error" event is to be raised immediately a failure is discovered, in particular before any remaining characters of the string containing the offending char are processed. One can envisage four kinds of conv that implementors might provide:

1. There are no conversions to be done, and all characters (i.e. all "c" for repr(0) $\leq$ c $\leq$ repr(max abs char) are legal).
2. There are no conversions as such, but certain characters (perhaps the non-printing ones) are to fail.
3. There are conversions to be done, and all characters are convertible.
4. There are conversions to be done, but certain characters are unconvertible and are to fail.

In case 3, where there can be no failures, it might be possible for the implementor to postpone the actual conversion until the whole line was ready for output, but in cases 2 and 4 he has no option but to go round a loop once per char doing the necessary tests. This, however, would be intolerably inefficient in case 1 where it should be possible to move a complete string into the line buffer using the hardware MOVE instruction with which most machines are provided. Moreover, case 1 is likely to be far and away the most common situation in practice, and it should not therefore have to pay for the inefficiencies of the others. The implementor should therefore provide a flag to indicate whether the null conversion (case 1) is in operation and should only enter the character by character loop if this flag is not set (this technique is used in Richard Fisker's implementation model).

Of course the Report nowhere obliges an implementor to provide any convs at all in his library-prelude, and indeed, having regard to the

difficulties mentioned above, it might be preferable for him not to do so. If he does decide to provide some convs, he still has complete discretion as to which ones he provides. There are certain characters that the Report expects to be able to output in "put" or to input in "get". These are the DIGIT-symbols, "_", ".", "+", "-", "a" to "f", the times-ten-to-the-power-symbol and the plus-i-times-symbol (acceptable alternatives for these last two are "e" and "i"). The implementor can save himself a lot of checking if he assumes that conversion of these symbols will never fail, and I have no sympathy at all for any implementor who tries to provide convs which do not include them.

6. Random files.

Even though an operating system may provide some means of random access to text files via the line number, it is unlikely to provide the access via both page numbers and line numbers that is envisaged in the Report. Many operating systems provide no features in this area at all. I expect, therefore, that implementors will be forced to adopt private formats for random books. The least that they are likely to have to do when opening an ordinary book for random access is to construct an index of where all the pages and lines start. Please do not let me discourage any implementor whose operating system is friendly enough to implement random access books directly - it is just that I fear that the majority of operating systems are nowhere near friendly enough for this purpose.

A newly "establish"ed random access book will have its LFE at the beginning, and the only way to change this state of affairs is to write sequentially using "put". If the book is "compressible", this will lead to lines (and pages) of varying length, with the necessity of constructing an index so that subsequent "set" operations can find them again. If the book is not "compressible", the situation is much simpler, because all lines and pages are now the same size, and a simple computation will enable "set" to find any required position. Indeed, although the properties of such books are well defined by the Report, it was never our expectation that anyone would implement them (and certainly, there is nothing in the Report to oblige any implementor to do so).

There is, however, room for a superlanguage feature allowing the lengths of lines to be changed dynamically, for use with those operating systems (e.g. MTS) which support such books directly.

7. Binary transput.

Binary transput is a mess. It was a mess in the Old Report and, although discussions had taken place in the Transput and Data Processing Sub-committee with a view to inventing a "record transfer" system that would be worthy of the language, these never came to anything. Therefore, knowing that it was a mess, we left it strictly alone, in order to provide the greatest possible incentive for someone to come along later with a decent superlanguage feature to do the job properly.

The only really well-defined property of the binary transput system is that, if you output something, you can get back exactly what you output. It is not defined how much space is occupied by values of any particular mode, nor that all values of a given mode occupy the same space, nor even that a given value occupies the same space on different occasions. Unfortunately, it is defined that the transput of structures and multiples is equivalent to the transput of their straightened sequences of primitive values, and vice versa. This makes efficient implementation impossible. There would be much to be said for implementing the sublanguage in which this requirement is not

made.

If the book has random access, then the only way a user can find the location of his data, for subsequent use of "set", is for him to enquire about the "page number", "line number" and "char number" before he outputs it. The procedures "space", "newline" and "newpage" may on occasions be useful for aligning data in convenient positions, but the fact that they work at all for binary transput is probably more due to the fact that they exist, than that they are useful. This is probably why they are not available for use with "put bin", "get bin", etc.

There are very severe restrictions on the mixing of character and binary transput in the same book. These restrictions are probably more severe than they need to be.

## 8. Conclusions.

I hope I have been able to show that, apart from conversions and binary transput, the Report stands up remarkably well to the onslaughts of actual operating systems. The only thing that I really regret is that we did not up the Gremlins in "reset". Certainly, I think that the solutions to the various problems that I have proposed are as good as any others that might be proposed in the light of the evident unfriendliness of the operating systems (and I would add that most of these features would probably be regarded as just as unfriendly by the implementors of other languages).

But I seem to hear murmurings in the background that, in hypothesising all these actions of the Gremlins, and of the various other procedures whose bodies are not defined in the Report, I am merely being wise after the event. Not so! We knew at the time that operating systems were going to be unfriendly. We did not know which particular unfriendly features were actually going to rear their ugly heads when implementors actually got down to business, but we knew that we did not know this. Therefore, we had to indulge in a huge overkill. The Gremlins and their allies can indeed do the most outrageous things and make utter nonsense of any program that a user might try to run. But, although we know that operating systems are unfriendly we presume that the implementors at least are on our side and will curb the worst excesses of the enemy, providing us with systems that are as comfortable as the particular operating system will allow.

## 9. Acknowledgments.

An earlier draft of this paper was discussed by the Transput Task-Force of the ALGOL 68 Support Sub-committee at the Oxford meeting of the Working Group, and I am grateful for the many helpful suggestions made by the Task-Force members.

## AB42.4.5 Proposals for Algol H-a superlanguage of Algol 68

A.P. Black, V.J. Rayward-Smith
School of Computing Studies, University of East Anglia

### §1. Introduction

This paper is devoted to the description of a proposed extension of Algol 68 which we shall call Algol H. The motivation for the development of this language comes from [1]. Sections 3 to 7 of this paper correspond to the sections in [1] with the same names. They describe constructions in Algol H which represent the abstractions of Hoare's theory.

Professor Hoare makes a vigorous disclaimer in the introduction to his paper: he is not embarking on the design of yet another programming language [1]. His abstract data structures are intended to assist in the formulation of abstract programs, and in their representation as concrete code. The transition from abstract to concrete is an essential part of the program design process, and there are good reasons why it should not be automated. Hoare draws a clear distinction between an algorithmic language and a programming language, his notations being an example of the former and Algol an example of the latter. However, Hoare admits there are advantages in the programming language being a subset of the algorithmic language. Many of his notations can be implemented with high efficiency; this is certainly true of unstructured data types and of elementary data structures, viz., Cartesian products, unions, arrays and powersets.

The advanced data structures, namely sequences, recursive structures and sparse structures, are fundamentally more difficult to implement by an automatic translator. Consequently, no proposals are made here to include these structures in Algol H (except in as far as transput is a representation of certain kinds of sequence).

Many of the new constructions proposed for Algol H have counterparts in the programming language Pascal [4,8]. Enumerations are available (always ordered), and so are subranges, although these are more restricted than the submodes of Algol H in that all subranges of a given parent type must be disjoint. The syntax of Pascal is markedly different from that of Algol H, which is intended to introduce these concepts to programmers more familiar with Algol 68. Some of the constructions of Algol H have previously been described as Algol 68 "might-have-beens" [7].

### §2. The Method of Definition

As an Algol 68 superlanguage, Algol H should be described by a (two-level, Van Wijngaarden or) W-grammar, as is used in the Report. (Here, and in all that follows, "Report" means the Revised Report on Algol 68 [2]. References to specific sections are given as, e.g. [R 2.2.2.c].) However,

this has certain disadvantages, for W-grammars have been held to be difficult to understand, particularly by those who do not know the language they describe. Whilst we feel this difficulty is often overestimated, it is none the less real. For this reason, the constructs of Algol H are described here only by means of examples and natural language. It is hoped that this will be "easier for the uninitiated reader", but for the initiated, part of the W-grammar definition of Algol H can be found in [3]. It should be noted that it is not easy to extend the grammar of Algol 68 so that it defines Algol H. For example, in Algol 68 the metanotion NEST, which carries a record of all the declarations forming the environment, has no need to envelop denotations, since the meaning of a denotation is independent of any nest [R.8.0.1]. In Algol H this is not so; it is possible to declare enumeration modes whose denotations are scoped.

## §3. Unstructured Data Types

All data structures in a program must be built up from unstructured components. Most programming languages provide some unstructured types, usually reals and integer, and in theory these are adequate for all purposes. In practice there are strong reasons for defining other unstructured types. For example, although character values can be represented as integers, it has become usual to provide a character type in text processing programming languages. This has two advantages. First, the potential range of values of a variable is made explicit, thus making the program clearer and subject to more compile-time checks, which can detect such errors as the addition of an integer to a character. Second, it is possible to devise an efficient representation; because the cardinality of the character set is usually much less than that of the set of permitted integers, characters can be represented in less bits in the computer memory.

The next step, after including in a language a wider choice of basic modes, is to allow the programmer to define his own unstructured modes, either by enumeration of values or by taking a subrange of some existing mode. However this is done, it is fundamental that different data types should be represented as different modes. Since this mode is known at compile time it is possible to ensure that unrelated data are not mixed. The protection thus provided is one of the principal benefits of the extensions.

### 3.1 Enumerations

In many programs integers are used to denote a choice from a small number of alternatives rather than numeric quantities. In such cases we may expect

**the documentation of the program to list the possible values with their intended interpretation. For example, one might find the following in a program concerned with bidding sequences in bridge.**

```
int trumps, bestsuit;
c  These variables refer to integers which
   indicate suit values as follows:

   0 - clubs
   1 - diamonds
   2 - hearts
   3 - spades
   4 - no trumps
c
```

**(i)**

**The notion of an enumeration enables quantities such as suits of cards, sexes or colours to be represented as separate modes, quite distinct not only from the integers but also from each other. In Algol H the following are legal mode declarations.**

```
mode suit = order (clubs, diamonds, hearts, spades, notrumps)
mode sex = scalar (male, female)
mode primary colour = scalar (red, green, blue)
mode dayofweek = order (Sunday, Monday, Tuesday, Wednesday,
                        Thursday, Friday, Saturday)
```

**The use of the** order-token (*order*) **indicates that the mode should be considered to be ordered; the** scalar-token (*scalar*) **denies any such ordering. The** bold-TAG-symbols (*clubs*, *male*, **etc.) are** MODE denotations **for the various** MODES, **in exactly the way that, in Algol 68,** *true* **and** *false* **are** boolean denotations. **To be pedantic, the parallel is not exact, for** *true* **is the representation of the** true-symbol **not the** bold-letter-t-letter-r-letter-u-letter-e-symbol. **Nevertheless, it is clear [R.4.2.2b] that the similarity is intentional. This distinction does not arise since in Algol H** *bool* **is not a primitive mode but is defined using** `mode bool = order (true, false)`. **The use of the new denotations throughout the program enhances its understandability considerably; the assignation**

```
trumps := spades
```

**conveys more information (to the human reader) than**

```
trumps := 3
```

**(ii)**

**which would be its counterpart on the assumption of trumps being an** *int* **variable, as in example (i).**

The situation can be alleviated in Algol 68 by use of identity declarations for appropriate integers:

*int clubs = 0, diamonds = 1, hearts = 2, spades = 3, notrumps = 4*

which may then be followed by

*trumps := spades*

in the same context as (ii).

This use of ascription is no substitute, however, for the ability to define new basic modes: the advantages of enumeration modes become clearer when control structures are introduced to manipulate them, but immediately we see that there can be better protection, more efficient store utilisation, and a closing of the gap between the data structures of a program and the real world objects they represent.

As far as protection is concerned, not only is the programmer less likely to assign objects of one type to variables of another (because of the mnemonic names), but such assignations are in any case syntax errors, since each type of object is represented by a distinct mode in the program.

The standard transput routines may, of course, be applied to enumeration modes - a facility not available in Pascal. The external forms of the values of such modes will in general be implementation dependent. It is suggested that where the bold-TAG-symbols used in the program text for denotations are represented by stropping, the characters transput should be those forming the corresponding TAG-symbol.

In Algol 68, declarers specify modes. A declarer is either a declarator, which explicitly constructs a mode, or an applied-mode-indication, which stands for some declarator by way of a mode-declaration [R.4.6]. Thus, to introduce new basic types, it is necessary to provide new declarators. Syntactically, this is done by providing a new descendent of the notion VIRACT-MOID-NEST-declarator [R.4.6.1a].

Algol 68 is quite specific about the equivalencing of modes. For example, the modes specified by the mode indications *a* and *b* in

*mode a = union (int, real);*
*mode b = union (real, int)*

are equivalent.

