ISSN 0084-6198

Algol Bulletin no. 41

JULY 1977

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 Professor J.E.L. Peck, Vancouver).

The following statement appears here at the request of the Council of 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 control 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 completely 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,
England.

Back numbers, when available, will be sent at $3 each. However, it is regretted that only AB32, AB34, AB35, AB37, AB38 and AB39 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.

AB41.0 EDITOR'S NOTES

Again, I have to apologise for the long delay since the last issue, and again the reason has been lack of contributions. Again also, the remedy lies in your hands, dear readers. As it turns out, due to various items turning up at the last minute, we have been able partially to make up for the delay by producing a rather thicker issue than we have had recently.

Although the contents of this issue are fairly typical of the sort of material we like to publish, there is one departure from the norm in the form of a rather substantial algorithm. Although I have no desire to compete with CACM, and other Journals which publish algorithms regularly, I think that there could well be a place in the ALGOL Bulletin for Algorithms of a specialized nature, such as those concerned with program handling (e.g. compiling, editing, formatting, etc.) or those which illustrate novel, or particularly neat, programming methods.

So, please send your contributions. Algorithms may be in ALGOL 60 (preferably ALGOL 60M) or in ALGOL 68, and the customary rules (notably those requiring evidence that the algorithm actually works) will apply. Of course, comments and certifications will also be welcomed in due course.

Now for some good news. Three issues ago, I had to increase the cost of the AB from $5 to $7 per three issues. At that time, I was finding it difficult to predict what future costs would be and there was very little fat in hand. Now things are much better, a good cash balance has been accumulated, and I therefore feel justified in declaring this to be a free issue. In other words, all those subscribers entitled to receive this issue (AB41) will automatically have their subscription extended by one, so that they will eventually receive four issues for their $7.

AB41.1 Announcements

AB41.1.1 ALGOL 60M

After it had gone to press, some serious misprints were discovered in the Supplement to the ALGOL 60 Revised Report (Comp. Jour. 19 3 Aug. 1976), and errata to correct these appeared, together with the full Modified Report, in Comp. Jour. 19 4 Nov. 1976. The errata are also reproduced at the end of the Report and those who have copies of that edition of the Supplement are invited to elaborate them without delay.

The full Supplement has now been published again (with those errata incorporated and hopefully with no new ones introduced) in SIGPLAN Notices.

AB41.1.2 Conference Proceedings: New Directions in Algorithmic Languages - 1976

The papers and discussions at the 1976 meeting of Working Group 2.1 at St. Pierre de Chartreuse have been edited by Steve Schuman in a similar format to last year's proceedings, and all ALGOL Bulletin subscribers should have had a copy by now. Additional copies may be obtained, so long as stocks last, from Stephen A. Schuman, IRIA Laboria, BP 5 - Rocquencourt, 78150 LE CHESNAY, France.

AB41.1.3 Conference Proceedings: 4th International Conference on the Design and Implementation of Algorithmic Languages.

The proceedings of this Conference, held at New York on June 14th - 16th 1976 (see AB39.1.4) may be obtained, for U.S. $12.00 from:

Miss Lenora Green,
Courant Institute,
251 Mercer Street,
New York, NY 10012,
U.S.A.

(Cheques to be made payable to New York University).

AB41.1.4 ALGOL 68

There are a few small misprints in the Acta Informatica edition of the Revised Report, and the relevant errata will be found on the last page of this AB. Please elaborate them without delay. Note that the TR74-3 edition is unaffected. Note that only very minor misprints are being corrected at this stage. There are in addition various bugs that have been found in the Revised Report, but no action is being taken on these until the Support Subcommittee has considered them separately (see the following item, describing the procedure that is to be followed.)

In the meantime, the complete Revised Report (with these misprints corrected) has been published in SIGPLAN Notices May 1977. Also, in that same issue of SIGPLAN Notices are "A Sublanguage of ALGOL 68" by P.G. Hibbard (the defining document of ALGOL 68S), there published for the first time and the Report on the Standard Hardware Representation, (originally published in AB40.5). Thus all three Reports on ALGOL 68, as approved by the Working Group and by IFIP, are now available in one volume. Reprints from ACM.

AB41.1.5 The Standing Subcommittee on Algol 68 Support Treatment of Questions asked about the Revised Report.

The process for answering questions about the Revised Report and other associated documents:

1. A question is first posed to the Subcommittee by sending a letter to the convener of the Subcommittee either to request interpretation or to report an alleged error, inconsistency or typographical misprint in the Revised Report on the Algorithmic Language Algol 68 or associated documents. The letter which becomes the property of Working Group 2.1 should include a self-referencing publication release because private communications are automatically subject to copyright by international convention.
2. The question should arrive sufficiently in advance of a meeting to allow appropriate distribution to the subcommittee before the meeting in order to be considered at that meeting.
3. During the meeting following receipt and distribution of the letter by the convener, the question will be scheduled for discussion, voting and action. The possible actions which can be taken include:

a Decide no problem is raised by the question, or that the problem raised by the question had already been discussed and resolve.

b Decide that a trivial problem exists raised by the question.

For possibilities a and b the convener will refer the letter to a member of the Subcommittee who will write an appropriate response.

c Decide that the question raises a problem.

The convener will appoint a taskforce to examine the question and report back. The taskforce has the responsibility to produce a written report which will explicate the problem as well as possible actions for its resolution. This should be accompanied by a statement of the relative advantages and disadvantages of each action.

In any case, a file of all questions and their answers will be maintained by the convener for the future use of the Subcommittee. At a future time an editing taskforce may be asked to compile a publishable document containing the essence of the file.

The current convener of the Standing Subcommittee on Algol 68 Support, R. Uzgalis, can be contacted at: University of California, Los Angeles, Computer Science Department, Boelter Hall 3731H, Los Angeles, California, 90024.

AB41.1.6 Informal Introduction - Revised Edition

The Revised Edition of the Informal Introduction to ALGOL 68, by C.H. Lindsey and S.G. van der Meulen has now been published by North-Holland at a price of Dfl. 35.00/US $14.50 Paperback (ISBN 0-7204-0726-5) or Dfl. 70.00/US $28.75 Hardback (ISNB 0-7204-0504-1). It may be obtained from booksellers or direct from North-Holland Publishing Company, P.O. Box 211, Amsterdam, The Netherlands. In the U.S.A. and Canada it is distributed by Elsevier North-Holland, Inc., 52 Vanderbilt Ave., New York, N.Y. 10017, and in Australia by Dutch-Australian Book Depot, 11-13 Station Street, Mitcham, Vic. 3132. Please be sure that you ask for, and obtain, the Revised Edition, which comes in a garish red cover to distinguish it from the lurid green of the first edition.

This Revised Edition is the volume referred to in 0.3 of the Revised Report. Although it still follows the same general plan as the original edition, it has been brought fully into line with the Revised Report. As before, it aims to describe the whole of the language and may thus be used both as a work of reference and as a text book (although it does not claim to be suitable as a primer for novices). Appendices have been added on ALGOL 68S and on the Standard Hardware Representation.

AB41.1.7 Report in BRAILLE

After the publication of the Revised Report on ALGOL 68 in 1975, the Mathematical Centre has undertaken the task of producing a braille version of the ALGOL 68 Report. This braille version is based on a copy of the tape from which the Revised Report has been typeset. Except for Winnie the Pooh and a few other pictures, the complete Report has been converted.
Any information on this braille version can be obtained from:

J.C. van Vliet,
Mathematical Centre,
2e Boerhaavestraat 49,
AMSTERDAM,
The Netherlands.

Copies of the Report will be made available in the form of either a large box of brailed paper or an (IBM-compatible) magnetic tape containing a one-to-one representation of the braille version.
The price will be limited to the cost of reproduction.

AB41.2 Letters to the Editor

AB41.2.1 Hardware Representation

The Editor,
ALGOL Bulletin.

Dear Sir,

I suggest a new formulation of section C.1 of the Report on the Standard Hardware Representation for ALGOL 68 (AB40.5 and SIGPLAN Notices May 1977), as follows:

The standard is defined in terms of worthy characters in order that program conversion will require only a transliteration of character codes.

The transliteration may be done automatically if each implementer provides the following:

1) Each translator has a single input representation corresponding to the standard. However, a program may be represented in another code than this input representation. A mapping program is needed which maps the program to the input representation. There are many mapping programs.

2) A portable program should provide a "cap" before the codes of the program in the porting file.

The structure of the cap

The file contains codes represented by some fixed number of bits. Each character (worthy character or helping character, see below) is represented by a word containing such a fixed length code. There are helping characters: disjunctor, new line, end of text and end of input. The cap is the sequence of representing words for the characters:disjunctor, new line, all of the worthy characters in the order in which they appear in *1, end of text, end of input, the upper-case national letters followed by a disjunctor (or only a disjunctor), the lower-case letters followed by a disjunctor or the lower-case letters and lower-case national letters followed by a disjunctor (if there were upper-case national letters) or only a disjunctor.

3) There is a general mapping program such that it may do the transliteration of new codes given the cap and an integer (the number of bits in each representing code).

(The cap is not a universal method of portability, but it is a satisfactory method in many cases.)

Yours faithfully,

A.N. Maslov
Department of Algorithmic Languages,
Faculty of Computer Science,
Moscow State University,
Moscow. U.S.S.R.

AB41.4.1 The Algollers

by R. de Morgan. (Reprinted from the Newsletter of the BCS ALGOL Group).

A long, long time ago (about eighteen years, give or take a furlong), several wise men sat down and designed a programming language. Being of a somewhat adventurous nature, they produced a somewhat adventurous language; indeed, so adventurous was this language that people debate to this day the properties of this wondrous language and others that owe some of their origins to it. It was called "Algol 60", but didn't seem to have any features of specific use to astronomers. They revised it a bit in 1962, but unlike later languages, did not update its number; indeed, most people were quite content to call it simply "Algol", and some of them spelt it with capital letters.

Algol had a wealth of features. Some indeed were quite extraordinary and could be used to perform wonderful feats of computation in mystical ways (the way it could find prime numbers with a single statement seemed to smack of witchcraft). Some of the features were left to the imagination and ingenuity of the implementers, resulting in a wealth of dialects of the language. Machine dependent features such as input-output were skilfully avoided so as to avoid contamination of programs. Nevertheless, implementers seemed to think that this was a desirable addition, and added input-output systems of every conceivable shape and size.

While the outside world were marvelling at the wonders of Algol 60, the wise men were busily at work designing its successor. They spoke of it as "Algol X", and there was even talk of an "Algol Y", but when it saw the light of day, it was called Algol 68. Here indeed was a magnificent language - it had a bigger, better Report, parts of which were written in a curious form called a W-grammar, and seemed to require many type fonts, not to mention italic full stops. "Why didn't they use BNF?" was the cry. Fortunately, someone pointed out that if one read the examples at the back of the report, it all became clear.

Meanwhile, halfway up a hill in darkest Worcestershire, at a Very Secret Place, Scientific Civil Servants were labouring night and day to produce the very first Algol 68 implementation. This was known as Algol 68R and became very famous. Following this, many other implementation sprang up, but implementers had great difficulties with some of the features, and various subsets were born.

But the Algol 60 devotees had not been idle. Meeting at secret locations in the English countryside, they set out to eliminate the dreaded Remaining Trouble Spots. They called their Algol 60 "Modified" (they did not like to call it Algol 76 for fear that the Algol 68 authors would become angry with them for having a higher number), and they even included a simple input-output system. They produced a Report, as was the custom, and published it in a Learned Journal.

Both the Algol 60 and Algol 68 devotees were members of a Secret Society, which was called the Algol Association. They would come from far and wide to listen to the wisdom and lore imparted by famous Algol mystics. They also communicated with each other by means of a Bulletin, speaking both in words and algorithms. Although there was some amount of rivalry between the Algol 60 and Algol 68 factions, they were united in their scorn of other societies such as the Cobolers and the Fortranners. These societies spoke strange tongues which were most un-Algol-like.

There had grown up a movement called Structured Programming, and the Algol devotees found that they could write structured programs without much difficulty. Indeed, by using Algol 68 they found that they could do away altogether with the hateful labels that many said spoiled the beauty of their languages. The Cobollers and Fortranners were very jealous of this, and tried to write structured programs of their own. The Algollers saw that this was futile and laughed them to scorn saying "How can they expect to write structured programs with such foolish languages?". But the people of the world were much confused by all this talk, and did not know which way to turn. Most of them were very conservative by nature and said "Why should we use these new languages that these mystics invent? Let us instead use the languages that our forefathers have always used." And so they went their way, and performed their Sorts and Merges, and entered Subroutines, and did other mundane things; for such was the way of the world.

AB41.4.2 On the ALGOL 68 Transput Conversion Routines

by

J.C. van Vliet (Mathematisch Centrum, Amsterdam).

ABSTRACT

In section 10.3.2.1. of the Revised Report on the Algorithmic Language ALGOL 68, a set of routines is given for the conversion of numerical values to strings and vice versa. If this set is used as an implementation model, the way in which the numerical aspects are dealt with causes considerable trouble. A new version of these routines is given in which numbers are first converted to a string of sufficient length, after which all arithmetic is performed on this string. In this way, for each direction only one place remains where real arithmetic comes in.

INTRODUCTION

In section 10.3.2.1. of the Revised Report on the Algorithmic Language ALGOL 68 [1] (in the sequel referred to as the Report), a set of routines is given for the conversion of numerical values to strings and vice versa. Compared with most other sections of the Report, this one seems to have received little attention from the editors.