Similarly, it is intended that the modes specified by the declarators *scalar (red, blue, green)* and *scalar (red, green, blue)* be equivalent. On the other hand, the declarators

*order (morning, afternoon, evening)*
*order (afternoon, evening, morning)*
and *order (morning, noon, night)*

all specify different modes (which cannot co-exist in the same reach, since applications of the denotations cannot be uniquely identified). In [3], the metanotion MODE is extended to include the metanotion BASIC as an additional alternative [R.1.2.1.A]. BASIC envelops all possible enumeration modes as its descendants. The mode equivalencing syntax then needs extending so that it can compare permuted scalar modes, for example, to detect the equivalence of *scalar (male, female)* and *scalar (female, male)*.

### 3.2 Subranges

Another common requirement is to deal with quantities which, though intrinsically of a basic type, will take only a limited range of values - a subrange. Clearly the parent type must be ordered. The bounds of the submode must be denotations, optionally preceded by a sign if the parent mode is integral.

```
mode dayofmonth = sub (1:31);
mode letter = sub ("a":"z")
```

The parent mode can be an enumeration mode; for example, in the reach of

```
mode dayofweek = order (Sunday, Monday, Tuesday, Wednesday,
                        Thursday, Friday, Saturday)
```

the following is a valid mode declaration.

```
mode workingday = sub (Monday:Friday)
```

A subrange may also be defined as the union, intersection or difference of a pair of subranges, provided they both have the same parent mode. These operations are represented by ∪, ∩ and \ with their usual set theoretic meanings.

It is not possible to declare a submode of a submode since the submode has no denotations. For example, *sub ("i":"n")* specifies a sub-character mode, not a sub-letter mode, because *"i"* and *"n"* are character denotations.

In Algol H, there is a widening coercion from a submode to a mode which is available in strong positions.

### 3.3 Manipulation

Hoare lists seven operations required for the manipulation of values of enumeration and subrange modes.

Test of equality: The equality and inequality operators are defined for all enumeration modes, and all subrange modes which are not submodes of real.

**Assignation**: The assignation of values to names is exactly as in Algol 68.

**Case discrimination**: Algol H has a choice-clause which is a generalisation of the choice-clause of Algol 68. The advantages of enumeration modes become obvious when we compare equivalent Algol 68 and Algol H. The Algol 68

```
case trumps +1
in   20 × bid, 20 × bid, 30 × bid, 30 × bid
,    40 + 30 × (bid-1)
esac
```

is almost incomprehensible without the comment of Section 3.1, and is error prone even with it. In Algol H we can similarly write (because *suit* is ordered)

```
case trumps
in   20 × bid, 20 × bid, 30 × bid, 30 × bid
,    40 + 30 × (bid-1)
esac
```

but without having to perform arithmetic on *trumps*. The $v^{th}$ constituent units of the in-choice-clause is elaborated when trumps takes the $v^{th}$ value of the ordered enumeration. If there are less units than values in the mode, then the out-choice-part is elaborated for the extra values.

Alternatively, the constituent units can be prefixed with specifications, as in the following.

```
case trumps
in   (clubs, diamonds): 20 × bid
,    (notrumps): 40 + 30 × (bid-1)
out  30 × bid
esac
```

Each value of the mode may be mentioned in at most one specification; if the enquiry clause takes a value not mentioned in any specification, then the out-choice-part is chosen. (If there is no out-part, a skip is elaborated, which may yield an undefined value.) If the MODE of the enquiry clause is ordered then sub-of-MODE-declarators are permitted as specificators; for example, given the declaration *mode letter = sub ("a":"z")* we write a skeleton scanner.

```
case ch := readch
in   (letter) :    serial clause
,    (sub("0":"9"),".") :    serial clause
,    ("""") :    serial clause
,    ("$") :    serial clause
,    ("'") :    serial clause
out    serial clause
esac
```

**If, on the other hand, the MODE of the enquiry clause is not ordered, then the form of the choice clause which does not use specifications is not permitted, since there is no implicit ordering of the values of the enumeration which can be used to choose the appropriate unit from the in-choice part.**

**Ordering relations: The dyadic operators <, <=, >, >= are declared in the standard prelude between operands of the same mode and yielding boolean results. They are defined for a sufficient set of ordered modes, each member of which** has order.

**Counting: The monadic operators** *succ* **and** *pred*, **invoking the successor and predecessor functions, are defined for all simple, ordered modes. These functions map each value of a mode onto the next higher or lower value (if there is one, otherwise their action is undefined).**

**Also, for each ordered mode for which there is a mode declaration in a given reach, if the mode indication defined by that declaration is some** bold-TAG-symbol, **then the identifiers** max-cum-TAG-symbol **and** min-cum-TAG-symbol **are declared and ascribed the maximum and minimum values of the mode. The example should make this clear.**

```
mode rank = order (private, corporal, sergeant,
     lieutenant, captain, major, colonel, general);
c    The effect on the nest is as if, in place of
     this comment, there stood the declaration
          rank maxrank = general, minrank = private
c
...
```

**The loop clause: It is frequently required to elaborate the same serial clause for all possible values of a given mode. This is indicated by the use of a loop-clause.**

```
loop dayofweek day
do
     dailytask (day)
od
```

The loop-symbol is followed by a formal-declarer which gives the mode of the identifier whose value is varied. If the mode is ordered and has cardinality n, say, then the serial-clause between *do* and *od* is elaborated n times in sequence as the identifier takes, in order, the values of the mode from minmode to maxmode. However, if the mode is not ordered, the effect is as if n copies of the serial clause were elaborated collaterally. The loop clause is available for all modes derived from SIMPLE, not just for integral as in Algol 68. However, the special *for from by to* construction of Algol 68 is retained for use by integers, since conceptually that mode has infinite cardinality.

Transfer functions: It is sometimes required to perform operations for a submode which were defined for the parent mode. This is simply accomplished by converting the submode value to the corresponding value of the larger type, then performing the operation, and finally converting back again if necessary. This requires transfer functions.

The first conversion is a widening, and can be accomplished by the coercion described in Section 3.2. If the context is not strong enough (as will be the case if the coercee is the operand of a formula) the coercion can be forced by the use of a cast.

```
mode smallint = sub (-9:9);
smallint si ,     sj ,     sk; int k
...
k := int (si) + int (sj)
```

It is, of course, possible to declare a version of + between small int operands, but this must be done explicitly by the programmer.

The second conversion represents a narrowing, and can be accomplished only with the aid of a standard operator.

```
sk := smallintval k;
```

This operator is automatically declared in any reach in which a mode declaration is given for a submode. The letters of the TAG used for the operator are given by concatenating the letters of the bold-TAG-symbol defined in the mode declaration with letter-v-letter-a-letter-l.

If this operator is applied to an operand whose value is outside the

range encompassed by the submode, then the result is undefined. It is therefore useful to be able to check if a given value is in a subrange. This can be done by a conformity relation, which is based on the conformity relation for unions which was part of the language defined in 1968 [6]. The symbols represented by *::=* and *ctab* denote a conform-to-and-becomes relation, whereby the object on the right hand side is assigned to the variable on the left (and true delivered) if it is in range, otherwise no assignation takes place (and false is delivered). The symbols represented by *::* and *ct* invoke the same test but without the assignation. The use of these relations is illustrated in the following examples.

```
char c; letter a; digit b;
read (c);
if a ctab c then c an assignation to a has just occurred c
                    serial clause
elif b ctab c then c an assignation to b has just occurred c
                      serial clause
else c neither a nor b have changed c
     serial clause
fi
```

(i)

```
if letter ct c then c c is in the subrange letter c
                      serial clause
fi
```

(ii)

In the second example, the conformity relation has the property that its left hand side is not elaborated, i.e. no space is generated on the heap. The generator *letter* is simply there for mode matching. Note that *ct* can be used even when no mode declaration has appeared for a submode, and consequently it is not identified by a bold-TAG-symbol, so no submode-cum-val operator is defined.

Very similar results could instead be achieved by standard operators, also identified by *ct* and *ctab*. Such operators would be defined between a sufficient set of modes as right operand, and a sufficient set of submodes of each mode as left operand. Such operators would have the disadvantage that both operands would always be evaluated; in the second example above, space for a *letter* value would be generated on the heap, and immediately become garbage.

## §4. The Cartesian Product

The Cartesian product is one of the simplest data structures. It corresponds to the record structures of PL/I [9] and Pascal or the Algol 68

structured modes. In fact, with one exception, Algol 68 structures provide all the facilities suggested by Hoare [1], although the notation is different. In particular, values of a structured mode are constructed, in Algol 68, from a collateral; where the required mode is not obvious from the context, a cast of the collateral is used. This corresponds to Hoare's suggestion that it would be convenient to leave the transfer function implicit in cases where no ambiguity would arise.

The exception is the *with* construction. In inspecting or processing a structured value, it is often required to make many references to its components within a single clause. Hoare favours a special construction which could be represented as

*with structure closed clause* .

Within the closed clause the fields of the structure may be referred to by their field selectors alone, instead of by the normal selector *of* primary construction. It is debatable whether the advantages of this construction, viz., the clarification and abbreviation of a section of program, are great enough to outweigh the disadvantage of introducing another construct into the language. Algol 68 is often called a complex language, and ascription provides most of the power of the with clause. For these reasons, it is proposed that Algol H does not include this construction.

## §5. The Discriminated Union

Although similar to Algol 68 union, Hoare's Discriminated Unions differ in certain respects. Each type in the union has an identifier associated with it, and every value of one of the unioned types is marked with its identifier, to indicate its derivation. Thus it is possible to construct a union by repeating the same type several times.

In Algol 68, in contrast, the declarator,

*union (date, date)*

is ill-formed. The intention that Algol H should be a superlanguage of Algol 68 dictates that the Algol 68 kind of union be included in Algol H.

Since the advantages of discriminated unions are debatable, and it would in any case be confusing for one language to provide two very similar but distinct structures, discriminated unions are not included in Algol H.

## §6. The Array

The array is a very familiar data structure, and may be regarded as a mapping from a domain of one type to a range of a possibly different type

(the type of the array elements).

In most programming languages, the most notable exception being Pascal, the domain type is restricted to be integral. This is true of Algol 68. Algol H allows more general arrays; the mode of the domain can be any enumeration mode, SIZETY integral, character or a submode of any permitted mode.

Some examples should make this clearer.

```
[suit] int trickvalue;
[day of week] bool holiday;
[sub (1:80)] char punchcard
```

Elements of these arrays can be selected in the usual way.

```
day of week today := sunday;
if not holiday [today] then work else sleep fi
```

The facility of using a subrange of integers as the domain mode does not replace the ordinary Algol 68 row with integer indices, because the limits of a submode can only be plus or minus a denotation, and it is therefore not possible to read in submode bounds at run-time.

## §7. The Powerset

The powerset of a set is the set of all subsets of that set. The powerset as a data type takes values which are sets of values selected from some other data type known as the base. *Primarycolour* has been declared as an enumeration mode with cardinality 3; the powerset of primarycolour is a mode which has a cardinality of $2^3 = 8$. Its values are as follows.

```
{ }          {red, green, blue}
{red}        {red, green}
{green}      {red, blue}
{blue}       {green, blue}
```

Declarators for powerset modes take the form illustrated by

```
setof primarycolour
```

so we may construct the mode declaration

```
mode colour = setof primarycolour
```

and then declare some identifiers

```
colour yellow = {red, green},
       cyan   = {blue, green},
       blue   = {blue},
       black  = { }
```

The object between and including the braces is a powerset clause. The values within it must all be of the same mode, but their order is immaterial and a repeated value is ignored, so that {*green*, *red*, *red*} represents the same value as {*red*, *green*} which would not be the case for a collateral clause. The empty powerset clause { } can only stand in a strong position. An alternative representation of the braces are the symbols *set* and *tes*.

## 7.1 Manipulation

For each powerset for which a mode declaration is given, a constant is declared corresponding to the universal set, where this has finite cardinality. In the example, the effect is as if the declaration

*colour allcolour* = {*red*, *blue*, *green*}

were elaborated after the mode declaration.

Where the cardinality of the base type is small the powerset can be efficiently represented by allocating one bit in store for each value of the base type. Thus, for example, values of type colour can be represented in three bits. The basic operations on powersets are usually available as single machine instructions, which makes this representation doubly attractive. However, when it is known that the cardinality of the base type is large, or perhaps conceptually infinite, the bit pattern representation loses its attraction, particularly when most values of the powerset will consist of only a small number of elements of the base. In these cases it will be necessary to represent the powerset as some advanced data structure, which an automatic translator cannot be expected to construct.

However, the utility and efficiency of the powerset of a small base type is such that it ought to be included in Algol H; powersets are therefore permitted providing the cardinality of the base type does not exceed some implementation defined maximum possessed by the standard prelude integer *setwidth*.

Various operations can be defined between sets of a given powerset mode and elements of its base mode. There is one operator defined between a set $s$ and an element $x$.

Membership : $x$ *isin* $s$ delivers *true* if the element $x$ is a member of set $s$

The following operators, with their usual mathematical meaning, are defined between two sets:

equality,
union,
intersection,
relative complement,
inclusion.

Algol H also allows assigning versions of union, intersection and relative complement.

| | | | |
|---|---|---|---|
| Union and becomes | : *s1* ∪:= *s2* | equivalent to | *s1* := *s1* ∪ *s2* |
| Intersection and becomes | : *s1* ∩:= *s2* | equivalent to | *s1* := *s1* ∩ *s2* |
| Difference and becomes | : *s1* \:= *s2* | equivalent to | *s1* := *s1* \ *s2* |

It is also useful to be able to select an element from a set, and simultaneously remove it. This is achieved by the operator *outof*: *x outof s* removes an element from *s* and assigns its value to *x*.

## §8. Implementing Algol H

A major part of the work of any parser for Algol 68 is to perform mode-checking. Because, in general, a given mode can be "spelled" in a large (sometimes infinite) number of ways, the syntax of Algol 68 includes complex devices which check if modes are equivalent and if a value of a given mode is "acceptable" to another mode [R.2.1.3.6, 2.1.4.1]. The latter test depends on the syntactic position (SORT) of the construction, as the list of applicable coercions vary with context. All this must be modelled within a parser, which is no small task.

However, that this task can be successfully completed is evidenced by the existence of several usable and efficient Algol 68 compilers. Given access to such a compiler and a description of its mode representation mechanism, one feels that it would not be too difficult to extend it to encompass new basic modes.

This feeling is reinforced by the comments of Wirth on the programming languages Pascal and Modula. Pascal has scalar types which correspond to the ordered enumeration modes of Algol H, and subrange types which provide some of the facilities of submodes. Arrays with general domains and powersets are also included. Nevertheless, Wirth states that one of his principal aims in developing Pascal was to produce a language which could be implemented reliably and efficiently, both at compiler and execution time. In [10] he writes "a most important consideration in the design of Modula was its efficient implementability". Modula is a language based on Pascal but with an improved syntax and the ability to associate access procedures with data in the manner of a Simula [11] class; it includes enumerations, which may form the indices of arrays, and is intended for use in real-time applications on mini-computers.

Thus there is justification for being confident that most of the new constructions of Algol H could be efficiently implemented by an extensible Algol 68 compiler. To the best of our knowledge no such compiler exists; we have in mind an extension device akin to the Syntax Macros of Cheatham [12] and Leavenworth [13]. In [3], a translator from Algol H to Algol 68-R [14]

is described where each Algol H construct is translated into Algol 68-R in such a way that all mode checking is done by the Algol 68-R compiler.

Acknowledgement

The authors wish to thank Professor R.J.W. Housden for helpful discussions. The majority of this work was done by A.P. Black under the supervision of Dr. Rayward-Smith in partial fulfillment of the requirements of the B.Sc. degree at the University of East Anglia, Norwich.

References

[1] C.A.R. Hoare, "Notes on Data Structuring" in "Structured Programming" by O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press (1972).

[2] A. van Wijngaarden, et al., "Revised Report on the Algorithmic Language Algol 68", Acta Informatica, 5 (1975).

[3] A.P. Black, "Algol H: Some Extensions to the Data Handling Facilities of Algol 68", Project Report, School of Computing Studies, University of East Anglia, Norwich (1977).

[4] N. Wirth, "The Programming Language Pascal", Acta Informatica, 1 (1971).

[5] C.H. Lindsey and S.G. van der Meulen, "Informal Introduction to Algol 68", revised edition, North Holland (1977).

[6] A. van Wijngaarden, et al., "Report on the Algorithmic Language Algol 68", Numerische Mathematik, 14 (1969).

[7] S.G. van der Meulen, "Algol 68 Might-Have-Beens", Proceedings of the Strathclyde Algol 68 Conference, Sigplan Notices, 12:6 (1977).

[8] K. Jensen and N. Wirth, "PASCAL: User Manual and Report", Lecture Notes in Computer Science, 18, Springer Verlag (1975).

[9] C.P. Lecht, "The Programmers' PL/1", McGraw Hill (1968).

[10] N. Wirth, "Modula: A language for Modular Multiprogramming", Software-Practice and Experience, 7:1 (1977).

[11] G.M. Birtwistle, O.J. Dahl, B. Myhrhang, K. Nygaard, "Simula Begin" Auerback (1973).

[12] T.E. Cheatham Jnr., 'The Introduction of Definitional Facilities into Higher Level Programming Languages", Proc AFIPS FJCC, 29 (1966).

[13] B.M. Leavenworth, "Syntax Macros and Extended Translation", CACM, 9:11 (1966).

[14] P.M. Woodward, S.G. Bond, "Algol 68-R Users Guide", H.M.S.O. (1974).

AB42.4.6 The Most Contrived Factorial Program

John P. Baker
University of Bristol

```
BEGIN
PROC readint = INT: (INT i; read(i); i);
INT one=1,two=2,three=3,four=4,five=5,six=6,seven=7,
    eight=8,nine=9,ten=10,eleven=11,twelve=12;
INT a=one;
PRIO#rity# ME=5, LOVE=7, MY=7, LORDS=7, LADIES=7,
    PIPERS=7, DRUMMERS=7, MAIDS=7, SWANS=7, GEESE=7,
    GOLD=7, COLLY=7, FRENCH=7, TURTLE=7, PARTRIDGE=6;
BOOL sent to=TRUE;
OP  THE = (BOOL a)BOOL:a,
    TWELFTH = (INT a)BOOL:a=twelve,
    ELEVENTH = (INT a)BOOL:a=eleven,
    TENTH = (INT a)BOOL:a=ten,
    NINTH = (INT a)BOOL:a=nine,
    EIGHTH = (INT a)BOOL:a=eight,
    SEVENTH = (INT a)BOOL:a=seven,
    SIXTH = (INT a)BOOL:a=six,
    FIFTH = (INT a)BOOL:a=five,
    FOURTH = (INT a)BOOL:a=four,
    THIRD = (INT a)BOOL:a=three,
    SECOND = (INT a)BOOL:a=two,
    FIRST = (INT a)BOOL:a=one,
OP  ME = (BOOL a,INT b)VOID:(a|print(b)),
OP  LOVE = (BOOL a,b)BOOL:(a|b|FALSE),
    MY = (BOOL a,b)BOOL:a  LOVE b,
OP  AND = (INT a)INT:a;
MODE DATE = STRUCT (INT day,month);
LOC DATE christmas:=(25,12);
OP  LORDS = (INT a,b)INT:a*b,
    LADIES = (INT a,b)INT:a*b,
    PIPERS = (INT a,b)INT:a*b,
    DRUMMERS = (INT a,b)INT:a*b,
    MAIDS = (INT a,b)INT:a*b,
    SWANS = (INT a,b)INT:a*b,
    GEESE = (INT a,b)INT:a*b,
    GOLD = (INT a,b)INT:a*b,
    COLLY = (INT a,b)INT:a*b,
    FRENCH = (INT a,b)INT:a*b,
    TURTLE = (INT a,b)INT:a*b,
OP  LEAPING = (INT a)INT:a,
    DANCING = (INT a)INT:a,
    PIPING = (INT a)INT:a,
    DRUMMING = (INT a)INT:a,
    MILKING = (INT a)INT:a,
    SWIMMING = (INT a)INT:a,
    LAYING = (INT a)INT:a,
    RINGS = (INT a)INT:a,
    BIRDS = (INT a)INT:a,
    HENS = (INT a)INT:a,
    DOVES = (INT a)INT:a,
OP  PARTRIDGE = (INT a,b)INT:a+b;
INT in a pear tree = 0;
print("FACTORIAL OF ");
print(day OF christmas := readint#reads an integer#);
print(" IS ");
(day OF christmas>12|print("TOO BIG FOR THIS PROGRAM");stop);
```

# now we are ready #

THE FIRST day OF christmas MY TRUE LOVE sent to ME
a PARTRIDGE in a pear tree;

THE SECOND day OF christmas MY TRUE LOVE sent to ME
two TURTLE DOVES AND
a PARTRIDGE in a pear tree;

THE THIRD day OF christmas MY TRUE LOVE sent to ME
three FRENCH HENS
two TURTLE DOVES AND
a PARTRIDGE in a pear tree;

THE FOURTH day OF christmas MY TRUE LOVE sent to ME
four COLLY BIRDS
three FRENCH HENS
two TURTLE DOVES AND
a PARTRIDGE in a pear tree;

THE FIFTH day OF christmas MY TRUE LOVE sent to ME
five GOLD RINGS
four COLLY BIRDS
three FRENCH HENS
two TURTLE DOVES AND
a PARTRIDGE in a pear tree;

THE SIXTH day OF christmas MY TRUE LOVE sent to ME
six GEESE LAYING
five GOLD RINGS
four COLLY BIRDS
three FRENCH HENS
two TURTLE DOVES AND
a PARTRIDGE in a pear tree;

THE SEVENTH day OF christmas MY TRUE LOVE sent to ME
seven SWANS SWIMMING
six GEESE LAYING
five GOLD RINGS
four COLLY BIRDS
three FRENCH HENS
two TURTLE DOVES AND
a PARTRIDGE in a pear tree;

THE EIGHTH day OF christmas MY TRUE LOVE sent to ME
eight MAIDS MILKING
seven SWANS SWIMMING
six GEESE LAYING
five GOLD RINGS
four COLLY BIRDS
three FRENCH HENS
two TURTLE DOVES AND
a PARTRIDGE in a pear tree;

THE NINTH day OF christmas MY TRUE LOVE sent to ME
nine DRUMMERS DRUMMING
eight MAIDS MILKING
seven SWANS SWIMMING
six GEESE LAYING
five GOLD RINGS
four COLLY BIRDS
three FRENCH HENS
two TURTLE DOVES AND
a PARTRIDGE in a pear tree;

THE TENTH day OF christmas MY TRUE LOVE sent to ME
ten PIPERS PIPING
nine DRUMMERS DRUMMING
eight MAIDS MILKING
seven SWANS SWIMMING
six GEESE LAYING
five GOLD RINGS
four COLLY BIRDS
three FRENCH HENS
two TURTLE DOVES AND
a PARTRIDGE in a pear tree;

THE ELEVENTH day OF christmas MY TRUE LOVE sent to ME
eleven LADIES DANCING
ten PIPERS PIPING
nine DRUMMERS DRUMMING
eight MAIDS MILKING
seven SWANS SWIMMING
six GEESE LAYING
five GOLD RINGS
four COLLY BIRDS
three FRENCH HENS
two TURTLE DOVES AND
a PARTRIDGE in a pear tree;

THE TWELFTH day OF christmas MY TRUE LOVE sent to ME
twelve LORDS LEAPING
eleven LADIES DANCING
ten PIPERS PIPING
nine DRUMMERS DRUMMING
eight MAIDS MILKING
seven SWANS SWIMMING
six GEESE LAYING
five GOLD RINGS
four COLLY BIRDS
three FRENCH HENS
two TURTLE DOVES AND
a PARTRIDGE in a pear tree

END

AB42.4.7 GRAMMAR ANALYSIS WITH ALGOL68

Steven Pemberton
Brighton Polytechnic, U.K.

There are many reasons why a compiler-writer needs to analyse a grammar: for instance, to check that it is LL(1); to generate tables for a parsing algorithm; or to re-arrange the grammar to make it suitable for a particular parsing method.

The initial aim when analysing a grammar by computer is to form an internal version of the grammar in the machine, ie. some sort of linked list in main memory, so that it can be subsequently examined.

Up to now, the technique for forming such an internal grammar has been to write a translator to read in a description of the grammar from some medium, and produce the required internal form ready for analysis.

This paper describes an Algol68 prelude for a number of modes and operators that once declared, allows the user to write grammars in a natural manner directly in Algol68 without the need for any initial input stage. It has the advantages of being fast, easily modifiable for particular needs, such as inserting actions in the grammar, and allowing full use of the language facilities provided in Algol68.

## The Internal Grammar

The required structure of the internal form may be demonstrated by the following declarations:

```
MODE RULE=STRUCT(REF NOTION notion, DEFINITION def,
                 REF RULE next),
     DEFINITION=STRUCT(ALTERNATIVE alt, REF DEFINITION next),
     ALTERNATIVE=STRUCT(UNION(NOTION,SYMBOL) member,
                        REF ALTERNATIVE next);
```

The definitions of NOTION and SYMBOL would largely depend on particular requirements, but presumably the mode of NOTION would include a REF DEFINITION.

## Grammar Specification

The basis of the method described here is a number of operators, chosen to look as much as possible like the characters used in conventional syntaxes, which, thanks to priority declarations, allow the straightforward transcription of grammars.

The first of these operators is asterisk, chosen for its (slight) resemblance to a point, and is used to separate rules (and not to terminate them, since it is an operator):

```
grammar: rule; grammar, asterisk, rule.        (1)
```

Since it is at the outermost level, and hence has the lowest binding, asterisk will have the lowest priority.

A rule consists of a notion and its definition. The obvious candidate for the required operator is equals:

rule: notion, equals, definition. (2)

It will have a priority higher than that of asterisk.

A definition consists of a number of alternatives separated by some suitable operator. An upward arrow is suitable here due to its similarity to the vertical bar in BNF:

definition: alternative; definition, up, alternative. (3)

(Alternatively, an exclamation mark could be used, but some implementations may use it as a representation of the stick symbol). Up will have a priority greater than that of equals.

Finally, an alternative is a number of members, separated by plus symbols, used for their suggestion of concatenation:

alternative: empty; member; alternative, plus, member.
member: notion; symbol. (4)

Plus will have the highest priority of the four operators.