This section may be looked upon from two different points of view: one may take it either as a definition of the intention of the conversion, or as some kind of implementation model. In any case, the following remark from section 10.1.3. of the Report applies:

> "Step 8: If, in any form, as possibly modified or made in the steps above, a <u>routine-text</u> occurs whose calling involves the manipulation of real numbers, then this <u>routine-text</u> may be replaced by any other <u>routine-text</u> whose calling has approximately the same effect;"

Taking the former point of view, one might wonder whether the intention is best described by a set of ALGOL-68 routines. (In that case, one should at least add an extensive description in some natural language too. For example, it took me quite some time to discover when exactly *undefined* is called. It seems to have been the intention to call *undefined* only when it is obvious that no string may be delivered satisfying the constraints set by the parameters, as in the case *fixed(x, 3, 4)*. However, when x and i are of the mode <u>real</u> and <u>int</u>, respectively, *whole(x, 1)* calls *undefined*, while *whole(i, 1)* does not.)

Using the routines as an implementation model, the remark from section 10.1.3. that is cited above will have to be invoked heavily. To give an example, it is impossible to print *L max real* by means of the routine *fixed* from the Report, because of the statement

$$\underline{L}\ \underline{real}\ y := x + \underline{L}\ .5 * \underline{L}\ .1 \uparrow after;,$$

which is used for rounding. Adding one half of the last decimal that is asked for excludes a whole class of numbers in the vicinity of *L max real* from conversion! Also, y may well be equal to x after execution of this statement if the number that is being added is relatively small compared to x; so the result is truncated rather than rounded.

The errors found in the section on conversion routines in the Report, are listed below. The problems caused by the way in which the numerical aspects are dealt with (overflow, accuracy) are also discussed. Next, a version of the routines is given which bypasses these numerical problems. Here, numbers are first converted to strings of sufficient length, after which all arithmetic is performed on these strings. This version may really be seen as an implementation model: for each direction of conversion, there is only one place where real arithmetic comes in.

The Control Data ALGOL 68 implementation [2] has been of great help in testing both the routines from the Report and the ones given below. Numerous talks with H. Boom, D. Grune and L. Meertens have contributed considerably to the polished form of the various routines.

When we try to use the routines from the Report as they are, the following numerical problems arise (apart from the one already mentioned in the introduction):

- The statement in *fixed*:

  <u>*while*</u> *y* + <u>*L*</u> *.5* * <u>*L*</u> *.1* ↑ *after* ≥ <u>*L*</u> *10* ↑ *length* <u>*do*</u> *length* +:= *1* <u>*od;*</u>

  assumes that integers may take on the same values as reals, for <u>*L*</u> *10* ↑ *length* has mode <u>L</u> <u>int</u>. This may well not be the case, thus yielding an integer overflow. Presumably, the intention has been to write <u>*L*</u> *10.0* ↑ *length*.

  Notice however that the left-hand side of the boolean expression may still cause a real overflow if *y* is approximately equal to *L max real*.

- The statement in *subfixed*:

  while *y* ≥ <u>*L*</u> *10.0* ↑ *before* <u>*do*</u> *before* +:= *1* <u>*od;*</u>

  may cause an overflow if *y* and *L max real* are of the same order of magnitude. One could write something like

  <u>*while*</u> *y* / <u>*L*</u> *10.0* ≥ <u>*L*</u> *10.0* ↑ *(before* − *1)* <u>*do*</u> *before* +:= *1* <u>*od;*</u>,

  but then the next statement will cause the overflow. One may combine the two statements as follows:

  <u>*while*</u> *y* ≥ <u>*L*</u> *1.0* <u>*do*</u> *y* /:= <u>*L*</u> *10.0;* *before* +:= *1* <u>*od;*</u>

  If, however, division is not too accurate, the repeated division may cause large numbers to be converted much less accurately than small numbers.

ANOTHER SET OF CONVERSION ROUTINES

The main differences between the set of conversion routines presented below and the set in section 10.3.2.1. of the Report are the following:

- numbers are converted to strings of sufficient length, after which the rounding is performed on the strings. This seems to be the only reasonable way to ensure that numbers like *L max real* may be converted using *fixed* or *float*. (One must be careful when rounding causes a carry out of the leftmost digit. For example, in *float* this will cause the decimal point to shift. This will in turn yield a new exponent which, after conversion, may need more (or less!) space.)
- the routines *fixed* and *float* are written non-recursively.
- no use has been made of the routine *L standardize*. In general, I have tried to minimize the number of places where real arithmetic comes in. Only (part of) the routine *subfixed*, and a few lines in *string to L real* use real arithmetic and may therefore have to be rewritten for a specific machine.

Care has been taken that *whole*, *fixed* and *float* behave exactly as the corresponding routines from the Report are intended to. However, as has already been discussed briefly in the introduction, it is difficult to see exactly when *undefined* is called. Therefore, I have decided to call *undefined* in all cases where *error characters* are returned.

The (hidden) routines *subwhole* and *subfixed* behave slightly differently from their namesakes in the Report. In particular, *error characters* are never delivered. Together with the removal of *L standardize*, this necessitates some changes in the editing of integers and reals in the routine *putf* in section 10.3.5.1. of the Report.

Conversion by means of *whole*.

The routine *whole* is intended to convert integer values. It has two parameters:

- *v*, the value to be converted, and
- *width*, whose absolute value specifies the length of the string that is produced.

Leading zeros are replaced by spaces and a sign is normally included. The user may specify that a sign is to be included only for negative values by specifying a negative or zero width. If the width specified is zero, then the shortest possible string is returned.

The routine *whole* proceeds as follows: First, using *subwhole*, a string *s* is built up containing all significant digits and possibly the sign of the number being converted. If the user has specified a width of zero, this string *s* is delivered as a result. Otherwise, the length of *s* should not be greater than the absolute value of the specified width. If it is, *undefined* is called and *error characters* are returned; if not, spaces are added in front of *s* if necessary, and the resulting string is delivered.

Examples:

whole(i, -4) might yield *"...0"*, *"..99"*, *".-99"*, *"9999"*, or, if *i* were greater than *9999*, *"****"*, where *"*"* is the yield of *errorchar*;

whole(i, 4) would yield *".+99"* rather than *"..99"*;

whole(i, 0) might yield *"0"*, *"99"*, *"-99"*, *"9999"* or *"99999"*.

```
proc whole = (number v, int width) string:
  case v in
    ⟨(L int x):
       (bool neg; string s:= subwhole(x, neg);
       (neg | "-" |: width > 0 | "+" | "") plusto s;
       if width = 0 then s
       elif int n = abs width - upb s; n ≥ 0
       then n * "." + s
       else undefined; abs width * errorchar
       fi)⟩,
    ⟨(L real x): fixed(x, width, 0)⟩
  esac;

proc ? subwhole = (L int x, ref bool neg) string:
  begin string s:= "", L int n:= abs x; neg:= x < L 0;
    while dig char(S (n mod L 10)) plusto s;
      n overab L 10; n ≠ L 0
    do skip od;
    s
  end;
```

Conversion by means of *fixed*.

The routine *fixed* is intended to convert real values to fixed point form (i.e., without an exponent). It has an *after* parameter to specify the number of digits required after the decimal point. The other parameters have the same meaning as those for *whole*.

From the value of the *width* and *after* parameter, the amount of space left in front of the decimal point may be calculated. (The values of the *after* and *width* parameter should be such that at least some number may be converted according to the format they specify. If this is not possible, *undefined* is called and *error characters* are returned.) If the space left in front of the decimal point is not enough to contain the integral part of the number being converted, digits after the decimal point are sacrificed. If the number of digits after the decimal point is reduced to zero and the number still does not fit, *undefined* is called and *error characters* are returned.

Implementation of the simple algorithm described above involved some nasty problems. Therefore, the comprehensive description of the new version of the routine *fixed* which follows is supplied with various examples to illustrate the places where great care is needed to maintain correctness. The routine proceeds as follows: If the value of the *after* parameter is less than zero, *undefined* is called immediately, and *error characters* are returned. Otherwise, using *subfixed*, an unrounded string *s* is built up, containing all significant digits before the decimal point, and *after+1* digits after the decimal point. As a side-effect, the variable *point* points to the digit after which the decimal point has to be inserted, while the boolean variable *neg* indicates the sign of the value submitted (*neg* ⇒ $v < 0$). Thus, for example,

s:= *subfixed*(3.13, 3, *point*, *neg*, <u>false</u>) ⇒ *s* = "31300" & *point* = 1,
s:= *subfixed*(0.75, 1, *point*, *neg*, <u>false</u>) ⇒ *s* = "75" & *point* = 0.

In both cases, *neg* gets the value *false*. Then, a value *w* is calculated indicating the number of positions available for digits <u>and</u> the decimal point. For example,

fixed(3.13, 10, 3) ⇒ $w = 9$,
fixed(0.75, 0, 1) ⇒ $w = 0$,
fixed(0.75, 2, 1) ⇒ $w = 1$.

In the last example, *undefined* will be called, because no number can be converted according to this format (the two positions specified are swallowed by the sign and the decimal point, so no space remains for the one digit specified after the decimal point). (Obviously, in case the value of the *width* parameter is zero, *undefined* will not be called.)
Subsequently, two cases are distinguished:

- The user specified a width of zero, i.e., the shortest possible string containing *after* digits after the decimal point has to be delivered. In this case the string is simply rounded starting from the last element. If this rounding causes a carry out of the leftmost digit, the decimal point has to be inserted one place further to the right (*fixed(0.95, 0, 1)* leads to *s* = *"95"* & *point* = *0* via *subfixed*, and *s* = *"10"* & *point* = *1* via *round*, ultimately resulting in the string *"1.0"* to be delivered);
- The user specified a non-zero width. Then, the number *digits* is calculated: the number of positions available for digits. This number obviously is either *w* - *1* or *w*: either a decimal point is to be delivered, or it is not. A decimal point will <u>not</u> be delivered if *after* = *0*, or if the decimal point just falls outside the available number of positions *w*. (Note that the case *after* = *0* does not present any problem and may safely be ignored.) Otherwise, the decimal point has to be inserted somewhere, so *digits* = *w* - *1*. (Note furthermore that if the room available for digits is not even sufficient to contain all digits of the integral part (i.e., *point* > *w*), a call of *undefined* will ultimately result.)
  The next step will be to round the string. Again, if the number of positions available for digits is greater than the number of digits to be delivered, the string is simply rounded starting from the last element. If this causes a carry out of the leftmost digit, the decimal point has to be inserted one place further to the right, and the longer string is delivered. Otherwise, the string is rounded starting from the digit at position *digits* + *1*. If this rounding causes a carry, the string has to be snipped at the position indicated by *digits*, except when the decimal point is now left just after position *w*. (This tricky case occurs, e.g., at the call *fixed(99.7, -3, 1)*. Following the flow of control, we see that *digits* = *2*, so a call *round(2, "9970")* results, which yields

true & *s* = *"100"*. As, however, the decimal point just shifted out of the available number of positions (3), the whole string can be returned.)

We are now left with a string *s* containing all significant digits to be delivered. If there is space for at least one more digit, and the decimal point is at the extreme left, *"0"* is added at the front end, thus delivering *"0.35"* rather than *"␣.35"* (and *"0"* rather than *"␣"* in a case like *fixed(0.3, -1, 0)*!).
As a last step, *undefined* is called and *error characters* are delivered if the room available for digits is not sufficient to contain all digits of the integral part of the value submitted, or the *after* and *width* parameters are such that no number may be converted using that format. In all other cases, a sign is added if necessary, and a decimal point may be inserted. If the specified width is non-zero, the remaining positions are filled with spaces. The resulting string is delivered.

Examples:

fixed(x, -6, 3) might yield *"␣2.718"*, *"27.183"*, *"271.83"* (one place after the decimal point has been sacrificed in order to fit the number in), *"2718.3"*, *"␣27183"* or *"271833"* (in the last two examples, all positions after the decimal point are sacrificed);

fixed(x, 0, 3) might yield *"2.718"*, *"27.183"* or *"271.828"*.

```
proc fixed = (number v, int width, after) string:
  if after < 0
  then undefined; abs width * errorchar
  else int point, bool neg;
    string s:= subfixed(v, after, point, neg, false);
    int w = abs width - (neg ∨ width > 0 | 1 | 0);
    if width = 0
    then (round(upb s - 1, s) | point +:= 1)
    else int digits = (w = point | w | w - 1);
      if digits > upb s - 1
      then (round(upb s - 1, s) | point +:= 1)
      else (round(digits, s) | point +:= 1; (point ≠ w | s:= s[:digits]))
      fi
    fi;
    (point = 0 ∧ (s = "" ∨ w - 1 > upb s) | "0" plusto s; point:= 1);
```

```
    if upb s < point ∨ (after ≥ w ∧ width ≠ 0)
    then undefined; abs width * errorchar
    else s:= (neg | "-" |: width > 0 | "+" | "") +
             (point = upb s | s | s[:point] + "." + s[point + 1:]);
      (width = 0 | s | (abs width - upb s) * "." + s)
    fi
  fi;
```

Notice that the above routine does not distinguish variable-length numbers; they are just passed down to *subfixed*. The same will hold for the routine *float* given below.

The routine *subfixed* performs the actual conversion from numbers to strings, and may be called from either *fixed* or *float*. When called from *fixed*, it has to return a string containing all digits from the integral part of the value submitted, and *after* + *1* digits from the fractional part. When called from *float*, it has to return a string containing the first *after* + *1* significant digits. In both cases, the last digit is truncated, and not rounded. (The rounding is done later on, and rounding the number twice may cause something like *9.46* to be converted to *"10.0"*.) Considering this string as a number, the value of the parameter *p* will be the shift of the decimal point from the first digit. The parameter *neg* will indicate the sign of the value submitted (true iff negative).

It goes without saying that the routine *subfixed* must be completely accurate: it will be used to measure the accuracy of numerical algorithms, and we want to be sure that that is really what is measured, and not the accuracy of the conversion. It is therefore impossible to give an ALGOL-68 routine that will do. Instead, we give the following semantic definition:

It is a unit which, given a value V, yields a value S and makes *p* and *neg* refer to values P and B, respectively, such that:

- B is true if V is negative, and false otherwise;
- it maximizes

$$M = \sum_{i = \underline{\text{lwb}}\, S}^{\underline{\text{upb}}\, S} c_i * {}_{10}^{P - i}$$

under the following constraints:

- lwb S = 1;
- upb S = P + after + 1 if floating is false, and after + 1 otherwise;
- for all i from lwb S to upb S:

  $0 \le c_i \le 9$, where c_i = char dig(S[i]);
- M ≤ |V|.

(If one wants to circumvent the need to know the storage allocation techniques used by the compiler (which is needed to build the string), one may construct an embedding like:

```
proc ? subfixed = (number v, int after, ref int p, ref bool neg, bool floating)
        string:
  begin int size; guess storage(v, after, size, floating);
        # size:= some sufficiently large integer, an upperbound for
          the number of digits that will result #
    [1 : size] char s;
    do subfixed(v, after, p, neg, floating, size, s);
        # the actual conversion; the characters are placed in s.
          As a side-effect, size indicates the number of digits placed
          in s #
    s[ : size]
  end;
).
```

The (hidden) routine *round* is used for rounding. The parameter *s* refers to the string that will be rounded, the parameter *k* refers to the last element of *s* that will be returned. The routine yields true if the rounding causes a carry out of the leftmost digit.

```
proc ? round = (int k, ref string s) bool:
  if bool carry:= char dig(s[k + 1]) ≥ 5; s:= s[ : k]; carry
  then
    for j from k by -1 to 1 while carry
    do int d = char dig(s[j]) + 1; carry:= d = 10;
      s[j]:= (carry | "0" | dig char(d))
```

```
    od;
    (carry | "1" plusto s); carry
  else false
  fi;
```

Conversion by means of *float*.

The routine *float* is intended to convert real values into floating point form. It has an *exp* parameter to specify the width of the exponent. Just as in the case of the *width* parameter, the sign of the *exp* parameter specifies whether or not a plus-sign is to be included. (This possibility is not mentioned too clearly in the Report.) If the value of the *exp* parameter is zero, *float* acts as if minus one were specified, i.e., the exponent is converted to a string of minimal length. (Again, this possibility is not mentioned clearly in the Report. Moreover, it contradicts Fisker's remark on page 3.4 of his thesis [3], where it is stated that in this case *float* acts as if the value of the *exp* parameter were one! This seems to be a mistake.) The other parameters are the same as those for the routine *fixed*. (However, the value of the *width* parameter may obviously not be zero.)

The routine *float* proceeds as follows: From the values of *width*, *after* and *exp*, it follows how much space is left in front of the decimal point (assuming no sign will be delivered). Then *subfixed* is called, which returns a string *s* containing a sufficient number of significant digits. As a side effect, *exponent* gets the value of the exponent, assuming the decimal point to be just in front of the first digit while *neg* gets to indicate the sign of the number. For example,

```
s:= subfixed(321.073, 4, exponent, neg, true) ⇒ s = "32107" & exponent = 3,
s:= subfixed(.004379, 4, exponent, neg, true) ⇒ s = "43790" & exponent = -2.
```

We now adjust *before* if a sign is to be delivered.
The number is then (conceptually) standardized, yielding the real exponent. This exponent now has to fit in a string *expart*, whose length is bounded by the width specified by the *exp* parameter. If this is not possible, the digits after the decimal point are sacrificed one by one; if there are no more digits left after the decimal point and the exponent still does not fit, digits in front of the decimal point are sacrificed too. Note that this has repercussions on the value of the exponent (and thus possibly on the width

of the exponent). More precisely, this process goes as follows: Let *before* and *aft* denote the number of digits before and after the decimal point, respectively. Let *expspace* be the width allowed for the exponent. If the exponent does not fit (**upb** *expart* > *expspace*), then one of the following happens:

i) If there are still digits after the decimal point to be given in (*aft* > *0*), then *aft* -:= *1*. If, however, as a result of this, *aft* = *0*, we threaten to deliver something like *3.e+5*, so the decimal point has to be left out too, which gives us one digit extra in front of the decimal point, so

before +:= *1*; *exponent* -:= *1*.

ii) If there are no digits left after the decimal point, digits in front of the decimal point are given in, so

before -:= *1*; *exponent* +:= *1*.

In either case, one position extra is assigned to the exponent, so *expspace* +:= *1*. This shuffling will end, and then the string is rounded. If this rounding causes a carry out of the leftmost digit, the exponent must be increased, which may cause some more shuffling. During this process, we have to check at each step whether all digits have been consumed (**sign** *before* + **sign** *aft* ≤ *0*, which also caters for wrong input parameters). In that case, *undefined* is called and *error characters* are delivered. Otherwise, the various parts are glued together and the resulting string is delivered.

Examples:

float(x, 9, 3, 2) might yield "-2.718$_{10}$+0", "+2.72$_{10}$+11" (one place after the decimal point has been sacrificed in order to make room for the exponent);

float(x, 6, 1, 0) might yield "-256$_{10}$1", "+26$_{10}$12" or "+1$_{10}$-9" (in case *x* has the value *0.996$_{10}$-9*).

```
proc float = (number v, int width, after, exp) string:
  begin int before := abs width - (after ≠ 0 | after + 1 | 0) - (abs exp + 1),
          exponent, aft := after, expspace := abs exp;
    bool neg, rounded := false, possible := true;
    string s := subfixed(v, before + after, exponent, neg, true), expart := "";
    (neg ∨ width > 0 | before -:= 1); exponent -:= before;
    while expart := (exponent < 0 | "-" |: exp > 0 | "+" | "") +
                  subwhole(abs exponent, loc bool);
```

```
    if sign before + sign aft ≤ 0
    then possible:= false
    elif upb expart > expspace
    then expspace +:= 1;
      (aft > 0 | aft -:= 1;
        (aft = 0 | before +:= 1; exponent -:= 1)
      | before -:= 1; exponent +:= 1); true
    elif rounded then false
    elif round(before + aft, s)
    then exponent +:= 1; rounded:= true
    else false
    fi
  do skip od;
  if ¬ possible then undefined; abs width * errorchar
  else (neg | "-" |: width > 0 | "+" | "") + s [: before] +
       (aft = 0 | "" | "." + s[before + 1 : before + aft]) +
       "₁₀" + (expspace - upb expart) * "0" + expart
  fi
end;
```

Conversion of strings to numbers.

The routine *string to L int* from section 10.3.2.1. of the Report works fine, so we will not pay any attention to it. Although the routine *string to L real* looks reasonable, it uses *L standardize*, and a new version of it is given below. The routine needs real arithmetic, and thus must be rewritten on most machines. The version given here is merely an outline of how things might be done.

The routine *string to L real* is hidden from the user. Therefore we may safely assume that the layout of the string supplied is correct. The first element of the string contains the sign of the number. Furthermore, the string may contain a decimal point, and it may contain an exponent.

The routine proceeds as follows: First, we search for the exponent part, the beginning of which is indicated by "*e*", and the decimal point ".". If there is an exponent part, it is converted using *string to int*, yielding an exponent *expart*. If the conversion of the exponent is unsuccessful, *string to L real* returns false, indicating unsuccessful conversion too. Otherwise, the first significant digit is sought, pointed to by *j*. The exponent *expart* is now adjusted so that it yields the exponent of the number assuming the decimal point to be just after the first significant digit. *L max real*, being the largest value that may result from the conversion, is adjusted in the same way, yielding a value *max* and an exponent *max exp*. Of course, conversion is unsuccessful if *expart* > *max exp*. Subsequently, the first *L real width* significant digits are converted. (Note that any further digits would not affect the value.) At each step of this conversion, we have to cater for the case where *expart* = *max exp*; for then, the next digit of *max* and the one from the string have to be compared to see whether conversion may still continue. As a last step, if conversion has been successful, the resulting number is (supplied with the correct sign) assigned to the parameter *r*. The routine yields true if the conversion has been successful, and false otherwise.

```
proc ? string to L real = (string s, ref L real r) bool:
  begin int e:= upb s + 1; char in string("e", e, s);
    int p:= e; char in string(".", p, s); int expart:= 0;
    bool safe:= (e < upb s | string to int(s[e + 1 : ], 10, expart) | true);
    if safe
    then int j:= 1;
      for i from 2 to e - 1
      while s[i] = "0" ∨ s[i] = "." ∨ s[i] = "."
      do j:= i od;
      expart +:= p - 2 - j;
      L real x:= L 0, max:= L max real, int length:= 0, max exp:= 0;
      while max / L 10.0 ↑ max exp ≥ L 10.0 do max exp +:= 1 od;
      (expart > max exp | safe:= false);
      for i from j + 1 to e - 1 while length < L real width ∧ safe
      do
        if s[i] = "." then skip
        elif int si = char dig(s[i]); length +:= 1; expart = max exp
        then int d = S entier (max / L 10.0 ↑ max exp);
          (si > d | safe:= false | x +:= K si * L 10.0 ↑ expart);
          max -:= K d * L 10.0 ↑ max exp; max exp:= expart -:= 1
        else x +:= K si * L 10.0 ↑ expart; expart -:= 1
        fi
```

```
      od;
      (safe | r:= (s[1] = "+" | x | -x))
    fi;
    safe
  end;
```

REFERENCES

[1] WIJNGAARDEN, A. VAN, et al (eds.), *Revised Report on the Algorithmic Language* ALGOL 68, Acta Informatica 5 (1975) 1-236.

[2] ALGOL 68 *Version I Reference Manual*, Control Data Services B.V., Rijswijk, The Netherlands, 1975.

[3] FISKER, R.G., *The Transput Section for the Revised* ALGOL 68 *Report*, Dissertation, Dept. of Computer Science, University of Manchester, August 1974.

AB41.4.3 Visibility and Teachability of I/O Processing in High-Level Languages. D. Holdsworth. (University of Leeds).

INTRODUCTION

A recent search for a widely available high-level language suitable for initial teaching to students in courses ranging over DP, mathematics and computational science has proved fruitless. In attempting to find a reason for this one is drawn to the conclusion that the DP student needs a greater control over I/O than is provided by most "scientific" languages, and DP languages (e.g. COBOL) take a derisory view of arithmetic. In addition where layout control is available (COBOL, FORTRAN, ALGOL68) it is provided by a lot of new syntax specific to I/O. This paper argues that the semantic rewards for learning this syntax are insubstantial, and consist primarily of a limited masking from the user of the characters making up a line of text. From an educational point of view this is probably a bad thing. Recent interest in teachable languages[1,2] has not tackled this problem.

We illustrate a possible solution for output, by proposing a scheme which is embedded in Algol68, and would argue that the result is in many ways an improvement in the facilities provided by the language definition[3]. The proposal involves no new syntax, and a partial implementation (using Algol68-R)[4] is included as Appendix 1. The major reason for the choice of Algol68 is the presence of generic user-defined operators in the language, thus making for a clean implementation of semantic concepts which are almost visible in COBOL. The system could be readily extended to include fixed-format input with a minimum of difficulty, but we still see some problems with free format input which seem to indicate that many languages make a mistake in imposing an artificial symmetry between input and output.

1. Algol68 I/O

The official I/O system of Algol68 not only involves new syntax in formats but also involves stretching unions close in dynamic typing, and so involves run time overheads, or "botching" the compiler to treat print etc. as special cases, as in Algol68C[5].

2. Mapping onto an output device

The scheme we propose is based on the idea that lines of characters are the only things that can be output. For this we use the outp operator. Where this is not true (i.e. interactive graphics devices) we envisage the provision of extra outp operators to perform the transput of non-character information. Thus conforming to our view that output can be handled by declarations utilising existing language features. The operator outp is either monadic or dyadic. The monadic form outputs a row of chars to standout, while the dynadic form has the name of the file as its first operand:

e.g.

<u>outp</u>"*line of output*"

is equivalent to

standout <u>outp</u> ***"line of output"***.

Control of paper motion can be achieved by providing system declarations of some other mode of object which defines the operation to be performed, as with the standard *newline* and *newpage*:

e.g. <u>outp</u> *newpage*

Probably the most convenient default is to have each <u>outp</u> produce a newline at the beginning, equivalent to:

print((*newline*,"*line of output*"))

New facilities like *sameline* could provide for requirements such as overprinting.

With such a scheme the choice between control operations and printing operations would be taken at compile time.

3. <u>Mapping onto rows of characters</u>

The above simple output system presupposes that there exist convenient syntactic constructs for construction of appropriate character strings. If Algol68 offered user-defined widenings, it would be possible to arrange that clauses such as:

line[7:11]:=i

would widen an integer i into a [1:5] <u>char</u>. However, our aim is to avoid introducing new language features largely for the benefit of the i/o system, although a user-defined widening opens up possibilities for other meaningful cases. Nonetheless, we shall not persue the avenue further. A very similar (superior?) facility is obtained by introducing a dyadic operator **repr** which stores a representation of its 2nd operand in its first operand. For character output of the sort we are considering, the 1st parameter would be of mode **ref**[1:]**char**, (see §4 for more powerful options offering layout control).