These definitions can be used directly to declare the required modes and operators:

```
PRIO *=1, ==2, ↑=3, +=4;

# from (1) #
MODE GRAMMAR=UNION(RULE, RULES);
MODE RULES=STRUCT(REF GRAMMAR gram, RULE rule);

OP *=(GRAMMAR g,RULE r)GRAMMAR:RULES(HEAP GRAMMAR:=g,r);

# from (2) #
MODE RULE=STRUCT(REF NOTION notion, DEFINITION def);

OP ==(REF NOTION n, DEFINITION d)RULE:(def OF n:=d; (n,d));

# from (3) #
MODE DEFINITION=UNION(ALTERNATIVE, ALTERNATIVES);
MODE ALTERNATIVES=STRUCT(REF DEFINITION def, ALTERNATIVE alt);

OP ↑=(DEFINITION d, ALTERNATIVE a)DEFINITION:
                    ALTERNATIVES(HEAP DEFINITION:=d,a);

# from (4) #
MODE ALTERNATIVE=UNION(VOID, MEMBER, MEMBERS);
MODE MEMBERS=STRUCT(REF ALTERNATIVE alt, MEMBER mem);
MODE MEMBER=UNION(NOTION, SYMBOL),
     NOTION= C as required C,
     SYMBOL= C as required, but not euivalent to NOTION C;

OP +=(ALTERNATIVE a, MEMBER m)ALTERNATIVE:
                MEMBERS(HEAP ALTERNATIVE:=a, m);
```

The use of the VOID in the mode of ALTERNATIVE happily allows us to use the void denotation EMPTY for empty alternatives.

Example

```
BEGIN
     NOTION expression, term, factor,
     SYMBOL plus, times, open, close, identifier;

     GRAMMAR g=

          expression= expression+ plus+ term↑
               term*

          term= term+ times+ factor↑
               factor*

          factor= open+ expression+ close↑
               identifier ;

     SKIP
END
```

These operators mean that if a grammar is mis-formed, the program will fail to compile. For instance, trying to define a symbol as a rule will cause some message about = not being declared for that pair of modes.

Conclusion

Operator and priority declarations have hitherto been regarded as a feature allowing programmers to define expressions for their own needs, and as such, only as a convenience. However, it appears that an additional advantage is that they allow constructs that would not normally be regarded as expressions to be written in a natural and straightforward manner, eliminating the need for some special-purpose translators. The use of a system such as described here may well give us a different, more dynamic, view of grammars to the static one we are used to.

Note. Those of us who use the British Algol68r dialect implementation that does not allow uniting in firm positions, will have to declare *,↑ and + separately for the elements of the unions, eg. * for (rule,rule)grammar as well as (grammar,rule)grammar, and so on.
This does result in quite a few more definitions.

AB42.4.8

# REMARKS ON ABSTRACTO*

Leo Geurts
Lambert Meertens
Mathematisch Centrum, Amsterdam

## 1. ABSTRACTO LIVES

If an author wants to describe an algorithm, he has to choose a vehicle to express himself. The "traditional" way is to give a description in some natural language, such as English. This vehicle has some obvious drawbacks. The most striking one is that of the sloppyness of natural languages. Hill [1] gives a convincing (and hilarious) exposition of ambiguities in ordinary English, quoting many examples from actual texts for instructional or similar purposes. The problem is often not so much that of syntactical ambiguities ("You would not recognise little Johnny now. He has grown another foot.") as that of unintended possible interpretations ("How many times can you take 6 away from a million? [...] I can do this as many times as you like."). A precise and unambiguous description may require lengthy and repetitious phrases. The more precise the description, the more difficult it is to understand for many, if not most, people. Another drawback of natural languages is the inadequacy of referencing or grouping methods (the latter for lack of non-parenthetical parentheses). This tends to give rise to GOTO-like instructions.

With the advent of modern computing automata, programming languages have been invented to communicate algorithms to these computers. Programming languages are almost by definition precise and unambiguous. Nevertheless, they do not provide an ideal vehicle for presenting algorithms to human beings. The reason for this is that programming languages require the specification of many details which are relevant for the computing equipment but not for the algorithm proper. The primitives of the programming language are on a much lower level than those of the algorithm itself.

The evolution of high-level programming languages is one in which the level of the available primitives increases towards the abstractions that human beings use when thinking about algorithms. Still, the gap is very, very large. Unfortunately, recent progress is not yet reflected in any major, generally known programming language.

However, high-level programming languages have had a direct influence on the presentation of algorithms in the literature. Many an author now employs a kind of pidgin ALGOL to express himself. The pidgin characteristics are all present: (a) the language is primarily a contact language, used between persons who do not speak each other's language; although each "speaker" may have his own variant, there is mutual understandability; (b) there is a limited vocabulary, and the syntax is stripped down to the bare necessities, with elimination of the grammatical subtleties that can only be mastered by a regular user; (c) the language is not frozen but permits adaptation to various universes of discourse. The main advantages to the author (and his audience) are that there is no need for a preliminary and boring exposition of the algorithmic notation, that mathematical notions and notations may freely be employed, and that the resulting description is sufficiently precise to convey the algorithm

---

* This paper is registered at the Mathematical Centre as IW 97.

without the deleterious burden of irrelevant detail.

This pidgin ALGOL is a language. It is not really a programming, nor a natural language, but it has characteristics from both. It is not steady, but evolving. How it will evolve we cannot know. But as any man-made thing, its evolution can be influenced by our conscious effort. This language on-its-way may be dubbed Abstracto. (The name "Abstracto" arose from a misunderstanding. The first author, teaching a course in programming, remarked that he would first present an algorithm "in abstracto" (Dutch for "in the abstract") before developing it in ALGOL 60. At the end of the class, a student expressed his desire to learn more about this Abstracto programming language.)

Abstracto '77 is a clumsy language, like any pidgin. Only when a pidgin language becomes a mother tongue, which is not picked up in casual contacts but is the primary language one learns and uses, can it become the versatile tool that allows the expression of complicated thoughts in a natural way.

There are at least two reasons for programming-linguists to study Abstracto. The first is that we may hope to speed up the evolution of Abstracto, by proposing and using suitable notations for important concepts, either derived from existing programming languages, or newly coined. (An excellent example are Dijkstra's guarded commands.) The second is that Abstracto may show us how to design better programming languages.

## 2. THE LANGUAGE OF MATHEMATICS

It is possible to draw a parallel with the language of mathematics. Only a few centuries ago, the simplest algebraic equation could only be described in an unbelievably clumsy way. This very clumsiness stood directly in the way of mathematical progress.

Take, for example, Cardan's description of the solution of the cubic equation $x^3 + px = q$, as published in his Ars Magna (1545). The following translation from Latin is as literal as possible, with some explanations between square brackets that would have been obvious to the mathematically educated sixteenth-century reader:

> RULE
> Bring [Raise] the third part of the number [coefficient] of things [the unknown] [i.e., p] to the cube, to which you add the square of half the number [coefficient] of the equation [i.e., q], & take the root of the whole [sum], namely the square one, and this you will [must] sow [copy], and to one [copy] you join [add] the half of the number [coefficient] which [half] you have just brought in [multiplied by] itself, from another [copy] you diminish [subtract] the same half, and you will have the Binomium with its Apotome [respectively], next, when the cube root of the Apotome is taken away [subtracted] from the cube root of its Binomium, the remainder that is left from this, is the estimation [determined value] of the thing [unknown].

Nowadays, there is a large basic arsenal of mathematical notions and corresponding notations that may be freely used without further explanation. Each specialism has, in addition, its own notations. Nevertheless, each author is free to introduce new notations as the circumstances require.

Which notations survive in the struggle for life is determined by several factors, of which the ease of manipulating expressions is probably

the foremost one. Still, several notations may coexist, each with its own advantages and disadvantages (like Newton's versus Leibnitz's notation for derivatives). Generally, mathematicians do not bother too much about syntactical ambiguity and do not even stoop down to indicate operator priorities, as long as the intended meaning is conveyed to the gentle reader. (How different from that adversary, the automaton!)

The wildgrowth of notations in new fields can, under circumstances, be effected beneficially by a more or less authoritative body (possibly one person). Donald Knuth's proposal [2] for, among others, the use of a Greek letter theta to denote the class of functions of some order, constitutes an intervention for lack of an established notation. Such interventions are not to be confused with standardization efforts! Only in a frozen field is it possible to standardize, else we have a case of death by premature exposure to frost (hopefully of the standard).

It is difficult to characterize what constitutes good notational practice. Not only is "elegant" vague, but where notation is concerned, it is just a synonym for "good to use". Some criteria are: conciseness, similarity to notations for similar concepts, and relative independence of context. There are, of course, enough dubious notations, such as lim f(x) = a, where the equality sign has a subtly different meaning. (An extremely bad case in ALGOL 60 is the switch declaration SWITCH s := l1, 12, 13.)

## 3. IN SEARCH OF ABSTRACTO 84

We expect that the introduction of better notations will prove as important for the development of "algorithmics", as it has been - and still is - for mathematics. One must, of course, first identify the concepts before a notation can be developed. It seems unlikely that progress will come from selecting mind-blowing concepts, if only because it is hard enough to think about algorithms without having one's mind blown. If the parallel with mathematics is not deceptive, the important point is the manipulation of "algorithmic expressions". From a paper by Bird [3], describing a new technique of program transformation, we quote: "The manipulations described in the present paper mirror very closely the style of derivation of mathematical formulas [...] As the length of the derivations testify, we still lack a convenient shorthand with which to describe programs, but this will come with a deeper understanding about the right sequencing mechanisms."

At first sight it may seem attractive to view an algorithm as a (constructive) solution satisfying a correctness formula

$$\{p\}\ X\ \{q\}.$$

One can develop a notation, like Schwarz's generic command $p \Rightarrow q$ [4], for a solution (or the set of solutions) of the correctness formula. There must be some constraint on the variables that may be altered by the algorithm, since it is hardly helpful to know that

$$x = x_0 \wedge y = y_0 \Rightarrow x = GCD(x_0, y_0)$$

is solved by

$$x := x_0 := y_0 := 3.$$

If v stands for the alterable variables, and we write q[v := e] for the result of substituting e for v in q, then p ⇒ q can already be expressed in Abstracto '77 by

$$v := \epsilon \{e : p \supset q[v := e]\},$$

where "$\epsilon$" denotes the (indeterminate) selection operator.

If one interprets p ⇒ q at the same time as a formula expressing the (proved) existence of a solution, some proof rules may be given. For example, we have a proof rule

$$\frac{p \supset q[v := e]}{p \Rightarrow q}$$

(corresponding to the solution v := e), the proof rule

$$\frac{p \Rightarrow q,\ q \Rightarrow r}{p \Rightarrow r}$$

(corresponding to p ⇒ q; q ⇒ r), and the proof rule

$$\frac{p1 \Rightarrow q1,\quad p2 \Rightarrow q2}{p1 \vee p2 \Rightarrow q1 \vee q2}$$

(corresponding to IF p1 → p1 ⇒ q1 ▯ p2 → p2 ⇒ q2 FI). By turning a derivation of p ⇒ q upside down, a solution is constructed. Unfortunately, there is no suitable rule for a solution of the form

DO b → p ∧ b ⇒ p OD.

(The rule

$$\frac{p \wedge b \Rightarrow p}{p \Rightarrow p \wedge \neg b}$$

does not express termination and allows the derivation of p ⇒ p ∧ ¬ b for arbitrary p and b.)

There are several other courses one may follow to search for more constructive elements of Abstracto. One is similar to the way high-level programming language elements originate: consider existing (Abstracto) programs, and find similar "code sequences" that appear to be the expression of the same more abstract concept. Just like

```
L1: IF NOT condition GOTO L2
    perform something
    GOTO L1
L2:
```

may be expressed more clearly by

```
DO condition → perform something OD,
```

one might wish to express

```
vopt := ∞;
FOR e ∈ s
DO IF ok1 (e)
   THEN IF v < vopt
        THEN eopt, vopt := e, v
        FI WHERE v = f1 (e)
   ELIF ok2 (e)
   THEN IF v < vopt
        THEN eopt, vopt := e, v
        FI WHERE v = f2 (e)
   FI
OD
```

as

```
eopt, vopt := FOR e ∈ s
              OPT ok1 (e) → f1 (e)
                □ ok2 (e) → f2 (e)
              TPO.
```

(This is not a serious proposal, but neither is it a mere joke.)

Instead of this bottom-up approach a more analytical consideration of the human way of thinking about algorithms may prove, in the long run, more fruitful. In contrast to the process of developing a program, given an algorithm, it appears that little is known about this subject. Descriptions of algorithms in natural languages do not provide much insight, presumably because of the poor expressiveness for algorithmic notions. (One tendency, however, is very noticeable, and is maybe an indication that is worth following up: what might be called the "and-so-on" descriptions, and the "afterthoughts". We surmise that this reflects the emergence of algorithms as the jump to the limit of a sequence of approximations.)

Perhaps the best approach is the following. Suppose a textbook has to be written for an advanced course in algorithmics. Which vehicle should be chosen to express the algorithms? Clearly, one has the freedom to construct a new language, not only without the restraint of efficiency considerations, but without any considerations of implementability whatsoever.

The following is an attempt to indicate some desiderata for Abstracto 84.

Orthogonality is a must. For a lingua franca without frozen and formal description, exceptions are out of the question.

Abstracto 84 has an ALGOL flavor, but is certainly not committed to the control structures or any other particular construct of any ALGOL whatsoever.

With the exception of truth values, Abstracto 84 has no predefined types, but only ways to construct new types from "application oriented" types. Operations on objects are outside the realm of Abstracto 84 proper, except such operations as have a generic meaning for a class of types constructed by means provided by Abstracto 84 (cf. Wilkes [5]).

Although there are variables for objects of any type, these variables

are not considered as new objects. There are no pointer values (except when introduced for a specific application).

Similarly, procedures are not considered as objects which may be assigned etcetera.

Conditions may contain defining identifiers which are also bound in the controlled clause selected if the condition succeeds.

## 4. GLIMPSES OF ABSTRACTO 84

Due to our near-sightedness, it is difficult to discern more than some outlines of Abstracto 84. Of some prominent features a glimpse may now and then be caught. It should go without saying that all mathematical notation remains welcome to Abstracto.

First of all, it is clearly settled, even in this early stage, that Abstracto is rich in "iterators" (operators or other constructs that operate on generators in an Alphard-like sense). For example, one may write a condition

$$\exists\ e \in s\colon p(e),$$

and if this succeeds, then in the scope of the selected clause, if any, e accesses some element from s satisfying the predicate p. Such constructions may provide a clear and concise description that is quite close to the algorithm originally conceived. Also, if it is immaterial for the algorithm in which order elements are selected, it is important that this be expressed.

The control structures of Abstracto 84 seem to be centered around guarded command sets (Dijkstra [6]) of the form:

$$C_1 \to S_1 \ \square\ C_2 \to S_2 \ \square \ \ldots \ \square\ C_n \to S_n.$$

The basic meaning of such a form is: if at least one of the $C_i$ holds (where the evaluation of a condition is supposed to have no side effects), then some corresponding $S_i$ is selected (but not yet evaluated). In the terminology of the ALGOL 68 Report, a scene is selected, composed from that $S_i$ and an environ whose most recent locale may have been added because of the declarative form of $C_i$.

The meaning of IF ... FI and DO ... OD may now be defined easily. It appears, however, that in Abstracto 84 several other control structures may be defined with the guarded commands at their cores, as suggested by the FOR ... OPT ... TPO construct in the previous section. The basic simplicity of the concept, in conjunction with its indeterminacy, should warrant ease of manipulation.

Many types, specifically those that can be treated satisfactorily by so-called axiomatic/algebraic specifications, can be defined in the way exemplified below:

```
tree ::= nil | atom (val: item) | pair (left, right: tree).
```

(We write "::=" to stress the similarity with BNF, although this "syntax" of objects is more abstract than usual, since the nodes in the "parse tree" of an object are labelled; in the example, "nil", "atom" and "pair" are node labels.) This notation is similar to Hoare's notation for recursive

data structures [7]; it carries no other information than is relevant from an abstract algorithmic point of view. There are three nice things about this way of defining types. In the first place, it is easy to derive in a straightforward way "axiomatic" specifications in the style of Guttag [8], but the notation is much more compact. (For the above example, we would obtain nine lines for the discernible functions and eighteen for the axioms.) Secondly, this way of defining offers a unification of three well-known concepts:

records, as in

```
complex ::= pair (re, im: real);
```

(disjoint) unions, as in

```
arithmetical ::= i (val: int) | r (val: real);
```

PASCAL scalars, as in

```
color ::= red | blue | green.
```

Finally, it is easy to instruct a compiler to handle such definitions. The only drawback is the inefficiency, reason why such definitions are maybe Abstracto rather than Concreto.

Objects of a thus defined type can now be subjected to a "conformity condition", as in

```
DO t FITS
   pair (t1, t2) → t := t2
OD.
```

In this example, if the condition succeeds, t2 accesses the tree t.right.

## 5. A POSSIBLE PITFALL

Unless we are very mistaken, program development by successive "program transformations", i.e., a sequence of manipulations on expressions which represent algorithms, has a promising future. Each transformation rule is a theorem. To us, computer maniacs, the perspective is tempting to create a data base of transformations to be applied mechanically. Since the applicability of each transformation is also checked mechanically, we have done away with all bugs (except for those in the original, pure, algorithm, possibly a problem specification). What vista! Of course, we must invent for our Abstracto language some syntactic notions to allow expression of the applicability of transformations.

The last sentence should make it clear already that the pursuit of this Utopian concept - unless one contents oneself with trivial transformations that might as well be applied directly by a compiler - spoils the simplicity of Abstracto. Worse yet, the concept wholly ignores the fact that in mathematics for none but the simplest theorems the applicability may be checked by "syntactical" means. If computers would have dated back to the inception of modern mathematical notation and only mechanizable transformations would have been studied, the so-called special products would, presumably, still be among the high-lights of mathematical knowledge.

To quote once more Bird [3]: "we did not start out, as no mathematician

ever does, with the preconception that such derivations should be described with a view to immediate mechanization; such a view would severely limit the many ways in which an algorithm can be simplified and polished."

REFERENCES

[1] Hill, I.D., Wouldn't it be nice if we could write computer programs in ordinary English - or would it?, Computer Bull. 12 (1972) 306-312.

[2] Knuth, D.E., Big omicron and big omega and big theta, SIGACT News 8 (1976) 2, 18-24.

[3] Bird, R.S., Improving programs by the introduction of recursion, Comm. ACM 20 (1977) 856-863.

[4] Schwarz, J., Generic commands - a tool for partial correctness formalisms, Computer J. 20 (1977) 151-155.

[5] Wilkes, M.V., The outer and inner syntax of a programming language, Computer J. 11 (1968) 260-263.

[6] Dijkstra, E.W., Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM 18 (1975) 453-457.

[7] Hoare, C.A.R., Recursive data structures, Stanford University Report CS-73-400 (1973).

[8] Guttag, J.V., Abstract data types and the development of data structures, Comm. ACM 20 (1977) 396-404.

AB42.4.9 'ABSTRACTO' PROJECT FOR AN ALGORITHM SPECIFICATION LANGUAGE

R. DEWAR, J. SCHWARTZ

COURANT INSTITUTE OF COMPUTER SCIENCES

DECEMBER 1977

## 1. Semantic and Syntactic Aspects of Programming Languages

Any programming language has two aspects which together determine much of its character. The first of these, which can be called the semantic side of the language, is defined by the *set of primitive operations which the language makes available.* It is useful to think of these as the operations of a logical machine, which we shall call the *interpreting machine* of the language. Then we can make the following remarks:

(a) If, for reasons of efficiency, the interpreting machine of a language L is restricted to operations which have a very elementary, hardware-like character, then L is bound to be of relatively low level.

(b) The development of progressively more powerful programming languages has been bound up with a progressive enrichment of the family of operators provided by the interpreting machines of these languages. However, language designers have often been reluctant to use operations until it become clear that these operations could be implemented very efficiently. The following is a rough account of major steps taken to date in the enrichment of interpreting automata:

i. FORTRAN: static call/return primitives; I/O operations.

ii. ALGOL-60: recursive call/return primitives, stack allocation.

iii. PL/I: allocate/free primitives for space allocation, pointers, structured records.

iv. ALGOL-68, LISP: allocation primitives supported by garbage collection, combined, in the ALGOL-68 case, with a structured record facility.

v. SIMULA: (also other simulation languages): multiple pseudoparallel processes, supported by process-creation and coroutine-call primitives.

vi. APL, SETL: arrays, sets; array, set, and map operations.

vii. SNOBOL (also various A.I. languages such as PLANNER,etc.): backtracking primitives.

Part of what we wish to suggest is that a deliberately radical leap forward from the semantic level reached thus far would be useful.

The second major aspect of a language, which can be called its syntactic side, appears as the external form of the language. Syntactic design aims to provide a *set of notations* which conduce to effective, correct use of a language's underlying semantic primitives. Significant subsidiary aims are as follows:

i. The notations should help to localize the network of logical relationships upon which correct program functioning rests, and should have the property that the logical import of a code submodule is conveyed as clearly as possible by its external form. This is partly a matter of providing adequate syntactic mechanisms for grouping and arranging code fragments.

ii. A language's syntax can include rules of redundancy which cut down on the likelihood that certain types of common errors will go undetected. These rules, which a compiler can enforce, can require a programmer to provide material apt to have significant documentary value, to arrange his code in ways apt to enhance readability, etc.

iii. A language's syntactic processor can include mechanisms which facilitate modest language extensions.

Some otherwise very interesting languages, such as APL and SNOBOL, have conspicuously ignored goals (i) and (ii). LISP certainly ignores goal (ii), while ALGOL-68 certainly achieves it in large measure. The design of SIMULA is interesting in connection with goal (i), which has also shaped the design of such recent languages as CLU and ALPHARD.

## 2. Pure Specification Languages

Whatever its syntax and semantics, a programming language serves two purposes. On the one hand, it serves a vehicle for communication with a computer. On the other hand, it is a system for *definitive specification of algorithms*. From the viewpoint of this latter purpose (and ignoring the former) efficiency is irrelevant. Thus a pure specification language can have the following character:

(a) Its underlying collection of semantic primitives can include any operations which are generally useful, rigorously and simply definable, and heuristically transparent, irrespective of the possibility of implementing these operations at all, let alone efficiently.

(b) Its syntactic structure can follow that of the best current languages, since syntax seems less bound by efficiency considerations than does the choice of a family of underlying operations.

Even if unimplementable, a pure specification language could be used as follows:

i. It can have a compiler, and be compiled. This would allow correctness checks based on redundancy to be performed, and some structural and documentation standards to be imposed.

ii. It can form part of a program verification system which allows the correctness of algorithms to be proved rigorously.

iii. It can be part of a program transformation system, which allows initial program variants to be transformed semi-mechanically into equivalent, but perhaps more efficient (e.g. implementable) forms.

iv. It can be used for explaining, teaching, and studying algorithms and large systems of algorithms.

Donald Knuth has remarked that *premature optimization is the root of all evil in programming*. But if no recognized language provides sufficiently powerful semantic primitives, premature optimization (at some level) must remain an inescapable part of the very statement of algorithms. Only a determinedly 'high level' specification language can allow algorithms to be stated in their most direct 'root' forms.

## 3. Semantic Primitives for a Pure Specification Language

It is clearly desirable for the semantic primitives of a specification language to be objects and operations in terms of which mathematical reasoning can conveniently and directly be conducted. In view of the central importance of set theory and predicate calculus within logic, this suggests the use of finite and infinite sets and sequences. More highly structured objects such as trees or graphs can readily be defined in terms of these primitive objects, though of course there may be some advantage in making such objects available directly. The following are a few significant operations:

(a) Map application $f\{x\} = \{y \mid [y,x] \in f\}$, and its n-parameter generalizations. Also, the corresponding 'storage' operator $f\{x\}:=s$, and the n-parameter generalizations of this.

(b) Map multiplication $f.g = \{[x,y] \mid (\exists z)[x,z] \in g \& [z,x] \in f\}$, together with its various n-parameter generalizations.

(c) Map inversion.

(d) Transitive expansion of a map: $f^*$ is the set of all sequences $[x_1, x_2, \ldots]$ such that $x_{2+1} \in f\{x_n\}$. Various n-parameter generalizations of this operation are also useful.

(e) Various operations on sequences, including concatenation, insertion, and selection of the first index and of the sequence of indices at which a given predicate is satisfied.

(f) Union, intersection, etc.

Any programmed function which references no variable external to itself can be considered to define a mapping with infinite domain. Note that all the operations of set theory extend in an obvious way to maps of this kind. In an implemented specification language, set theoretic operations applied to such mappings will become operations of $\lambda$-calculus type which bind parameters, compose functions, apply function calls, etc.

Semantic primitives significant for control flow are:

(a) Procedure call with recursive stacking.

(b) Backtracking. An elegant version of this can be based upon a unary nondeterministic selection operation $\in$, for which $\in$s selects an arbitrary element of the set s, with backtracking to the point of selection in case of subsequent failure. Significantly, this very powerful primitive admits a quite simple proof rule. This operator also has interesting set theoretic connections, and suggests various potentially useful constructs and notations.

(c) Pseudoparallelism with an <u>await</u> primitive. Although the proof rules for this construct are not simple, pseudoparallel dictions afford the most natural way of modeling real-world situations which are inherently parallel. In situations of this kind, verifiable conformity to an external situation, rather than logical correctness in any abstract mathematical sense, is often the crucial issue.

## 4. <u>Syntactic Forms for a Specification Language</u>

As stated, we expect present syntactic forms, including procedures, calls, <u>if</u>,<u>while</u>, and <u>case</u> statements, etc. to retain much of their present form in a high-level specification language. Syntax appropriate for the representation of nondeterminism and of multiple pseudoparallel processes may also be needed. Additional dictions which describle important program transformations may also be needed: for example (see section 5 below) dictions indicating that the values of particular subexpressions are to

be kept on hand and updated differentially rather than being calculated *de novo* when needed.