It is envisaged that in normal use there will be a character buffer used for assembling output, say:

[1:120]**char** *line*;

The construction of a line of output consisting of the values of an integer *i* and **real**s *x* and *y* would procede as follows:

> **clear** *line*;
>
> *line*[:5] **repr** *i*;
>
> *line*[7:16] **repr** *x*; *line*[18:27] **repr** *y*;
>
> **outp** *line*

This is undoubtedly longer than:

print((*newline*, *i*, *x*, *y*))

but it can be taught without invoking **union**s and row displays, and does in fact embody more layout control. A fairer equivalent would be:

printf((*$1ddddd,zzd.dddddddd,zzd.dddddddd$*,
i,*x*,*y*))

or *print*((*newline*,*whole*(*i*,5),*fixed*(*x*,-12,7),*fixed*(*y*,-12,7)))

There is ample scope for discussion about the default action on things such as zero suppression and signs. The system shown in appendix 1 suppresses leading zeros and the + sign. The character positions thus suppressed are left unchanged. This gives the user freedom to initialise the field with the zero suppression character. Others may argue for space filling.

Appendix 2 shows an example of a program to print solutions to an ordinary differential equation. The procedure *spr* produces the output which includes a simple graph in addition to numerical values.

4. Layout control for real numbers

The facilities already proposed include control of the field width of number output. For output of reals we commonly need to control the presence or absence of an exponent and the number of digits after the decimal point. In addition, for both ints and reals we may wish to control printing of signs and leading zeros.

It seems inevitable that increasingly fine control of layout will involve increasing amounts of detail. One option is to head straight for an all-embracing system. However, there seems to be genuine value in a means of controlling precision of output for reals while still taking default action for signs and zero suppression. We therefore introduce two new modes eformat and fformat (with deference to FORTRAN) whose ref[]char fields select the fields within a line which are to be used for different parts of the number:

```
mode eformat = struct(ref[]char mantissa,exponent)
mode fformat = struct(ref[]char  ipart, fpart);
```

If we now wish to enhance the example of section 3 to print *x* in fixed point with 5 decimal places and *y* in floating point with 4 decimal places we would write:

```
clear line; line[:5]repr i;
eformat yform = (line[18:23],line[24:27]);
fformat xform = (line[7:10],line[11:16]);
      ¢ in practice    the above 2 statements
      would be outside any loop ¢
xform repr x; yform repr y;
outp line;
```

Appendix 3 shows a modified version of *spr* of appendix 2 which utilises the above facilities.

The templates *xform* and *yform* play a role analogous to that of PICTURE's in COBOL, and repr is acting in a way similar to the MOVE verb.

5. General layout control

Clearly one can go on introducing increasingly complex structures, or have global variables to control the options such as zero suppression. Another option is to offer a general structure most of whose fields are unions.

This is perhaps the most attractive solution as the definition of this structure would be a formal (nearly) description of the layout facilities available, and any particular structure would be a syntax tree for the particular layout required. The initialisation of all the fields in such a structure would be tiresome, and a system would therefore provide some default skeletons (with <u>nil</u> for the <u>ref</u>[]<u>char</u> fields) into which a user could overwrite his own choices. Of course, we are now back to a large amount of run time analysis, but we have not introduced any special purpose syntax.

6. <u>Efficiency</u>

In the examples of appendices 1 and 3 we have manually selected only those declarations of <u>repr</u> and <u>outp</u> which our program invoked. This corresponds to a system where invokation of system library routines is automatic (as in Algol68-R). The appendix 2 version using <u>outp</u> and <u>repr</u> is 1000 words (24-bit) smaller than the standard version. The appendix 3 version is 5000 words smaller than its standard formatted i/o counterpart. Comparison of run times also favours the <u>outp</u>/<u>repr</u> version. This program was not created for the purpose of these examples but was originally written as a student exercise in Algol 60.

7. Extension to cover input

The extension to cover formatted input is fairly straightforward and involves an inp operator and possibly rper (?). The notion of a general layout control which specifies a syntax tree is interesting in the context of input. However, the more common requirement is for free-format input. Perhaps in this case we could have rper take the required input from the beginning of the []char operand, assign the value to the other operand and deliver as a result either the number of characters used, or a row consisting of the rest of the input string. However, this lacks some of the essential simplicity that we sought to introduce for teaching purposes. (The languages does contain a precedent in the very useful '/:='.) Appendix 4 shows an example where a matrix is input using this system after first reading bounds from a single line.

8. Conclusion

We have produced a blueprint for an output system for Algol68 without use of syntactic or semantic extensions to the language. We deal only in output of basic types, but the system makes easy the definition of user-defined repr's which will output any of a user's structures. The necessary looping for handling arrays is already provided in the language by the do constructs. It is suggested that the concepts involved in this output system are valuable to DP students, computer scientists and mathematicians alike.

There seems to be an obvious disadvantage of greater verbosity, but this is no bad thing if greater clarity and readability are a result. As to teachability - the system is untested in this area.

One facility which has arisen by accident, is the ability to print a row of reals with all the integral parts on one line and all the fraction parts below by use of formats of the form:

fformat *splitter* (*line* 1[?:?], *line* 2[?:?]).

We also have the ability, to edit the character output before printing by normal manipulation on the row of characters.

From a purely pedagogic point of view the separation of data transfer from character conversion seems valuable in a language which offers rows of characters. As an illustration of the minimal nature of semantic extension we may observe that the implementation *ioseg* (appendix 1) uses only 2 code patches (each one instruction) and each is very system dependent - the peripheral transfer extracode (in outp) and the paper feed field (pfcc). The last one could be eliminated by use of []char, but with some loss of efficiency.

Finally, let us compare the repr operator with the conversion operators of §10.3.2.1. of the Algol68 report[3]. While these routines offer the capability to deliver a string as a result of conversion they do not give the same feeling of mapping values into fields within a line; nor do we have the uniformity of syntax for different modes of values, a syntax which may be extended to cover user-defined modes by further declarations of <u>repr</u>.

REFERENCES

1. Designing a Beginner's Programming Language, L. Geurts and L. Meertens - Maths Centre Amsterdam preprint IW 46/75 - also in "New Directions in Algorithmic Languages 1975", edited by Stephen A. Schuman - IRIA.

2. Reliability, Portability, Teachability: Three Issues for New Programming Languages, O. Lecarme - "New Directions in Algorithmic Languages 1975".

3. Revised Report on the Algorithmic Language Algol68, A.van Wijngaarden et al - Springer Verlag 1976.

4. Algol68-R Users Guide, P.M. Woodward and S.G. Bond - H.M.S.O.

5. Algol68C Reference Manual, S.R. Bourne, A.D. Birrell, J. Walker. Computer Laboratory, Corn Exchange St., Cambridge.
N.B. the single bracket on print§5.4.4.

```
    [1:124]CHAR buff;
    REF BYTES pfcc = REF BYTES CODE 100,1/buff[1] EDOC;
    pfcc := "000A";  C pfcc for next record C

    INT ca := 8r200000, rep, nchars;
                              C setting up a control area for lp C
    REF CHAR addr := buff[4]; C get chars 3 pos C
    [1:60]CHAR errorline;
    errorline[:44] := "OUTPUT ERROR : 00 DIGIT FIELD WILL NOT HOLD ";

  OP OUTP = ( []CHAR line ) :
  BEGIN
    buff[5:UPB line + 4] := line;
    nchars := UPB line + 1;
    CODE 157,0/ca EDOC;        C transfer to lp C
    pfcc := "000A"             C reset pfcc to default C
  END;

  MODE PFC = BYTES;  C mode for paper feed control C
  PFC newline = "000A", newpage = "000I", sameline = "0001";

  OP OUTP = ( PFC control ) :
  BEGIN
    pfcc := control
  END;

  OP REPR  =  ( REF[]CHAR ch, INT i ) :
  Converts integer held in i into a row of chars in ch.
    Leading zeros are suppressed the resulting character
    positions are left unchanged.  Plus signs are suppressed
    and any minus sign is placed before the most significant
    digit.
  C
  BEGIN
    INT end := UPB ch + 1,  rest := ABS i;
    C end is the most sig end of the chars output so far
      rest is integer which remains to represented to the left of end
    C
    INT minend = ABS ( i<0 ) + 1;   C minimum allowed value of end
                                       - allows for minus sign C
    WHILE
      IF  end > minend
      THEN
        ch [ end MINUS 1 ]  :=  REPR ( rest'/:='10 );
        rest # 0      C stop when only zeros to left C
      ELSE
        FALSE
      FI
    DO
      SKIP;

    IF  i < 0     C minus sign needed C
    THEN
      ch [ end MINUS 1 ] := "-"
    FI;

    IF   rest # 0    C if integer was too big for layout C
    THEN
      errorline[16] := " ";     Clear 1st char  because of sero sup C
      errorline[16:17] REPR  UPB ch;
      ( errorline[45:] := "..............." )  REPR  i;
      OUTP errorline      C error report on standout C
    FI
  END;
```

```
MODE EFORMAT  =  STRUCT( REF[]CHAR m, e);
MODE FFORMAT  =  STRUCT( REF[]CHAR i, f);



OP REPR  =  ( FFORMAT ch, REAL x ) :
C Fixed Format Decimal.
  sign ( if -ve ) and integer part go into iOFch,
  and decimal point and fraction part go into fOFch.
C
BEGIN
  INT sign = SIGN x;
  INT i = ENTIER  ABS  x;
  REAL f := ABS x - i;
  INT w;   C working variable C

  iOFch  REPR  sign*i;   C handle integer part C

  ( fOFch )[1] := ".";  C decimal point C
  FOR i  FROM 2 TO UPB fOFch      C produce requird no of fraction digi
ts C
  DO
  BEGIN
    f := f*10;  C i part of f is next digit C
    w := ENTIER f;
    ( fOFch )[i] := REPR w;
    f := f - w
  END
END;


OP REPR  =  ( EFORMAT ch, REAL x )  :
C put real no in x into floating decimal in ch  C
BEGIN
  C any appropiate algorithm for conversion C
END;


OP REPR  =  ( REF[]CHAR ch, REAL x ) :
C  Represents x in the given field as appropriate.
   The format within ch is chosen so to give the most
   readable representation which fits the field
   without loss of accuracy.
C
BEGIN
  C any appropriate conversion routine C
END;


OP REPR  =  ( REF[]CHAR ch, BOOL b )  :
C bool to chars C
BEGIN
  IF  UPB ch < 5   C use 0 1 rep for short strings C
  THEN
    ch[1] := ( b ! "1" ! "0" )  C other chars unchanged - bad idea ? C

  ELSE
    ch[1:5] := ( b ! "TRUE " ! "FALSE" )
  FI
END;
```

Appendix 2

```
BEGIN
  REAL  a, b, ya, y, h, hj, fs, x, nf;
  INT i, j, nj, ns, nhalf;

[1:66]CHAR line;     C output line - used in spr C
REF[]CHAR yform = line[12:20];
REF[]CHAR xform = line[:9];

PROC spr = VOID :
COMMENT prints one line output for one x value C
BEGIN
  INT yn;                          set line to spaces
  CLEAR line;
  xform REPR x;  yform REPR y;               convert x and y
  IF  ABS y  <  fs    C if graph in scale C
  THEN
    yn := ENTIER(nf*y);       C scaled onto integer C
    line[41]  := "I";
    line[41+yn]  := "+"     C point to mark value C
  FI;
  OUTP line                          transfer to output
END;

PROC stepint = VOID :
COMMENT do one step of ode integration C
BEGIN
  REAL y1, y2, y3, y4;
  y1  := x*x + y*y;
  y2  := ( x + y*y1) * 2.0;
  y3  := ( y1*y1 + y*y2 +1 ) * 2.0;
  y4  := 6 * y1*y2 + 2*y*y3;
  y  := (((y4*h/4+y3)*h/3+y2)*h/2+y1)*h+y;
  x  := x+h
END;

a := 0;  b := 0.9;  ya := 1.0;
  nj := 5; ns := 5; nhalf := 1; fs := 15.0;
  nf := 20.0/fs;      C  scale for printer graph C
  region ( a, b,  ( fs < 0 ! fs ! 0 ) , ABS fs );
    CLEAR line;
  line[:5] REPR nj;  line[6:10] REPR ns; OUTP line;
  axessi ( 0.1, 1.0 );
  TO nhalf
  DO
  BEGIN
    ns := ns + ns;
    hj := ( b - a )/nj;  x := a;
    h := hj/ns;   y := ya;

    point ( x, y );
    TO nj DO
    BEGIN
        spr;
      FOR  i TO ns DO
        (  stepint;  join ( x, y);  plotas ( x, y, "1   ")  )
    END;
    spr;
    frame;
    OUTP "----------------------------------------";
    OUTP " "
  END
END
```

Appendix 3

```
EFORMAT yform = ( line[10:16], line[17:20] );
FFORMAT xform = ( line[:3], line[4:9] );

PROC spr = VOID :
COMMENT prints one line output for one x value C
BEGIN
  INT yn;
  CLEAR line;
  xform REPR x;  yform REPR y;
  IF  ABS y  <  fs     C if graph in scale C
  THEN
    yn := ENTIER(nf*y);       C scaled onto integer C
    line[41] := "I";
    line[41+yn] := "+"     C point to mark value C
  FI;
  OUTP line
END;
```

Output from above procedure

```
   5     5
 0.00000 1.0000E  0                      I+
 0.18000 1.2216E  0                      I+
 0.36000 1.5829E  0                      I +
 0.54000 2.2651E  0                      I   +
 0.72000 3.9516E  0                      I      +
 0.90000 1.4293E  1                      I                         +
-----------------------------------------
```

Output from program as given in Appendix 2

```
   5     5
 0.00E  0  1.0000000                     I+
0.1800000  1.2216790                     I+
0.3600000  1.5829401                     I +
0.5400000  2.2651024                     I   +
0.7200000  3.9516723                     I      +
0.9000000  14.293022                     I                         +
-----------------------------------------
```

Appendix 4

```
C ****** INPUT ( TENTATIVE ) ********** C

OP RPER  =  ( REF INT i, []CHAR ch )  INT :
C  takes an integer from the row ch regarding any character
   which cannot form part of an integer as terminator.
   Leading spaces are ignored.  The value of the resulting
   integer is assigned to i  and the result of the operator
   is the number of characters read.
C
     ..............

[1:120]CHAR line;

INT m, n,       C matrix bounds C
    i;          C character pointer used in reading C

   INPline;      C  read a line of input  C

   i :=  m RPER line;   C reads value of m and leaves i so that .. C
   n RPER line[i:];    C .. it can be used to read the rest of the line
C

[1:m, 1:n]REAL a;

   FOR  j  TO  n
   DO
   BEGIN
     i := 0;    INP line;  C initialise pointer and read input C
     FOR k  TO m
     DO
       i PLUS ( a[k,j] RPER line[i:] )
   END;    C converts into a[k,j] and increments the char pointer C

C  We do seem to have lost some of the desired simplicity !!! C
```

Afterthought: User-defined conversions from []CHAR to a user-defined mode cound be used by languages such as Alphard and CLU (see same book as references 1 and 2) to define the format of literals in program text.

This would mean that these routines would need to be executed at compile time.

AB41.4.4

THE SYNTAX OF AN ALGOL PROGRAM

A. N. Walker

[Dept of Mathematics, The University, Nottingham NG7 2RD.]

Abstract: '<u>begin</u> <u>real</u> x; x := 1 <u>end</u>' is proved to be a syntactically correct Algol 68 particular-program.

The folk-lore of Algol 68 has it that the Report and the Revised Report are such formidably obscure documents that it is quite impossible actually to follow through the syntax for any real programs. Despite discovering that at least one of the Revised Editors - who should perhaps remain nameless - thought so too, I attempted the task for the (hopefully) particular-program '<u>begin</u> <u>real</u> x; x := 1 <u>end</u>' and was pleasantly surprised to discover that it isn't really all that bad.

Fig (i) gives the complete production tree, except that (a) 'reference to real' is abbreviated to 'RR', (b) predicates are given separately - see fig (ii) -, and (c) productions of 'NEST' are abbreviated to 'N2', 'N2 new' or 'N4' as appropriate. 'N2' corresponds to all the declarations of the standard and particular preludes and is rather long if written out in full. 'N2 new' is the nest which also includes the labels before the '<u>begin</u>' (there aren't any!), and 'N4', which is 'N2 new new RR letter x', also includes an entry for '<u>real</u> x;'. Fig (iii) lists all the metaproductions actually used in deriving the production tree from the syntax rules quoted. (Note that fig (iii) does not include metaproductions used only in deriving fig (iii): for example, in order to derive 'NOTION: go on.', 'NOTION: go o, go, g.' and 'ALPHA: g, n, o.' are reqired, but are not given in fig (iii).)

Fig (i) contains 35 productions, which is, as it happens, exactly as many as are required in the Algol 60 syntax for the same program. However, Algol 68 has verified (as Algol 60 cannot) that the 'x' in 'x := 1' is that declared in '<u>real</u> x', that 'x' is suitable to have '1' assigned to it, and indeed that '1' must be widened in the process. Admittedly, the Algol 68 rules are slightly longer.

Fig (i) constitutes a proof that '<u>begin</u> <u>real</u> x; x := 1 <u>end</u>' is a particular-program provided that we verify that each of the predicates in fig (ii) holds. (We should also verify fig (iii), which is left as an

```
                         N2 particular program
               +--------------- <1011g> --------------+
               |                                      |
 N2 new joined label definition of,    strong void N2 new closed clause
          + <1011h>                                <311a>
          |    |                                     |
         [A], ø         strong void N2 new serial clause defining new RR letter x bold pack
          |         +----------------------------- <133d> --------------------------+
          ø         |                                |                              |
              bold begin token, strong void N2 new serial clause defining new RR letter x, bold end token
                    ↓                              <321a>                           ↓
                 'begin'                             |                            'end'
                                     strong void N4 series with RR letter x
         +--------------------+------------------- <321b> -----------------+
         |                    |                      |                     |
        [B],     N4 declaration of RR letter x,  go on token,    strong void N4 series with
         |                  <411a>                   ↓             +---- <321b> ---+
         ø                    |                     ';'            |               |
             N4 RR variable declaration of RR letter x            [A],      strong void N4 unit
               +---------- <441e> -----------+                     |             <321d>
               |                             |                     ø               |
RR N4 local sample generator, N4 RR variable joined definition of RR letter x   strong void N4 assignation coercee
   +------- <5231b>                        <411c>                   +----------- <611a> ----------+
   |           |                             |                      |              |              |
  [C], actual real N4 declarer  N4 RR variable definition of RR letter x   [G], voided to void N4 assignation, [H]
   |         <461a>        +--------------- <441f>                  |            <671b>           |
   ø           |           |                 |                      ø              |              ø
               |          [D], RR N4 defining identifier with letter x  unchanged from RR N4 assignation
               |           |             +- <481a> -+                            <611f>
               |           ø             |          |                              |
```

```
             |                                   |       |                                     |
             |                                  [F], letter x token                    RR N4 assignation
real i N4 applied mode indication with real      |       ↓                  +-----------------+----- <5211a> ---+
      +------ <481b>                             ø      'x'                 |                 |                 |
      |      |                                                     RR N4 destination, becomes token, real N4 source
     [E], real token                                                    <5211b>               ↓        +-- <5211c> -+
      |      ↓                                                            |                 ':='       |            |
      ø    'real'    soft RR N4 applied identifier with letter x coercee                   strong real N4 unit, [J]
                                       <611e>                                                    <321d>         |
                                          |                                                        |            ø
                 unchanged from RR N4 applied identifier with letter x      strong real N4 denoter coercee
                                       <611f>                                +------------ <611a> ------------+
                                          |                                  |               |                |
                        RR N4 applied identifier with letter x              [K], widened to real N4 denoter, [L]
                                  +-- <481b> --+                             |             <651a>             |
                                  |            |                             ø               |                ø
                                 [I], letter x token                         unchanged from integral N4 denoter
                                  |            ↓                                           <611f>
                                  ø           'x'                                            |
                                                                                     integral N4 denoter
                                                                  +-------------------- <801a>
                                                                  |                          |
                                                       pragment sequence option, integral denotation
                                                               <133a>                    <8111a>
                                                                  |                          |
                                                                  ø                  fixed point numeral
                                                                                         <8111b>
                                                                                             |
                                                                                   digit cypher sequence
                                                                                          <133b>
                                                                                             |
                                                                                        digit cypher
                                                                                          <8111c>
                                                                                             |
                                                                                      digit one symbol
                                                                                             ↓
                                                                                            '1'
```

Notation:

```
     ø: EMPTY.
    RR: reference to real.
    N2: new new ... new ... letter p,
          a production of NEST2 - see rule
          <1011g> - containing layers for the
          standard & particular preludes.
    N4: N2 new new RR letter x.
 <...>: rule in Revised Report.
 '...': reference language representation.
   [.]: predicate - see fig (ii) & text.
     ↓: abbreviation for the production rules between token
          or symbol and the reference language representation.
             Thus:              NOTION token
                           +------- <911f> -------+
                           |                      |
                  pragment sequence option, NOTION symbol
                          <133a>                <941>
                           |                      |
                           ø             'some representation'
```

fig (i) - production tree for the reference language particular-program
'<u>begin</u> <u>real</u> x; x := 1 <u>end</u>'.

[A]: where () is ()
[B]: where (reference to real letter x) is (reference to real letter x)
[C]: where (local) is (local)
[D]: where (real) is (real)
[E]: where real i real identified in N4
[F]: where reference to real letter x independent
[G]: where (N4 assignation) is (N4 assignation)
[H]: unless (voided to void) is (deprocedured to void)
[I]: where reference to real letter x identified in N4
[J]: where real deflexes to real
[K]: where (N4 denoter) is (N4 denoter)
[L]: unless (widened to real) is (deprocedured to void)

fig (ii) - table of predicates used in fig (i)

exercise. None of it will take you very far afield except the expansions for 'NOTION' and 'NOTETY' which get very tedious unless you use theorem 1, below.)

Theorem 1 Any non-empty sequence of small syntactic marks other than '(' and ')' { ie of letters 'a', 'b', ..., 'z'; see section 1.1.3.1.a in the Revised Report } is a terminal metaproduction of 'NOTION'.

Proof Otherwise, note that any single such mark is a terminal metaproduction of 'ALPHA', and let S be a shortest counterexample. Then S contains more than one mark, and can therefore be written as the concatenation of a shorter non-empty sequence S1 and a (single) mark, M say. But S1, by hypothesis, is a 'NOTION' and M is an 'ALPHA', so S may be produced from 'NOTION ALPHA'. This contradiction establishes the result.

Corollary Any (possibly empty) such sequence is a terminal metaproduction of 'NOTETY'.

Theorem 2 The predicate 'where (NOTETY) is (NOTETY)' holds.

Proof We use the following lemma.

Lemma 'where (NOTETY) begins with (NOTETY)' holds.

Proof { left as an exercise! Use contradiction on a shortest counter-example, by rules 1.3.1.a, 1.3.1.i, 1.3.1.j and 1.3.1.k, and theorem 1. }

Theorem 2 follows immediately from 1.3.1.g, 1.3.1.c and the lemma.

COMMON: reference to real variable.

COMORF: N4 assignation; N4 denoter.

DECS: reference to real letter x.

DECSETY: .

DIGIT: digit one.

EMPTY: .

ENCLOSED: closed.

FORM: N4 applied identifier with letter x; N4 assignation; N4 denoter.

INDICATOR: identifier; mode indication.

LABSETY: .

LAYER: new reference to real letter x.

LEAP: local.

MEEK: unchanged from.

MODE: real; reference to real.

MODINE: real.

MOID: integral; real; reference to real; void.

NEST: N2; N2 new; N4.
(See fig (i).
N2: new new D1 new D2 label letter s letter t letter o letter p.
N4: N2 new new reference to real letter x.
D1: <declarations of standard prelude>.
D2: <declarations of particular prelude>.)

NOTETY: strong void N2 new serial clause defining new reference to real letter x.

NOTION: becomes; bold begin; bold end; digit cypher; go on; letter x; pragment sequence.

PACK: bold pack.

PROP: reference to real letter x.

PROPSETY: reference to real letter x; .

QUALITY: real i; reference to real.

REF: reference.

SIZETY: .

SOFT: unchanged from.

SOID: strong void.

SOME: strong real N4; strong void N4.

STRONG: voided to; widened to.

STYLE: bold.

TAB: real.

TAG: letter x.

TALLY: i.

TAX: letter x; real.

TERTIARY: applied identifier with letter x coercee.

UNIT: assignation coercee; denoter coercee.

VIRACT: actual.

fig (iii) - metaproductions used in fig (i)

Application of theorems 1 and 2 shows that predicates [A, B, C, D, G, K] all hold. Predicates [H, L] require a sort of converse:

Theorem 3 Let S1 and S2 be terminal metaproductions of 'NOTETY' of different lengths. Then 'unless (S1) is (S2)' holds.

Proof Again, we need a lemma.

Lemma If S1 is shorter than S2, 'unless (S1) begins with (S2)' holds.

Proof § again, left as an exercise! Use contradiction on a counter-example having S1 as short as possible, and rules 1.3.1.e, 1.3.1.h and 1.3.1.j. §

Theorem 3 now follows immediately. A more general result can be proved without enormous difficulty, but would take us through the obscurities of 1.3.1.l and 1.3.1.m.

Theorem 3 establishes [H, L]. Predicates [F, J] yield immediately to rules 7.1.1.b and 4.7.1.a respectively. Unfortunately, the remaining predicates, [E, I], take us through some remarkably obscure syntax. Take first [I]. From 'where RR letter x identified in N2 new new RR letter x', we produce (7.2.1.a) 'where RR letter x resides in RR letter x', and hence (7.2.1.c) 'where reference to real equivalent reference to real'. However, this does not 'obviously' hold, as you will soon discover if you start following the syntax from 7.3.1.a. There is no general theorem 'where MODE equivalent MODE', because the syntax also checks that 'MODE' is well-formed. This is, of course, exactly the sort of side-effect that many of us complain about when it is perpetrated by our students. Theorem 4, below, deals with this particular case, but I should hate to have to prove a mode equivalent to itself if there were a couple of (perfectly innocent?) structs and unions around.