An important issue is the extent to which syntactic forms which rigourously isolate program layers from each other will be provided. Object-type extension mechanisms providing this kind of isolation may be desirable. If provided, such extension mechanisms should preserve the abstract and general character of the specification language itself.

Syntactic extension mechanisms of a more general sort may also be desirable. In fact, it may be appropriate to give the specification language, not one fixed syntactic form, but only some clean internal syntax with which a variety of parsers can communicate. Of course, even if this is done, it will still be desirable to publish some initial external syntax attractively representing the facilities of the language.

## 5. Implementation of Subsets of a Specification Language

Purely manual inspection of an algorithm's text can easily overlook flaws, and full formal algorithm verification is still prohibitively expensive. It is therefore still useful to execute an algorithm in a well-chosen variety of test cases, which can at least enhance one's confidence that egregious blunders of various kinds have been eliminated. Example: let a directed graph g be given as a set of pairs [x,y]. Then the graph contains a cycle if and only if there exists a subset s of the graph such that the end of every edge in s is the start of some other edge in s. Thus in a set-theoretic specification language we can write the following one-line test for the existence of a cycle:

there_is_a_cycle:= ∃s ∈ power_set(g) |(∀x∈s|(∃y ∈ s) x(2) = y(1)).

But this is wrong! To state the test correctly, we must exclude the case s = null_set, and write

there_is_a_cycle:= ∃s ∈ power_set(g)|(s ≠ null_set & (∀x ∈ s |
(∃y ∈ s) x(2)=y(1))).

Inspection can easily overlook, while testing will immediately reveal, the flaw in the first version of this algorithm.

For this and other reasons we judge it quite useful to implement as broad a subset of the specification language as is feasible. As a matter of fact, it now appears likely that rather broad subsets can be implemented at a reasonable level of efficiency. We may therefore hope that such an implementation can be developed as a valuable tool for program prototyping and experimentation with algorithms.

A perspective rather different from the one we have set out would regard this possibility rather pessimistically, consider efficiency to be an inescapably central issue in programming language design, and prefer therefore to make available, not a family of powerful abstract semantic primitives, but rather a kit of extensibility tools based upon efficient, near machine-level operations only. Such an approach will aim to facilitate the build-up of efficient variants of generally useful abstract operations in forms tailored to the particular contexts in which they are to be applied. Ideally, this would be done in a syntactic framework structured to isolate the 'inner' detail of these operations and the objects on which they act from 'outer' layers of the algorithms in which they are used.

Against this perspective, we raise the following objections:

(a) From the pure specification point of view, it is surely preferable to admit all well-known, precisely defined, general, heuristically appealing operations into one's initial collection of primitives. If necessary, layered extension can begin from, rather than culminate at, this point.

(b) Even to know what general systems of operations can be efficiently implemented is difficult. To design an efficient implementation for them is more difficult still. A good implementation of a specification oriented language will attack this problem in a concentrated way, once, and can

provide a spectrum of operations, some of which retain substantial efficiency since they only relize logically simple primitives, and this only in favorable contexts. A user left to his own devices is likely to founder in the very process of realizing the high-level operations which he needs, and may indeed not even realize what these are. A community of users which is not carefully coordinated is likely to suffer from incompatibilities if complex objects and operations need to be designed. Example: To design map-like structures which allow records to be accessed efficiently via simple key-word lookup is not hard; but to go on from this to the design of efficient structures which allow records, or perhaps entire substructures, as search keys is not substantially easier than to implement major portions of a full specification language.

## 6. Research Efforts Which Wide Acceptance of a Specification Language Would Facilitate and Focus

A number of techniques which allow concise but inefficient algorithms to be transformed into more efficient forms have been recognized. Among these are:

(a) Elimination of backtracking by use of pre-compiled success/failure tables. This technique can, for example, transform simple very general backtracking parsers into efficient LR and LL parsers.

(b) Transformation of recursions into much more efficient iterations. Rules governing a variety of transformations of this type have been collected by Darlington, Burstall, Strong, and Walker.

(c) Abstract strength reduction. Here we keep the value of one or more expressions constantly up-to-date, so as to avoid expensive, from-scratch calculation of these expressions.

(d) Choice of particularly advantageous data structures, as allowed by logical context.

These techniques, while still fragmentary, point to a style of *programming by transformation* in which detailed, possibly efficient algorithm versions are derived from condensed abstract initial specifications by explicit transformations. Two styles of transformation can be conceived. The first of these is a *system guaranteed* style, in that it takes place within a system which will only permit a transformation to be applied when equivalence of an untransformed program with its transformed version is certain. A second possibility is to operate without this guarantee, thereby leaving certification of the logical propositions needed to justify a transformation to the programmer. Thus, for example, a hypothetical declaration

represent x,y by list

can be used to transform each set union operation x:= x ∪ y appearing in some program into a list concatenation, (and also to transform each iteration over x to a list iteration), leaving it to the programmer to guarantee that x and y have disjoint values at each point at which a set union operation is actually replaced by a list concatenation. Both transformational styles realize the notion of 'top-down' programming in a specific technical sense and appear as promising directions for future development.

Transformation by declared choice of data structures appears to be a presently feasible technique promising substantial advantage. A system-guaranteed variant of this form of transformation can supply data structures which either realize abstract operations efficiently, or abort if some assumption implicit in a declared data choice facts.

Experience with program transformation will reveal typical transformational patterns and make explicit the logical assumptions on which these transformations depend. Where widely advantageous transformations are seen to rest on relatively superficial

logical assumptions, it will be possible to design global program analyser/optimizer routines which automatically justify and apply these transformations. Thus the effort we propose should derive advantage from, but also serve to focus, the rapidly developing mass of work on program optimization.

# AB42.4.10 ON LANGUAGE DESIGN FOR PROGRAM CONSTRUCTION

by
Michel Sintzoff
Centre de Recherche en Informatique *
Nancy, France

ABSTRACT. It is argued that the structure of a specification and programming language must be based on program formation rules which are well defined, well justified and easily applicable. Two related case studies are considered : how to eliminate risks of failures from non-deterministic programs, and how to design communicating programs in terms of non-deterministic ones. It is shown that the discovery of adequate construction rules is essential as well as very difficult, and that such rules provide a necessary guidance in the quest of simple and systematic means for specifying and expressing algorithms.

## 1. INTRODUCTION

A programming language should be a language for programming, viz. a language for making programs. It is then sensible to base the structure of such a language on good program formation rules. Good rules are rules which have the following characteristics : their definition is foolproof; it is proved that they inevitably yield some desired property of programs; and their use is easy in the sense that the rôle of the human intuition is strictly reduced. This excludes verification rules, which come too late in program design, as well as equivalence rules which give no guarantee of convergence in the program design process; for instance, the use of the fold- and unfold-transformations is very delicate when no adequate strategy is provided.

To substantiate our discussion, we shall develop two case studies : how to improve non-deterministic programs by reducing the risks of failures such as non-termination or abortion, and how to design communicating programs (i.e. programs without common access to global data spaces). More general views will be presented in the last section.

---

* until February 1978. Permanent address : MBLE Research Laboratory, 2, av. Van Becelaere, B-1170 Brussels.

## 2. INCREASING THE CHANCES OF SUCCESS

Consider the following, arch-known problem : to reach a goal $q \equiv a > 0 \wedge a = b$ by successive applications of the rules $f_i$ from the system ff hereafter.

$$ff \begin{cases} f1 : \underline{true} \to a := a-b \\ f2 : \underline{true} \to b := b-a \\ f3 : \underline{true} \to a := -a,\ b := -b \end{cases}$$

One of the difficulties, of course, is that ill-chosen successive applications of f3 may never reach q. Let us then try to find more intelligent conditions $y_i$ for the substitutions (or assignments if you so wish) in $f_i$. We denote by $\hat{ff}$ such an improved system :

$$\hat{ff} \begin{cases} \hat{f}1 : y1 \to a := a-b \\ \hat{f}2 : y2 \to b := b-a \\ f3 : y3 \to a := -a,\ b := -b \end{cases}$$

In order to ensure some sort of completeness, we do require that the domain of possible success by ff is equal to the domain of guaranteed success by $\hat{ff}$ : in other words, if we start in a state from which there is some way to reach q by ff, then by using $\hat{ff}$ we will be certain to reach q without any risk of non-termination or of dead end.

The central question is thus : how to find the conditions $y_i$'s ? Here are a number of ways.

(1) An omniscient oracle gives the $y_i$'s. If we do not trust the oracle, we verify his gift by using a Floyd-like termination function or a Dijkstra-like weakest precondition. We reject this because the availability of oracles, the invention of functions and the induction of limits of predicates are all too difficult.

(2) The conditions $y_i$'s are obtained themselves as fixed points of predicate transformations. We abandon this for the same reason as above.

(3) The conditions $y_i$'s are directly driven by a complexity function, akin to a termination function and to be invented, as proposed by H. Boom. This is to be rejected too.

(4) Recalling functional analysis, we try to approximate the unreachable $y_i$'s by feasible conditions $\bar{y}_i$ or $\hat{y}_i$ (the bar for sea level and the circumflex for mountains) : an $\bar{y}_i$ approximates $y_i$ from below, i.e. $\bar{y}_i \supset y_i$, it is safe, but may be difficult; each $\hat{y}_i$ is an approximation from above, viz. $y_i \supset \hat{y}_i$, it is unsafe and rather easy. Both approximations can be quite imprecise, since $\underline{false} \supset y_i$ and $y_i \supset \underline{true}$. Since we found no constructive way to ensure a sufficient precision, we have abandoned this method also. The techniques (2) and (4) are discussed in [ 3 ], where more detailed references are given.

We now present still another method, which again is approximate and yet yields complete solutions when it does succeed. This method is based on sufficient criteria for the equality of the fixed points of two functions of predicates. We shall keep our formal apparatus to a minimum, and present here only a basic version of the proposed method.

Definitions. A system ff is made of rules $f_i$ each of which consists of a condition p and a substitution s. The backwardsapplication of such a system to a predicate r is noted ff.r and is simply defined by

$$ff.r = \bigvee_i f_i.r$$

$$(p \to s).r = p \wedge (s.r), \qquad \text{or} \qquad p \wedge (s.r)\ \{p \to s\}r$$

$$(\ldots a_i := E_i \ldots).r = r[\ldots a_i \leftarrow E_i \ldots]$$

($\leftarrow$ for substitution)

For instance, in our g.c.d. example above, $ff.q \equiv a > b \wedge a = 2b \vee a > 0 \wedge 2a = b \vee -a > 0 \wedge -a = -b$. Moreover, we use the following obvious notations :

$$f+g \equiv \lambda r.(f.r \vee g.r)$$
$$f;g \equiv \lambda r.(f.(g.r))$$
$$\underline{1} \equiv \lambda r.r$$
$$p \to (p' \to s) \equiv (p \wedge p') \to s$$

We observe that $(f;(g+h)) \equiv ((f;g) + (f;h))$. We recall that the domain of possible success by ff for a given goal q is the least fixed point of the function $\lambda r.(q \vee ff.r)$; the domain of guaranteed success by ff for q is the least fixed point of $\lambda r.(q \vee ff.r \wedge \neg\, ff.\neg r)$, viz. the weakest precondition; ff is continuous.

Theorem 1. Consider a goal q and a system $ff = \sum_{i=1}^{N} f_i$.

IF the conditions $y_i$ verify, for all i,j, k = 1...N,

(0) $y_i \wedge y_j =$ false for $i \neq j$

(*) $f_i.q \supset q \vee \bigvee_{j<i} f_j.q \vee y_i \vee \bigvee_{j>i} y_j \wedge f_j.q$

(**) $f_i.y_k \supset CR[(f_i;f_k), \underline{1}+ff] \vee \bigvee_{j<i} CR[(f_i;f_k), (f_j;ff)]$

$$\vee\ y_i \vee \bigvee_{j>i} y_j \wedge CR[(f_i;f_k), (f_j;ff)]$$

where $CR[h, \sum_i g_i] = \bigvee_i CR[h, g_i]$

and $CR[h,g] = \begin{cases} \text{true} \\ \text{false} \end{cases}$ if $h.v \equiv g.v$ for each variable v, otherwise

THEN the domain of possible success by ff for q is the domain of guaranteed success for q by $\hat{ff} \equiv \sum_i \hat{f}_i$ where $\hat{f}_i \equiv (y_i \to f_i)$.