Now consider [E]. The intention is clearly to arrive at the mode-declaration of 10.2.2.d, and indeed it is not too hard to verify that 'where real i real independent P' holds for P being 'reference to real letter x', empty, and the 'PROPSETY' of the particular-prelude (section 10.5.1; note the sentence beginning 'However, ...' and that each declaration in 10.5.1 is an identifier-declaration and is therefore independent of any mode-declaration), so that predicate [E] is reduced to 'where real i real identified in N1' where 'N1' is 'new new D1' in the notation of fig (iii). CHL argues that 10.2.2.d proves that 'D1' is of form 'DECSETY real i real PROPSETY' and that to enquire further is metaphysical speculation. This seems to be a weakness in the definition of pseudo-comments (10.1.3, step 7), because there is a universal panacea for nasty closed clauses (eg '(pragmat code machine-code for some horrible operation pragmat skip)'), but no way out for indescribable declarers. (The device 'mode real = struct (int exponent, long int mantissa)' avoids some but not all of the problems.) Anyway, if we accept CHL's argument, [E] reduces quickly to 'where real i real resides in real i real', which in turn reduces to 'where real equivalent real'. This too is easier to prove from theorem 4 than directly.

Theorem 4 'where SAFE1 PREFSETY PLAIN equivalent SAFE2 PREFSETY PLAIN' holds.

Proof Otherwise, let M be a shortest terminal metaproduction of 'PREFSETY PLAIN' which permits (for suitable 'SAFE1', 'SAFE2') a counterexample. By hypothesis, and the first production of 7.3.1.b, 'unless (SAFE1) contains (remember M M) or (SAFE2) contains (remember M M)' holds. If M is 'PLAIN', then 'where (M) is (M) and remember M M SAFE1 equivalent SAFE2' holds by theorem 2 and 7.3.1.q, and 'where SAFE1 M develops from SAFE1 M and SAFE2 M develops from SAFE2 M' holds, by 7.3.1.c, theorem 2 and 7.4.1.a. If, alternatively, M is 'PREF PREFSETY PLAIN', then 'where (PREF) is (PREF) and remember yin SAFE1 PREFSETY PLAIN equivalent yin SAFE2 PREFSETY PLAIN' holds by theorem 2 and the hypothesis, and 'where yin SAFE1 M develops from SAFE1 M and yin SAFE2 M develops from SAFE2 M' holds by 7.3.1.c, theorem 2 and 7.4.1.b. In either case, there is a contradiction, which establishes the theorem.

Corollary 'where PREFSETY PLAIN equivalent PREFSETY PLAIN' holds.

Proof 7.3.1.a and theorem 4.

The corollary establishes 'where reference to real equivalent reference to real' and 'where real equivalent real' and hence completes the verification of all predicates. Thus 'begin real x; x := 1 end' is indeed a particular-program. It even appears to be a meaningful particular-program provided that 'maxint' is at least one.

AB41.4.5 A Token Recognizer For The Standard Hardware Representation of Algol 68.

by

R. Bell, Department of Computer Science,
Teesside Polytechnic,
Middlesbrough,
England.

0 Introduction

This token-recognizer is designed to scan texts which purport to be Algol 68 particular-programs in the standard hardware representation defined by Hansen and Boom[1]. It will seek to parse each given text into a sequence of language tokens, digestible by, for instance, the syntax analyser of an Algol 68 compiler.

As the word "token" bears a specialized meaning in Algol 68, this document will instead speak of "words", which are, broadly Algol 68 TAX-symbols, denotations or other NOTION-symbols[2]. Each activation of the recognizer will deliver a representation of just one such "word" to the superior routine that drives it.

This recognizer may serve, it is hoped, as a general purpose front-end component, not only for full compilers but also for syntax checkers or preprocessors.

The algorithm is presented in Algol 68. Readers are warned that it has not been machine-checked directly (because the author has no access to any compiler for canonical Algol 68). However, an analogous program in Algol 68R has been written and compiled and is being tested.

1 Hansen W.J. and Boom H.
Report on the Standard Hardware Representation for Algol 68, (AB 40.5) in Algol Bulletin 40 (pp 24 - 43), 1976. (hereinafter designated by "HR").

The other fundamental document is, of course:

Wijngaarden A.v. and others,
Revised Report on the Algorithmic Language Algol 68, Springer Verlag, 1976 (and elsewhere). (designated by "RR").

2 In this document, Algol 68 paranotions are hyphenated where necessary and (except in section 2) underscored.

1 Words

This recognizer does not deal with the following contexts in particular-programs :-

(a) interiors of pragments (and by implication, their terminators);
(b) interiors of format-texts (and by implication, their terminators), except that it is applicable to closed-clauses, CHOICE-clauses, units or denoters discovered inside format-texts .

This recognizer may encounter, where it is applicable, six classes of "words". The initial character of a word implies its class.

It is assumed here that the set of "base characters" which occur in texts is identical to the set of "worthy characters" defined in HR1, and may include both upper and lower case letters.

The six classes of words are:-

(1) Tags, i.e. TAG-symbols, which are identifiers, label-identifiers or field-selectors ;

(2) Bold-words:
There are 61 specified bold-words which are fixed as representations of certain NOTION-symbols (see Appendix). Any other bold-word must be a bold-TAG-symbol, and as such either a mode-indication (TAB-symbol) or an operator (TAO-symbol) ;

HR3.5 explains how tags and bold-words are differentiated: mainly by "stropping", of which there are three alternative standard regimes, "point", "upper" and "res".

(3) integer-denotations, real-denotations, bits-denotations (also digit-symbols in priority-definitions) ;

(4) character-denotations, string-denotations ;

(5) operators which are not bold-Tag-symbols, i.e. DOP-BECOMESETY-symbols ; also the is-defined-as-symbol ;

(6) Some other NOTION-symbols (e.g. ¢, |, |:) .

Outside character- and string-denotations, "point" and "res" stropping do not distinguish between upper and lower case letters (a and A are regarded as the same character); "upper" stropping does distinguish (indeed, requires both cases to be used), and confines upper case letters to bold-words.

The classifying powers of initial characters of words are as follows :-

CHARACTER	SIGNIFICANCE	CLASS
a letter	"point" stropping : start of a tag "upper" stropping : lower case letter : start of a tag upper case letter : start of a bold-word "res" stropping : start of a tag or of a reserved bold-word or of a tag followed by a reserved bold-word	1 1 2 1 2 1,2
. (point)	if followed by a letter : start of a bold-word if followed by a digit : start of a real-denotation otherwise an incorrect character at this level	2 3 -
a digit	start of an integer- or real- or bits-denotation, or a digit-symbol (priority)	3
" (quote)	start of a character- or string-denotation	4
% + - < > / *	start of an operator	5
=	start of an operator, or is-defined-as-symbol	5
: (colon)	label- or colon- or up-to- or routine-symbol, or start of becomes- or is- or isnt-symbol	6
| (stick)	brief-then/in//else/out-symbol or start of brief-elseif/ouse-symbol	6
= ¢ ( ) , ; @ [ ]	various NOTION-symbols	6
' _	incorrect characters at this level	-

Spaces and newlines are of no significance at this level. Logical-end-of-text might be treated as a fault, as Algol 68 particular-programs are supposed to be well-closed.

2 Algorithm

This is presented in an "upper stropped" representation of Algol 68, except that, as in RR10, there are certain particular constructs whose precise forms are left to the discretion of implementors: these are informally described by "pseudo-comments" which are bounded by the marks <u>C</u> ... <u>C</u> .

The algorithm is given in two parts:
the recognizer procedure, called "get word"
and (preceding "get word")
declarations necessary to create the environment for "get word".

Two details of the algorithm should be particularly noted.

Under "res" stropping it may be found that a reserved bold-word follows a tag. This possibility must be resolved during one activation of the recognizer: the tag is delivered and the reserved bold word is held in a (non-local) variable until it (and no subsequent word) is delivered on the next activation of the recognizer.

Certain concatenations of characters starting with DOP-symbols are ambiguous until more information about the context is known (which the recognizer in itself cannot provide). In concatenations such as <= , <=:= , the final "=" might be part of the operator or a separate <u>is-defined-as-symbol</u>. The latter is the case if it is a defining occurrence of the operator (i.e. in an <u>operation-definition</u> or a <u>priority-definition</u> and the next "word" is not also "=". All these ambiguous concatenations are split into two words by the algorithm.

{2.1 Environment}

```
COMMENT
        The following declarations are to be made in ranges
        embracing the declaration of the recognizer procedure

        1:  Forms dealing with character classification,  cf HR C6
COMMENT

INT  upletter = max abs char + 1,
     adigit = max abs char + 2,
     another = max abs char + 3 ;

[ : ]INT  chartype
          =
          C  A row of integers with bounds  [0 : max abs char] ,
             having the property  #implementation-dependent#

              chartype[i] =
              IF   REPR i  is neither a letter nor a digit
              THEN  another
              ELIF REPR i  is a digit
              THEN  adigit
              ELIF REPR i  is an upper case letter
              THEN  upletter
              ELSE #(REPR i  is a lower case letter)#
                   ABS  the corresponding upper case letter
              FI
          C

PROC(REF CHAR)BOOL  uletter = (REF CHAR c)BOOL :
                              chartype[ABS c] = upletter ,

                    sletter = (REF CHAR c)BOOL :
                              IF   INT  ti = chartype[ABS c] ;
                                   ti <=  max abs char
                              THEN
                                   # ( c  refers to a lower case
                                     letter, which is replaced by
                                     the corresponding upper case
                                    letter ) #
                                   c :=  REPR ti  ;
                                   TRUE
                              ELSE
                                   FALSE
                              FI ;

PROC(REF CHAR)BOOL  letter = (REF CHAR c)BOOL :  uletter OR sletter ;

PROC(CHAR)BOOL  digit = (CHAR c)BOOL :  chartype[ABS c] = adigit ;

STRING  emptystring = "" ,

CHAR  underscore = "_" ,  space = " " ,  quote = """" ,

      apostrophe = C The denotation of the apostrophe character C  ;
```

```
COMMENT
          2:  Forms dealing with reading the input text
COMMENT


REF CHAR  char =  LOC CHAR :=  space
#(to hold the character in hand)#,

REF BOOL  eol =  LOC BOOL :=  FALSE   #(see below)# ;

PROC(REF CHAR)VOID  get next character
  = (REF CHAR ch)VOID
  :  C
        A routine which reads the next available character from the
        input text and assigns it to  ch
        ( and perhaps also transcribes the input text to a listing
        ( into which warning and fault messages etc may be
        interpolated ) ).

        Event routines for whichever file is currently accessing the
        input text should behave as follows :-
         (a)  On logical file end - resort to the operating-system,
              which may either (if commanded and able to)
              mend the file so that reading can continue from
              another input text (book)
              and make  eol  (see above) :=  TRUE ,
              or abort the run ;
         (b)  On page end - call  newpage  and make  eol := TRUE ;
         (c)  On line end - call  newline  and make  eol := TRUE

        #(hence if an event occurs and is cleared,
        eol = TRUE  and the character from the next good position
        is assigned to  ch )#
     C ;


#("point" stropping will be the default regime;
Stropping regimes are switched by pragmats, see HR3.5 )#

REF BOOL  upperstrop = LOC BOOL := FALSE ,
          resstrop = LOC BOOL := FALSE ;


# If the fixed-point-numeral of an INTREAL-denotation is followed
by a point, it is necessary to look ahead to see if the point is
followed by a letter, in which case INTREAL- is integer- and the
point must be deemed to be the strop for a following bold-word  #

REF BOOL  intpointletter  =  LOC BOOL  :=  FALSE  ;
```

```
COMMENT
            3:  Forms associated with information generated by the
                recognizer

            Each time it is called the recognizer generates a "word",
            which is a structured value consisting of a string and a
            procedure.  The procedure will depend on what the word is
            that has been recognized in the input, and on the use to
            which the recognizer is being put.
            The routines to be ascribed to these procedures are
            therefore left undefined here;  provision is made for
            these routines to have parameters various in numbers and
            modes, by proposing that all the "word" procedures have
            one parameter whose mode is a union of a sufficient set
            of modes (left undefined here)
COMMENT

MODE  WORDPARAMS  =  UNION ( C of a sufficient set of modes C ) ;

MODE  WORD  =
     STRUCT ( STRING repstring ,  PROC(WORDPARAMS)VOID wordproc ) ;

PROC(WORDPARAMS)VOID

    C definitions of procedures with the following identifiers :-

       atproc,  boldbeginproc,  bitsmodeproc,  ......
       ......  and similarly for all the reserved bold words  .....
       ......  unionproc,  voidproc,  whileproc,

       # and #

       boldtagproc,  tagproc,  bitsdenproc,  badbitsdenproc,
       intdenproc,  realdenproc,  badrealdenproc,  chardenproc,
       stringdenproc,  estringdenproc,  tadproc,  taoproc,
       badtaoproc,  equalsproc,  colonproc,  becomesproc,
       badisntproc,  briefthinelseoutproc,  briefelifouseproc,
       hashcommentproc,  formatterproc,  lparenproc,  rparenproc,
       andalsoproc,  goonproc,  briefsubproc,  briefbusproc,
       badcharproc

    C  ;

   COMMENT
            In two instances (as will be seen) the recognizer has to
            look one word ahead in the input text
   COMMENT

REF BOOL  word held =  LOC BOOL :=  FALSE ,
REF WORD  held word =  LOC WORD  ;
```

{2.2 Recognizer Routine}

```
PROC (REF WORD) VOID  get word

   = (REF WORD  w ) VOID

   :  w :=

      IF   word held

      THEN
           word held :=  FALSE ;
           held word

      ELSE

           # read and ignore any typographical features
             preceding a word #

           WHILE  char = space
             DO  get next character (char)  OD ;
           eol :=  FALSE  ;

           IF     #1#

                BOOL  ul = uletter(char),
                      ll = sletter(char),
                      dgt = digit(char),
                      pt =  char = "."
                      #  only one of these can be  TRUE  #  ;

                IF  pt  THEN  get next character (char)  FI  ;

                BOOL  ptsameline =  pt AND NOT eol
                                    OR intpointletter  ;
                intpointletter  :=  FALSE  ;

                BOOL  pul = ptsameline AND uletter(char) ,
                      pll = ptsameline AND sletter(char) ,
                      pdgt = ptsameline AND digit(char)
                      #  and only one of these can be  TRUE  #  ;

                pt  AND  NOT( pul OR pll OR pdgt )

           THEN   #1#

                C  emit a fault message (impermissible character) C ;

                ( "."  ,  badcharproc )

           ELIF   #1#

                ul  OR  ll  OR  pul  OR  pll
```

```
THEN   #1#

     #  a bold word or a tag  #

     PROC (PROC VOID) VOID  break in tag
        = (PROC VOID  p ) VOID
        :   WHILE
              BOOL  le = eol  ;
              eol  :=  FALSE  ;
              BOOL  u =  char = underscore  ;
              IF   u
              THEN C emit a warning
                     (unwanted underscore in tag) C
              FI  ;
              IF   u  OR  char = space
              THEN get next character (char)  ;
                   TRUE
              ELSE le
              FI
               DO   p   OD   ;

     BOOL  pointstrop =  NOT ( upperstrop OR resstrop ) ;

     IF     #2#

          pointstrop AND NOT pt  OR  upperstrop AND ll

     THEN   #2#

          # a tag  (for tags under resstrop see later) #

          PROC (PROC(REF CHAR)BOOL) WORD  tagscanner
             = (PROC(REF CHAR)BOOL  charbool ) WORD
             :   BEGIN
                   REF STRING  tagstring = LOC STRING
                                         := char  ;
                   WHILE
                     get next character (char) ;
                     IF   NOT eol
                          AND char = underscore
                     THEN get next character (char)
                     FI  ;
                     break in tag (VOID:SKIP) ;
                     # one underscore is allowed
                       after each taggle,
                       newlines and spaces between
                       taggles are immaterial #
                     charbool (char)
                      DO
                          tagstring PLUSAB char
                      OD ;

                   ( tagstring,  tagproc )

                 END  ;
```

```
      IF
           upperstrop
      THEN
           tagscanner ((REF CHAR c)BOOL
                       : sletter(c) OR digit(c) )
      ELSE
           tagscanner ((REF CHAR c)BOOL
                       : letter(c) OR digit(c) )
      FI

ELSE  #2#

      # a bold word if pointstrop or upperstrop,
        either (or both) if resstrop #

      PROC (STRING, REF WORD) BOOL matchres
         = (STRING charstring, REF WORD rword) BOOL
         :  BEGIN
              # tests if  charstring  matches any
                reserved bold word #
              [ : ]WORD restable
                       = ( ("AT"    , atproc         ),
                           ("BEGIN", boldbeginproc),
                           ("BITS" , bitsmodeproc ),
                           (  C .....  and so on
                                for all the reserved
                                bold words  .....
                                ..........  .  C    ),
                           ("UNION", unionproc     ),
                           ("VOID" , voidproc      ),
                           ("WHILE", whileproc     ) ) ;

              [ : ]STRING resstrings
                         = repstring OF restable ;
              INT top =  UPB resstrings ;
             STRING firstres = resstrings[1] ,
                    lastres = resstrings[top] ;
              REF BOOL  found =  LOC BOOL :=  FALSE ;
              IF
                   charstring>=firstres
                   AND
                   charstring<=lastres
              THEN
                   REF INT  i  =  LOC INT ;
                   IF   found :=  charstring firstres
                   THEN i  := 1
                   ELIF found :=  charstring lastres
                   THEN i  :  top
```

```
                        ELSE
                          # seek a match by binary chop #
                          REF INT  s =  LOC INT
                                     :=  (top + 1) OVER 2 ;
                          i  :=  s ;
                         WHILE
                            STRING entry = resstrings[i] ;
                            NOT (found := charstring=entry)
                            AND s > 1
                             DO
                                s := (s + 1) OVER 2 ;
                                IF   charstring < entry
                                THEN  i MINUSAB s
                                ELSE  i PLUSAB s
                                FI
                             OD
                        FI  ;
                        IF   found
                       THEN rword :=  restable[i]
                        FI
                   FI  ;
                   found
                 END   ;

IF      #3#

     pt  OR  upperstrop AND ul

THEN  #3#

     PROC (PROC(REF CHAR)BOOL) WORD  boldscanner
        = (PROC(REF CHAR)BOOL charbool ) WORD
        :   BEGIN
               REF STRING  boldstring = LOC STRING
                                      := char  ;
              WHILE
                get next character (char) ;
                NOT eol AND charbool(char)
                 DO
                     boldstring PLUSAB char
                 OD   ;
              IF
                   REF WORD  rbw =  LOC WORD ;
                   matchres(boldstring, rbw)
              THEN
                   rbw
              ELSE
                   (boldstring, boldtagproc)
              FI
            END   ;
```

```
        IF
              upperstrop
        THEN
              IF   ul OR pul
              THEN
                   boldscanner ((REF CHAR c)BOOL
                                : uletter(c)
                                  OR  digit(c) )
              ELSE
                   #point followed by lower case#
                   boldscanner ((REF CHAR c)BOOL
                                : sletter(c)
                                  OR  digit(c) )
              FI
        ELSE
              boldscanner ((REF CHAR c)BOOL
                           : letter(c) OR digit(c) )
        FI

ELSE   #3#

        # resstrop and word does not
          begin with a point #

        REF BOOL  tag held =  LOC BOOL :=  FALSE ,
                  resposs =  LOC BOOL :=  TRUE ,
                  resfound =  LOC BOOL :=  FALSE ;

        REF STRING  tagstring =  LOC STRING
                              :=  emptystring ,
                    taggle =  LOC STRING  ;

        REF WORD  rbw =  LOC WORD  ;

        WHILE

          taggle :=  char  ;

          WHILE
            get next character (char) ;
            NOT eol
            AND
            (letter(char) OR digit(char))
               DO
                  taggle PLUSAB char
               OD   ;
```

```
    IF
         NOT eol AND char=underscore
    THEN
         resposs :=  FALSE ;
         get next character (char)
    ELSE
         # an apparent taggle may be a
           reserved bold word if it is
           bounded by disjunctors and
           not adjacent to an underscore #
         IF   resposs
         THEN resfound :=
                  matchres(taggle, rbw)
         FI
    FI  ;

    break in tag (VOID: resposs := TRUE) ;
    # if there are typographical display
      features then resposs is reset ready
      for the next apparent taggle #

    NOT resfound
   AND
    (BOOL l = letter(char) ;
     resposs := resposs AND l ;
     l OR digit(char) )
    # a taggle may start with a letter or
      a digit, but every reserved bold
      word starts with a letter #

       DO
          tag held :=  TRUE  ;
          tagstring PLUSAB taggle
       OD   ;

# the input may contain a tag followed by
  an object recognized firstly as an
  apparent taggle and secondly as a
  reserved bold word; i.e. two words may
  be recognized in one activation of the
  recognizer; alternatively, the first
  apparent taggle may or may not be a
  reserved bold word #

IF
     tag held
THEN
     IF   resfound
     THEN
          word held :=  TRUE ;
          held word :=  rbw
     FI  ;

     (tagstring, tagproc)

ELSE

     rbw

FI
```

```
            FI      #3#

        FI      #2#

         # finished with tags and bold words #

    ELIF   #1#

        dgt OR pdgt

    THEN   #1#

        # an INTREAL-denotation or a bits-denotation
          (or a digit-symbol in a priority-definition) #

        REF STRING  denstring  =  LOC STRING  :=
                    IF pdgt THEN "0." ELSE emptystring FI + char ;

        PROC VOID  get digits
           = VOID :  WHILE  get next character (char) ;
                            NOT eol  AND  digit(char)
                        DO  denstring PLUSAB char  OD ;

        PROC BOOL  aletterproc
           = BOOL :  IF   eol
                     THEN  FALSE
                     ELIF upperstrop
                     THEN  sletter(char)
                     ELSE  letter(char)
                     FI  ;

        get digits ;

        BOOL  aletter  =  aletterproc  ;

        IF      #2#

            dgt  AND  aletter  AND  char = "R"

        THEN   #2#

            # a bits-denotation #

            denstring PLUSAB "R"   ;
            REF BOOL  radixright  =  LOC BOOL  :=  TRUE  ,
                      digits  =  LOC BOOL  :=  FALSE  ;

            [ : ]CHAR  bitsdigits
                      =  IF   denstring = "2R" THEN "01"
                         ELIF denstring = "4R" THEN "0123"
                         ELIF denstring = "8R" THEN "01234567"
                         ELIF denstring = "16R"
                          THEN "0123456789abcdef"
                                 +
                                IF upperstrop
                                THEN emptystring
                                ELSE "ABCDEF"
                                FI
                         ELSE     radixright := FALSE ;
                                  SKIP
                         FI  ;
```

```
IF     #3#
      radixright
THEN   #3#
      WHILE
        get next character (char)  ;
        NOT eol
         AND
        char in string (char, LOC INT, bitsdigits)
          DO
             digits := TRUE  ;
             denstring PLUSAB (sletter(char) ; char)
             # changes any lower case letters
               to upper case                     #
          OD  ;

      IF   digits
      THEN
           ( denstring, bitsdenproc )
      ELSE
           C  emit a fault message
              (no digits in bits-denotation)  C  ;
           ( denstring, badbitsdenproc )
      FI

ELSE   #3#
      C  emit a fault message (wrong radix
         in supposed bits-denotation)  C  ;
      WHILE
        # may maul the next word #
        get next character (char) ;
        NOT eol
         AND
        ( IF upperstrop THEN sletter(char)
                        ELSE letter(char) FI
            OR
          digit(char)  )
         DO
             denstring PLUSAB char
         OD  ;

      ( denstring, badbitsdenproc )

FI     #3#
```

```
ELIF   #2#

     BOOL  intpoint   =  dgt AND NOT eol AND char ="." ;

     BOOL  intandfracpart = IF   intpoint
                            THEN
                             get next character(char) ;
                             intpointletter :=
                                         letter(char) ;
                             NOT intpointletter
                            ELSE
                             FALSE
                            FI   ,
           intandexpart   = dgt AND aletter
                                AND char = "E" ;

     dgt  AND  NOT( intandfracpart OR intandexpart )

THEN   #2#

     # an integer-denotation or a digit-symbol #

     ( denstring, intdenproc )

ELSE   #2#

     # a real-denotation #

     REF BOOL  fracright  =  LOC BOOL  ;

     BOOL  expart   =   IF   pdgt
                        THEN fracright := TRUE ;
                             aletter AND char = "E"
                        ELIF  intandfracpart
                        THEN IF   fracright := digit(char)
                             THEN
                              denstring PLUSAB "."+ char ;
                              get digits ;
                              aletterproc AND char = "E"
                             ELSE
                              FALSE
                             FI
                        ELSE #intandexpart#
                             denstring PLUSAB ".0"  ;
                             fracright  := TRUE
                        FI  ;

     IF      #3#
          expart
     THEN    #3#
          denstring PLUSAB "E" ;
          get next character (char) ;
          IF   char = "+" OR char = "-"
          THEN denstring PLUSAB char ;
               get next character (char)
          ELSE denstring PLUSAB "+"
          FI  ;
          get digits
     FI      #3#  ;
```

```
                IF
                     fracright AND digit(denstring[UPB denstring])
                THEN
                     ( denstring, realdenproc )
                     # integral-part and fractional-part
                       of denstring will contain
                       at least the digit 0 #
                ELSE
                     C  emit a fault message
                        (ill formed real-denotation)  C  ;
                     ( denstring, badrealdenproc )
                FI
           FI     #2#
      ELIF   #1#
           char = quote
      THEN   #1#
           # a character- or string-denotation #
           PROC VOID  eol in string
              = VOID :  IF   eol
                        THEN C  emit a warning  # see HR C4 #
                                (string-denotation
                                 broken by end of line)      C ;
                             eol  :=  FALSE
                        FI  ;
           REF STRING  denstring  =  LOC STRING  :=  emptystring ;
```

```
WHILE

  get next character (char) ;
  eol in string ;

  IF   char = apostrophe
  THEN
       get next character (char) ;
       eol in string ;
       IF   char ≠ apostrophe
       THEN  C  # see HR A3.1 #  a routine to deal
                 with the situation where a single
                 apostrophe in a string-denotation
                 is used as an escape character,
                 otherwise a fault condition        C
              # (two apostrophes form
                 the apostrophe-image) #
       FI  ;
       TRUE
  ELIF char = quote
  THEN
       get next character (char) ;
       IF   NOT eol  AND  char = quote
       THEN # quote-image #
            TRUE
       ELSE WHILE char = space
               DO  get next character (char)  OD  ;
            eol := FALSE ;
            IF   char = quote
            THEN # string-break, see HR 3.1 #
                 get next character (char) ;
                 TRUE
            ELSE # end of string #
                 FALSE
            FI
       FI
  ELSE
       TRUE
  FI

    DO
       denstring PLUSAB char
    OD  ;

CASE  1 + UPB denstring

  IN ( emptystring, estringdenproc )
   ,  ( denstring, chardenproc )

 OUT ( denstring, stringdenproc )

ESAC
```

```
ELIF   #1#

     REF INT  dyadnum  =  LOC INT ;
     char in string ( char, dyadnum, "%+-<=>/*" )

THEN   #1#

     # DOP-BECOMESETY-symbol (operator)
       and/or is-defined-as-symbol    #

     PROC (WORDPARAMS) VOID  opproc
        =   IF   dyadnum <= 3
            THEN taoproc  # operator could be monadic #
            ELSE tadproc  # operator must be dyadic #
            FI  ;

     REF STRING  opstring  =  LOC STRING  :=  char ;
     get next character (char) ;

     BOOL  colon2 =  char = ":" ,  equals2 =  char = "=" ;

     IF     #2#

          eol
           OR
          NOT
          (colon2 OR char in string(char, LOC INT, "<=>/*"))

     THEN   #2#

          # one character only  e.g.  "%"  or  "="  #

          ( opstring, IF opstring =  "=" THEN equalsproc
                                         ELSE opproc  FI )

     ELIF   #2#

          PROC WORD  colonequals
             = WORD :  IF
                            opstring PLUSAB ":"  ;
                            get next character (char) ;
                            eol  OR  char ≠ "="
                       THEN
                            C  emit a fault message
                               (ill formed operator)  C  ;
                            ( opstring,  badtaoproc )
                       ELSE
                            opstring PLUSAB "="  ;
                            get next character (char) ;
                            ( opstring,  opproc )
                       FI  ;

          colon2
```

```
THEN   #2#

     colonequals   # e.g.  "%:=" #

ELSE   #2#

     # second character not ":" #

     opstring PLUSAB char  ;
     get next character (char)  ;

     BOOL  colon3 =  char = ":" , equals3 =  char = "=" ;

     IF     #3#

          eol  OR  NOT(colon3 OR equals3)

     THEN   #3#

          IF
               equals2  # n.b. second character #
          THEN
               # have we a DYAD-cum-equals-symbol
                                               e.g. "<="
                 or
                 DYAD-symbol, is-defined-as-symbol  ?
                 Assume the second, think again when
                 the context is determined           #

               word held  :=  TRUE  ;
               held word  :=  ( "=" ,  equalsproc )  ;
               ( opstring[1] ,  opproc )
          ELSE
               ( opstring,  opproc )   # e.g.  "%<" #
          FI

     ELIF   #3#

          equals2  AND  colon3   # e.g.  "%=:" #

     THEN   #3#

          opstring PLUSAB ":"  ;
          get next character (char)  ;
          IF   char = "="
          THEN #  Assume DYAD-cum-assigns-to-symbol,
                  is-defined-as-symbol              #
               word held  :=  TRUE  ;
               held word  :=  ( "=" , equalsproc )
          FI  ;
          ( opstring,  opproc )

     ELIF   #3#

          colon3

     THEN   #3#

          colonequals   # e.g.  "%<:=" #
```

```
                ELSE   #3#

                     opstring PLUSAB "="  ;
                     get next character (char)  ;
                     IF
                          eol  OR  char ≠ ":"
                     THEN
                          # DYAD-cum-NOMAD-symbol,
                            is-defined-as-symbol,
                            e.g.  "%<", "="        #

                          word held  :=  TRUE  ;
                          held word  :=  ( "=" ,  equalsproc ) ;
                         ( opstring[1:2],  opproc )
                     ELSE
                          # DYAD-cum-NOMAD-cum-assigns-to-symbol,
                            e.g.  "%<=:"     #

                          opstring PLUSAB ":"  ;
                          get next character (char)  ;
                          ( opstring,  opproc )
                     FI

                FI     #3#

           FI     #2#

      ELIF   #1#

           char = ":"

      THEN   #1#

           get next character (char)  ;

           BOOL  eq  =  char = "=" ,  slash =  char = "/"  ;

           IF     #2#

                eol  OR  NOT(eq OR slash)

           THEN   #2#

                ( ":" ,  colonproc )

           ELIF   #2#

                eq

           THEN   #2#

                get next character (char)  ;
                IF
                     eol  OR  char ≠ ":"
                THEN
                     ( ":=" ,  becomesproc )
                ELSE
                     get next character (char)  ;
                     ( ":=:" ,  isproc )
                FI
```

```
        ELSE   #2#

              get next character (char)  ;
              IF
                   eol  OR  char ≠ "="
              THEN
                   C  emit a fault message
                      (ill formed isnt-symbol)  C  ;
                   ( ":/" ,  badisntproc )
              ELSE
                   get next character (char) ;
                   IF
                        eol  OR  char ≠ ":"
                   THEN
                        C  emit a fault message
                           (ill formed isnt-symbol)  C ;
                        ( ":/=" ,  badisntproc )
                   ELSE
                       get next character (char)  ;
                        ( ":/=:" ,  isntproc )
                   FI
              FI

        FI    #2#

   ELIF   #1#

        char = "|"

   THEN   #1#

        get next character (char)  ;

        IF
              eol  OR  char ≠ ":"
        THEN
              ( "|" ,  briefthinelseoutproc )
        ELSE
              get next character (char)  ;
              ( "|:" ,  briefelifouseproc )
        FI
```

```
        ELIF   #1#

             REF INT  i  =  LOC INT  ;
            char in string ( char, i,  "#$(),;@[]" )

        THEN   #1#

             get next character (char) ;

             [ : ]WORD (   ( "#" , hashcommentproc ) ,
                           ( "$" , formatterproc   ) ,
                           ( "(" , lparenproc      ) ,
                           ( ")" , rparenproc      ) ,
                           ( "," , andalsoproc     ) ,
                           ( ";" , goonproc        ) ,
                           ( "@" , atproc          ) ,
                           ( "[" , briefsubproc    ) ,
                           ( "]" , briefbusproc    )     ) [i]

        ELSE   #1#

             C  emit a fault message
                (impermissible character)  C  ;
             CHAR  c  =  char  ;
             get next character (char)  ;

             ( c ,  badcharproc )

        FI     #1#

    FI


COMMENT   end of PROC get word   COMMENT

COMMENT   end of token-recognizer algorithm   COMMENT
```