Proof. Since $\hat{ff}$ is deterministic because of (0), the domains of guaranteed success and of possible success by $\hat{ff}$ are identical. Let us denote by zz the domain of possible success using ff and, similarly, $\hat{zz}$ w.r.t. $\hat{ff}$. We shall prove $zz \equiv \hat{zz}$ by induction; zz is the limit of the iterates

$zz^0 := q$; $zz^{n+1} := zz^n \vee ff.zz^n$, or $zz^n \vee \bigvee_i f_i.zz^n \wedge \neg \bigvee_{j<i} f_j.zz^n$

Similarly for $\hat{zz}$, using

$\hat{zz}^0 := q$; $\hat{zz}^{n+1} := \hat{zz}^n \vee \bigvee_j y_j \wedge f_j.\hat{zz}^n$

Subthesis 0 : $zz^0 = \hat{zz}^0$; clear.
Subthesis 1 : $zz^1 = \hat{zz}^1$,
i.e. $q \vee \bigvee_i f_i.q \wedge \neg \bigvee_{j<i} f_j.q = q \vee \bigvee_j y_j \wedge f_j.q$;
this is implied by (*).
Subthesis 2 : for all $n \geq 1 : zz^{n-1} = \hat{zz}^{n-1}$, $zz^n = \hat{zz}^n \vdash zz^{n+1} = \hat{zz}^{n+1}$.
The consequent amounts to (for all i)

$$f_i.zz^n \supset zz^n \vee \bigvee_{j<i} f_j.zz^n \vee y_i \vee \bigvee_{j>i} y_j \wedge f_j.zz^n$$

given the definitions of $zz^{n+1}$ and $\hat{zz}^{n+1}$, and replacing $\hat{zz}^n$ by $zz^n$. Let us now unfold $zz^n$ by using the restrictive definition ($\hat{zz}^n$) on the lhs and the liberal one ($zz^n$) on the rhs. For all i, k,

$$f_i.y_k \wedge f_i.f_k.zz^{n-1} \supset zz^{n-1} \vee ff.zz^{n-1} \vee \bigvee_{j<i} f_j.ff.zz^{n-1}$$
$$\vee\ y_i \vee \bigvee_{j>i} y_j \wedge f_j.ff.zz^{n-1}.$$

This is indeed implied by (**) because $f_i.f_k.zz^{n-1} \supset g.zz^{n-1}$ holds when $CR[(f_i;f_k),g] = \underline{true}$.

Exercise. Let us try to apply this theorem in our small g.c.d. example :

(0)
$y2 \supset \neg\, y1$
$y3 \supset \neg\, y1 \wedge \neg\, y2$

(*)
$a=2b \wedge a>b \supset y1$
$2a=b \wedge a>0 \supset y2$
$a=b \wedge a<0 \supset y3$

We now compute the simplification criteria CR [ h,g ], and we find

CR [ (f1;f3), (f3;f1) ] = true, because

| | a | b |
|---|---|---|
| f3 | -a | -b |
| f1 | -a+b | -b |

and

| | a | b |
|---|---|---|
| f1 | a-b | b |
| f3 | -a+b | -b |

CR [ (f2;f3), (f3;f2) ] = true, similarly.
CR [ (f3;f3), 1 ] = true, which is obvious.

Hence the subsystem (**) can here be reduced to

$$(**)\quad \begin{cases} f1.y1 \supset y1 \\ f1.y2 \supset y1 \\ f1.y3 \supset y1 \vee y3 \end{cases} \quad \begin{cases} f2.y1 \supset y2 \\ f2.y2 \supset y2 \\ f2.y3 \supset y2 \vee y3 \end{cases} \quad \begin{matrix} (f3.y1 \supset \underline{true}) \\ (f3.y2 \supset \underline{true}) \\ (f3.y3 \supset \underline{true}) \end{matrix}$$

It remains to derive $y_i$'s verifying (0), (*) and (**). Note that *any* solutions are acceptable, not just, say, least ones. We can thus try to find solutions which are expressed in a very limited language; and we shall here consider nothing more than Algol-like boolean expressions on the variables a,b and the constant 0, without arithmetic operations.

We solve our system of inequations by successive eliminations. Let us start with y1; it is characterized by $a = 2b \wedge a>b \supset y1$ and $(a := a-b).y1 \supset y1$, i.e. by

$$y1 \equiv y1 \vee a=2b \wedge a > b \vee (a := a-b).y1$$

We look at the first iterates

$$y1^{(0)} := \underline{f};\ y1^{(1)} := a=2b \wedge a > b$$

The expression for $y1^{(1)}$ is already outside our expressive power; we approximate it as well as we can by an acceptable boolean expression :

$$y1^{(1)} := a > b \wedge b > 0$$

This is a drastic approximation but is quite easy to find.
Then, $y1^{(2)} := a > b \wedge b > 0 \vee a-b > b \wedge b > 0$, i.e. $a > b \wedge b > 0$, i.e. $y1^{(1)}$.
Our tentative solution for y1 is thus found :

$$y1 \equiv a > b \wedge b > 0.$$

Observe that the least solution is $y1 \equiv b > 0 \wedge \exists c : (c \geqslant 2 \wedge a=c.b)$; for us, this is too difficult to induce. The next unknown is y2, defined by the five implications

$$y2 \supset \neg(a > b \wedge b > 0)$$

$$2a=b \wedge a > 0 \supset y2 \qquad (b := b-a).a > b \wedge b > 0 \supset y2$$

$$(a := a-b).y2 \supset a > b \wedge b > 0 \qquad (b := b-a).y2 \supset y2$$

or, after simplifications, by

$$y2 \supset b \geqslant a \wedge a > 0 \qquad 2a \geqslant b \wedge b > a \supset y2$$

$$(b := b-a).y2 \supset y2$$

Using drastic approximations again, we easily find

$$y2 \equiv b > a \wedge a > 0.$$

The four remaining implications define y3. Happily, a simple expression can also be found in this case, for instance

$$y3 \equiv a \leqslant 0 \wedge b \leqslant 0.$$

Thus, we have derived the improved system

$$\hat{ff} \begin{cases} \hat{f1} : a > b \wedge b > 0 \rightarrow a := a-b \\ \hat{f2} : b > a \wedge a > 0 \rightarrow b := b-a \\ \hat{f3} : a \leq 0 \wedge b \leq 0 \rightarrow a := -a,\ b := -b \end{cases}$$

With $\hat{ff}$, we always reach $q \equiv a = b \wedge a > 0$ when starting on any initial state from which q may be reached using the original system ff (without the conditions $y_i$). Outside that domain, there is no guarantee of any sensible action : for example, q can never be reached by ff from $a = 0 \wedge b = 0$, and then $\hat{ff}$ cycles indefinitely without reaching q.

Refinements. If we want to stop when q is reached by $\hat{ff}$, we simply enforce the additional constraint $y_i \supset \neg q$. If we want to permit nondeterminism in $\hat{ff}$, we first add within ff a new rule $p1 \wedge p2 \rightarrow (s1 | s2)$ for each pair of rules $p1 \rightarrow s1$ and $p2 \rightarrow s2$; of course, $(s1 | s2).r \equiv (s1.r) \wedge (s2.r)$. If theorem 1 is not strong enough, we strengthen it; this is necessary if, in our g.c.d. exercise, we replace f3 by

f3 : true → a := -a
f4 : true → b := -b

One way to improve theorem 1 is to establish conditions (0), (*), (**) and (***) which imply $zz = \hat{zz}$ by an induction of depth three; and so on at will. Another way to strengthen the theorem is to define CR [ h,g ] in terms of weaker criteria ensuring that $\forall n : h.zz^n = g.zz^n$; for instance, we can say that CR [ h,g ] = true if $h.q \equiv g.q$ and if $(h;ff;h^{-1}) \equiv (g;ff;g^{-1})$, with the assumption $(h^{-1};h) = 1$ and $(g;g^{-1}) = 1$. Note also that the applicability of theorem 1 may depend on the order chosen for the rules in ff : a different order may yield more simplifications in (**), and may yield different solutions $y_i$'s anyway.

All these refinements are elaborated and reported elsewhere.

Discussion. Many questions are left unanswered. How to choose the best simplified inequations, (**) or (**...*) ? How to derive solutions of these inequations ? How to tackle richer program structures than just conditional substitutions ? How to know the domain zz explicitly ?

Yet, the method proposed above does have some advantages. Assume that a user specifies q and ff, that he knows the domain zz of possible success, and that a problem-solver proposes $\hat{ff}$ : then, $\hat{ff}$ can often be verified by a simple application of theorem 1, without any induction or invention whatsoever. The fact that the domain of success remains implicit is classical for parser generators : the additional contexts which allow for deterministic reduction rules do not describe the well-formed sentences of the language explicitly, and this is why these contexts can be constructed. The approximation of unfeasible properties by sufficient ones has also been the central reason of many successful methods of program checking and optimization by compile-time interpretation on non-standard models.

The g.c.d. exercise amounts to replace Malc'ev-like conditional equations defining g

$a > 0 \rightarrow g(a,a) = a$
$g(a,b) = g(a-b,b)$
$g(a,b) = g(a, b-a)$
$g(a,b) = g(-a, -b)$

by equations for computing g without entering dead alleys :

$$a = b \wedge a > 0 \rightarrow g(a,b) = a$$
$$a > b \wedge b > 0 \rightarrow g(a,b) = g(a-b,\ b)$$
$$b > a \wedge a > 0 \rightarrow g(a,b) = g(a,\ b-a)$$
$$a \leq 0 \wedge b \leq 0 \rightarrow g(a,b) = g(-a,\ -b)$$

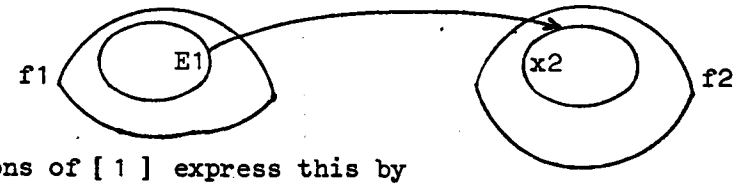
This kind of replacement is also the aim of the method worked out by Knuth and Bendix [ 2 ]; the main differences are that we consider only very restricted equations since we accept simple substitutions only, that we assume a goal q to be given, that we do tackle the case of commutative axioms such as g(a,b) = g(-a, -b), and that we use conditional equations instead of pure equations.

A theorem by Strachey (see 12.72 in [ 4 ]) gives sufficient conditions for the equivalence of two while-loops; these conditions test commutativity properties of the bodies of the loops. The spirit of that theorem is not unlike that of theorem 1, and can be traced back to Tarski's fundamental theorems on fixed points. The main difference is that theorem 1 is not an equivalence theorem : the formation of $\widehat{ff}$ on the basis of ff is strictly oriented towards an *increase* of the chances of success. This is a striking illustration of the fact that the discovery of convergent formation rules on the basis of blind transformation rules is not always trivial. See also [ 5 ] (4.4, 6.3) regarding equivalence rules.

## 3. SIMPLE COMMUNICATION CONSTRUCTS

In order to avoid the use of global variables to access a common memory, Milner, Hewitt, Dijkstra with Hoare, and others, have proposed to express all the interactions between program modules in terms of one-way communications of data. The existing proposals still lack transparent definitions of semantics or of proof rules. Our aim is to find a direct way to design communicating programs by formation rules for non-deterministic programs as developed in the previous section. We present a rather naïve attempt in this direction, shunning any concern for erudite semantics for lack of time and competence.

Consider two modules, f1 and f2, and a communication of data E1 from f1 to the local variable x2 in f2 :




the notations of [ 1 ] express this by

f1 : ... f2! (E1)...  f2 : ... f1? (x2)...

To a channel of communication from f1 to f2, we shall associate a port-identifier c12. Such a port-identifier may be used, in the source module f1, only in a test for emptiness or in an export-operation :

f1 : ... void(c12) → E1 ex c12 ...
(Read : ... IF c12 is void THEN E1 is exported through c12 ...)

In the destination module f2, the port-identifier c12 may be used only in a test for availability or in an inport-operation :

f2 : ... ¬void(c12) → c12 in x2 ...
(Read : ... IF c12 is full THEN c12 ports data into x2 ...)

The inport-operations correspond to tennisball-assignments : after you hand on a tennisball, you do not hold it anymore. In order to tackle concurrency in terms of serial nondeterminism, we must prevent any logical interference between concurrent modules. To this end, we shall enforce the structure (export; inport)* on the port-operations. Here are the rules for defining and using ports.

Syntax

| | | |
|---|---|---|
| Definition of c12 from f1 to f2 | : ex f1 port c12 in f2 | |
| Applications of c12 in f1 | : void (c12) | (a boolean expression) |
| | E1 ex c12 | (E1 : local expression) |
| Applications of c12 in f2 | : void (c12) | (a boolean expression) |
| | c12 in x2 | (x2 : local variable) |

Semantics
c : port-identifier
r : any predicate not containing void (c).

Axioms :
(Export) void (c) ∧ r [ c ← E ] {E ex c} ¬void (c) ∧ r
(Inport) ¬void (c) ∧ r [ x ← c ] {c in x} void (c) ∧ r

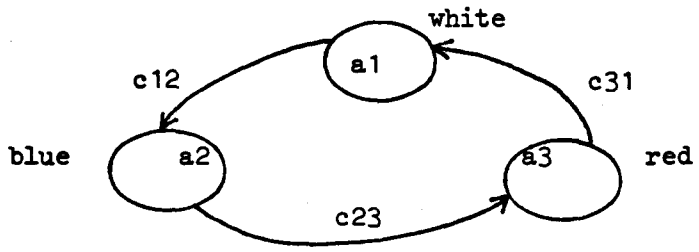
This is all we need in order to consider concurrent, communicating programs to be non-deterministic ones as in section 2. Note the property

(Transport) void (c) ∧ r [ x ← E ] {E ex c; c in x} void (c) ∧ r

Thus an assignment x2 := E2, within f2, to a local variable x2 corresponds to a transport (E2 ex c22; c22 in x2) through a port c22 local to f2, i.e. going from f2 to f2. Let us now illustrate these rules on two childish examples.