Appendix : Reserved Bold Words

(The algorithm assumes a well-behaved letter collating sequence)

AT, BEGIN, BITS, BOOL, BY, BYTES, CASE, CHANNEL, CHAR, CO, COMMENT, COMPL, DO, ELIF, ELSE, EMPTY, END, ESAC, EXIT, FALSE, FI, FILE, FLEX, FOR, FORMAT, FROM, GO, GOTO, HEAP, IF, IN, INT, IS, ISNT, LOC, LONG, MODE, NIL, OD, OF, OP, OUSE, OUT, PAR, PR, PRAGMAT, PRIO, PROC, REAL, REF, SEMA, SHORT, SKIP, STRING, STRUCT, THEN, TO, TRUE, UNION, VOID, WHILE

(Total : 61)

Epilogue

The author will be pleased to hear from anyone who has queries or finds mistakes, and will undertake to inform the Algol Bulletin and any individual correspondents of necessary amendments. Enquiries about the analogous Algol 68R program are also invited.

Please write to:

Mr. R. Bell,
Department of Computer Science,
Teesside Polytechnic,
Borough Road,
Middlesbrough,
Cleveland,
TS1 3BA,
England.

AB41.4.6

Some ALGOL 68 Compilers

During the recent ALGOL 68 Conference held in the University of Strathclyde (SIGPLAN Notices, Volume 12, Number 6, June 1977) the following information was gathered on the status of a number of ALGOL 68 compilers. The information was supplied by delegates and is to some extent second hand in the cases where the compiler writers themselves were not present at the conference. The conference organisers therefore cannot guarantee the accuracy of the information but thought it of sufficient interest to make it available to those interested in the implementation of ALGOL 68.

University of Strathclyde, Glasgow, Scotland
May 1977

R.B. Hunter
R. Kingslake
A.D. McGettrick

Language	Authors	Organisation	Start Date	Finish Date	Written in	Runs on	Remarks
A68B	Koch Oeters et al	Technical University Berlin	1974	1978?	CDL2	IBM 370	Full ALGOL 68 plus separate precompilation
A68C	S. Bourne M.J.T. Guy A.D. Birrell C.J. Cheney I. Walker	University of Cambridge	~1970	1977	A68C	IBM 360/370 ICL 4130 PDP 10/11 CAP	
A68C	Many	Ruhr University Bochum	1976	1977	A68C	TR 440	Transported from Cambridge
A68H	H. Boom	Mathematical Centre Amsterdam	1974	1978/9	ALGOL W	IBM 370	Produces JANUS for IBM 370
A68N	R.D. Knott	University of Nottingham	1976	1977/8/9	A68R	1900	Cross compiler for PDP 11
A68/RIAD	Tseytin Terekhov et al	Leningrad University	1971/72	1977	1. IBM Macro 2. A68 subset	RIAD 30	
A68RS	I.F. Currie et al	R.S.R.E. Malvern	1976	1977?	A68R	ICL 1900 2900 etc.	2 passes

AB41 p.71

A68S	P. Hibbard	University of Liverpool	?	1974?	Modular 1 Assembler	Modular 1	
A68S	M. Munro	University of Durham	?	1976	PL/360	IBM 370	
A68S	P. Hibbard	Carnegie-Mellon University Pittsburgh	1975	1976	BLISS	PDP 10/11	
A68S	C.H. Lindsey	University of Manchester	1976	1977	PASCAL	CYBER 72	Transliteration of P. Hibbard's BLISS
A68TUM	Hill Wossner et al	Technical University of Munich	1969?	1975?	TEXAS	TR4	Emulator for TR440 available
A68/19	Guy Louis R. Rijpens	Royal Military Academy Brussels	1971	1972	360 Assembler	IBM 360/30	
A68	C.H. Lindsey et al	University of Manchester	1971	?	A68R + CDL	ICL 1900 → MU5	Bootstrap from 1900, variable number of passes
A68	Many	Mathematical Centre Amsterdam	1972	1981	ALEPH	various	Machine independent Full language
A68	Cunin Delaunay Simonet Voiron	University of Grenoble	~ 1972	1976 not continued	CDL + LL(1) system + PL 360	IBM 360 CP/CMS-OS	Simple I/O Poor error recovery No parallelism
A68	JJFM Schlichting	CDC (CYBER)	1973	1976	SIMPLE + COMPASS	CYBER 70 SERIES	Full revised language Separate compilation

A68	D. Taupin	CNRS University of Paris XI	1973	1975	FORTRAN V	UNIVAC 11xx	8 passes As close as possible to Revised Report.
A68	R.B. Hunter R. Kingslake A.D. McGettrick	University of Strathclyde GLASGOW	1974	?	A68R	ICL 1900	Uses SID (LL(1))
A68	D.J. Morgan	University of Nottingham	1977	?	A68R	ICL 1900	
A68	Banatre et al	University of Rennes	?	1976	LP 70	CII 10070 IRIS 80	One pass Declare before use No <u>flex</u>
A68	Kral	Prague	?	1976		TESLA	
SERA	Pleyette et al	University of Rennes	?	1976	LP 70	CII 10070 IRIS 80	Teaching subset

AB41.5.1 **Errata**

The following errata appeared in the published version of 'A Supplement to the ALGOL 60 Revised Report' *(The Computer Journal)*, Vol, 19, 276-288.

1. Page 277, col 1, line 12: 'Level (IFIP)' should read 'Level 3(IFIP)'.
2. Page 280, section 4.2.4: '*entier* $(E + 0{\cdot}5)$' should read '*entier* $(E + 0{\cdot}5)$ where E is the value of the expression'.
3. Page 282, section 4.7.5.5: 'Add to this section' should read 'Replace this section by'.
4. Page 282, section 4.7.5.5: After 'string identifier' the following should appear (starting on a new line) 'If the actual parameter is itself a formal parameter the correspondence (as in the above table) must be with the specification of the immediate actual parameter rather than with the declaration of the ultimate actual parameter'.
5. Page 283, section 5.4.2: After the first sentence the following should appear: 'In procedure *Absmax* insert "**value** *n, m;*" before the specifications of formal parameters. After '$y := 0$;' insert '$i := k := 1$;'.'
6. Page 283, section 5.4.4: There should be no comma following 'If a function designator'.

AB41.5.2 Errata to the Revised Report 15 Mar 1977

The following corrections should be made to the Revised Report on the Algorithmic Language ALGOL 68, as published in the following editions:

Acta Informatica, Vol. 5, pts 1, 2 and 3, Dec 1975.
Springer-Verlag, 1976.
Mathematical Centre Tracts 50, Mathematisch Centrum, Amsterdam, 1976.

Misprints

p.108	8.0.1.a +2	# (94d)	=>	#
p.110	8.1.4.1.d	# item	=> item'	#
p.116	9.3.c +5	# *BEGIN*	=> *BEGIN*,	#
p.118	9.4.1.b+13	# $_{10}$	=> $_{10}$	#
p.132	10.2.3.4.a	# *a)*	=> a)	#
p.173	10.3.4.1.1.A −4	remove spurious line		
p.194	10.3.5.h +15	# *(f));*	=> *(f))*	#
p.196	10.3.5.1.a "*edit L real*" −2	# ***L 0***	=> ***L 0***	#
p.197	10.3.5.1.a "*edit L compl*" −8	# *a;*	=> *a);*	#
p.199	10.3.5.1.a "*gpattern*" +13	# ***L* int** *i):* *i*⊣, ⊣ ***L* real** *r):*	=> *(**L* int** *i):* *i*⊣, ⊣*(**L* real** *r):*	#
p.201	10.3.5.2.a +3	# **in**	=> *x*[*k*] **in**	#
p.207	10.3.6.2.a "**case** *y*[*j*]" +1	# *from*	=> *(from*	#
	10.3.6.2.a "**case** *y*[*j*]" +9	# *from*	=> *(from*	#