Tricolor national flags (source : E.W. Dijkstra). There are three bags, a1, a2, a3, containing pebbles which are white, red, or blue. A pebble can be transferred from a1 to a2, or from a2 to a3, or from a3 to a1, but in no other way. The goal is : all white pebbles in a1, all blue ones in a2, and all red ones in a3. The specifications are thus (i = 1...3, and 3+1 = 1)

Ports : ex f [ i ] port c [ i, i+1 ] in f [ i+1 ]
Begin : (∀b ∈ aa : blue (b) ∨ red (b) ∨ white (b))
& aa = ∪ a [ i ] & (∀i = 1...3 : void (c [ i, i+1 ]))
i
End : (∀b ∈ a1 : white (b)) & (∀b ∈ a2 : blue (b)) & (∀b ∈ a3 : red (b))
& ∪ a [ i ] = aa & (∀i = 1...3 : void (c [ i, i+1 ]))
i




We assume the availability of an indeterminate conditional assignment some b in a sth p(b); this assigns to b some element e of a such that p(e) holds, if any, and yields true if there is one and false otherwise. The formal details of the program derivation are rather obvious; the result is

$$f1 : \begin{cases} f11 : \text{void}(c12) \wedge \underline{\text{some}}\ b1\ \underline{\text{in}}\ a1\ \underline{\text{sth}}\ \underline{\text{not}}\ \text{white}(b) \\ \qquad \rightarrow (b1\ \underline{\text{ex}}\ c12;\ a1 := a1 \approx b1) \\ f12 : \neg\text{void}(c31) \rightarrow (c31\ \underline{\text{in}}\ d1;\ a1 := a1 \tilde{+} d1) \end{cases}$$

The modules f2 and f3 are obtained by symmetry, using not blue and not red instead of not white. The variables b1, d1, ..., are local.

Let us now see what happens if some other communication structure is provided. Assume for example that pebbles can be transferred between any two pairs of bags :

$$\text{Ports} : \underline{\text{ex}}\ f[i]\ \underline{\text{port}}\ c[i, i+1]\ \underline{\text{in}}\ f[i+1] \qquad i=1...3, \text{ and } 3+1 = 1$$
$$\underline{\text{ex}}\ f[i+1]\ \underline{\text{port}}\ c[i+1, i]\ \underline{\text{in}}\ f[i]$$

$$f1 \begin{cases} f11{:}\text{void}(c12) \wedge \underline{\text{some}}\ b1\ \underline{\text{in}}\ a1\ \underline{\text{sth}}\ \text{blue}(b1) \rightarrow (b1\ \underline{\text{ex}}\ c12; a1 := a1 \approx b1) \\ f12{:}\neg\text{void}(c31) \rightarrow (c31\ \underline{\text{in}}\ d1;\ a1 := a1 \tilde{+} d1) \\ f13{:}\text{void}(c13) \wedge \underline{\text{some}}\ b1\ \underline{\text{in}}\ a1\ \underline{\text{sth}}\ \text{red}(b1) \rightarrow (b1\ \underline{\text{ex}}\ c13;\ a1 := a1 \approx b1) \\ f14{:}\neg\text{void}(c21) \rightarrow (c21\ \underline{\text{in}}\ d1;\ a1 := a1 \tilde{+} d1) \end{cases}$$

and similarly for the two other modules.

Christmas cards (source : C.A.R. Hoare). There are N persons. Each one must send one card to each other one, and displays on his wall all the cards he receives. There is a port between any two persons. Thus, for i,j = 1...N, i ≠ j,

$$\text{Ports} : \underline{\text{ex}}\ f[i]\ \underline{\text{port}}\ c[i,j]\ \underline{\text{in}}\ f[j]$$
$$\text{Begin} : \forall i,j : \text{void}(c[i,j]) \ \&\ \text{drawer}[i] = \bigcup_j \text{card}[i,j]\ \&\ \text{wall}[i] = \emptyset$$
$$\text{End} : \forall i,j : \text{void}(c[i,j])^j \ \&\ \text{drawer}[i] = \emptyset\ \&\ \text{wall}[i] = \bigcup_j \text{card}[j,i]$$

$$f[i] \begin{cases} f1[i,j] : \text{drawer}[i]\ \underline{\text{has}}\ \text{card}[i,j] \rightarrow (\text{card}[i,j]\ \underline{\text{ex}}\ c[i,j]; \\ \qquad\qquad \text{drawer}[i] := \text{drawer}[i] \approx \text{card}[i,j]) \\ f2[i,j] : \neg\text{void}(c[j,i]) \rightarrow (c[j,i]\ \underline{\text{in}}\ b[i]; \\ \qquad\qquad \text{wall}[i] := \text{wall}[i] \tilde{+} b[i]) \end{cases}$$

The identifiers drawer [ i ] and wall [ i ] are set-variables local to f [ i ]. The conditions of the rules f1 [ i,j ] do not have to test void(c [ i,j ]) since $\neg\text{void}(c[i,j]) \wedge (\text{drawer}[i]\ \underline{\text{has}}\ \text{card}[i,j]) \equiv \underline{\text{false}}$.

Discussion. To express all communications in terms of such one-place, one-way ports is of course debatable. At any rate we have achieved our limited goal : the very simple properties of the port-operations permit a direct use of construction rules available for non-deterministic programs. There is no risk of global non-determinism, viz. a premature decision to wait on some condition whereas another condition is already true : the definition $ff.r \equiv \bigvee_i f_i.r$ requires to apply some $f_i$ if at least one of the conditions is true. Exclusive access to local variables could be taken care of by the transport-property : we replace rules such as $p \rightarrow x := E$ by two rules $p \wedge void(cx) \rightarrow E$ ex cx and $p \wedge \neg void(cx) \rightarrow cx$ in x, where cx is a local port-identifier uniquely associated with the local variable x.

We did not require that an export-operation and the subsequent inport-operation must coïncide in time. This is one of the reasons for the simplicity of our scheme. In case these operations should be synchronized, we can add an auxiliary export-operation in the destination and the corresponding inport in the source : then, the source halts at the communication point as long as the destination.* For implementing the tests void(c) or $\neg$void(c), auxiliary signals should be transmitted along the channel c to acknowledge receipt or to ring up on the line. This can be and has been criticized ; note that similar problems are raised when one assumes that a process may broadcast to all the others an end-signal telling its job is finished. See [ 6 ] for the mathematical semantics of a pure port-model in which synchronization is primitive.

## 4. CONCLUSION

Section 2 was devoted to the constructive derivation of certainly successful programs from possibly successful ones. The moral of the story is that we have to find successful rules of program formation on the basis of possibly successful transformation rules. In Section 3, we suggested that feasible construction rules for restricted means of expression are preferable to elaborate notations with hazardous construction rules. After all, in the tale of the wizard of Oz, Dorothy is eventually victorious thanks to her innocent use of plain water.

Programmers should concentrate their precious energy on strategic thinking rather than on tactical chores; and they should not hope in their lifetime for automatic programming by artificial intelligence. We thus favour a language for programming programs, viz. a language for describing and communicating strategies of program development, in other words a language for supporting the construction of algorithms which produce algorithms on the basis of user's specifications.

---

*P.S. : There is a neater, logical way. We insert a test void(c12) after each export-operation through c12 :

f1 : ... void(c12) → E1 ex c12; void(c12) → ...
f2 : ...¬void(c12) → c12 in b2 ...

This is related to the transport-property, and is consistent with our principle : to *separate* the condition of an operation from the associated substitution.

Acknowledgements. The work reported here has been done during 1977 in Nancy (courtesy CNRS and MBLE). B. Mayoh gave an excellent opportunity for in-depth discussions on parallelism, by organizing a seminar at Aarhus University in June. The meeting of WG 2.3 in Augustus provided an indirect (none of this was ready) but strong incentive. G. Huet frankly described the current state of the works based on [2]. The present, first version benefitted from the vivid meeting of WG 2.1 in December and from a cordial reception by the Oxford Programming Research Group.

REFERENCES

[1] C.A.R. Hoare, Communicating sequential processes, CACM, 1978.

[2] D. Knuth and P.B. Bendix, Simple word problems in universal algebras, in J. Leech (ed.), Computational Problems in Abstract Algebra, Pergamon, Oxford, 1969, pp. 263-297.

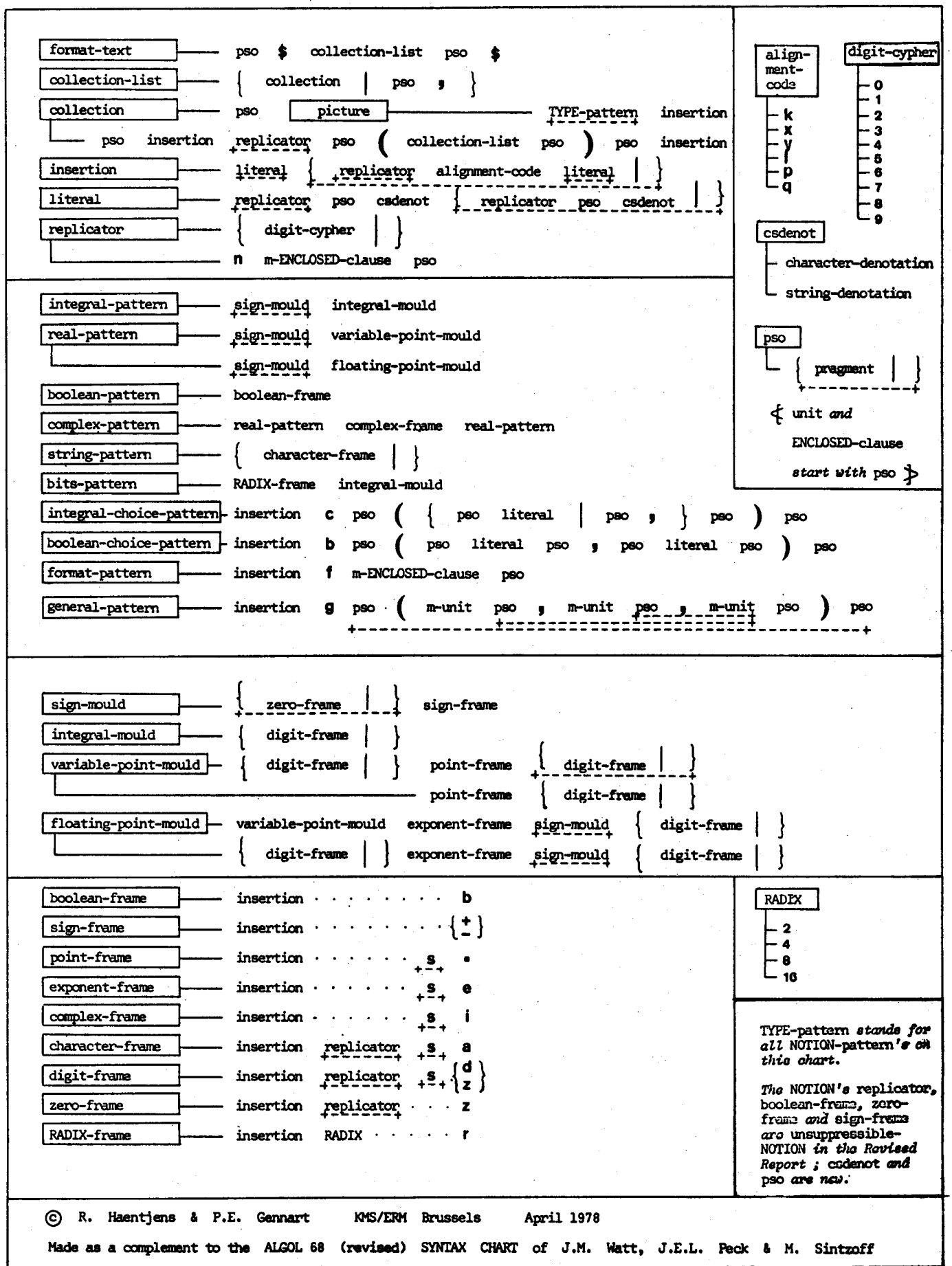
[3] M. Sintzoff, Inventing program construction rules, Proc. IFIP Conf. on Constructing Quality Software, North-Holland, 1978.

[4] J. Stoy, Denotational Semantics, M.I.T. Press, 1977.

[5] J.W. de Bakker, The fixed point approach in semantics : theory and applications, in J.W. de Bakker (ed.), Foundations of Computer Science, Mathematisch Centrum, Amsterdam, 1975, pp. 3-56.

[6] G. Milne and R. Milner, Concurrent processes and their syntax, CSR-2-77, Dept of Computer Science, Univ. of Edinburgh, 1977.

# ALGOL 68 (revised) format-text SYNTAX CHART






Made as a complement to the ALGOL 68 (revised) SYNTAX CHART of J.M. Watt, J.E.L. Peck & M. Sintzoff