stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 61/76 SEPTEMBER
(DEPARTMENT OF COMPUTER SCIENCE)

J.C. VAN VLIET

ON THE ALGOL 68 TRANSPUT CONVERSION ROUTINES

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

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

AMS(MOS) subject classification scheme (1970): 68A10, 68A15

ACM-Computing Reviews-categories: 4.41

On the ALGOL 68 transput conversion routines [*)]

by

J.C. van Vliet

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.

KEY WORDS & PHRASES: ALGOL 68, *transput*, *conversion*.

---

*) This paper is submitted for publication elsewhere.

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 routine-text occurs whose calling involves the manipulation of real numbers, then this routine-text may be replaced by any other routine-text 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 real and int, 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.

ERRORS AND PROBLEMS

The following errors have been found in section 10.3.2.1. of the Report:

1. In *fixed*, a number less than *1* is, if possible, converted to a string starting with *"0."*. However, the test whether *"0"* should be placed in front of the string is performed on the non-rounded number. After rounding, things may be different. For example, *fixed(0.99, 5, 2)* will yield *"01.00"* instead of *"+1.00"*.

2. The routine *float* starts with the computation of the number of digits that have to be placed before the decimal point. This computation is not correct, since it does not cater for the situation in which the user wants to suppress the plus-sign. Instead of

   int *before* = **abs** *width* − **abs** *exp* − *(after* ≠ *0* | *after* + *1* | *0)* − *2;*

   it should read

   int *before* = **abs** *width* − **abs** *exp* − *(after* ≠ *0* | *after* + *1* | *0)* −
   (x < **L** *0* ∨ *width* > *0* | *2* | *1);*

   For otherwise, in case *x* ≥ **L** *0* and *width* < *0*, the resulting string starts with *"+"*, which is never intended when *width* < *0*.

3. The statement

   *s:= fixed(sign x * y, ...);*

   in *float* is illegal. The two operands of * are of type **int** and **L real**, respectively. The statement might read:

   *s:= fixed(***K sign** *x * y, ...);*

4. Errors in *string to L real*: The line

   for *i* **from** *j* + *1* **to** *e* − *1* **while** *length* < **L** *real width*

   should read

   for *i* **from** *j* + *1* **to** *e* − *1* **while** *length* < *L real width.*

   A few lines further down, **L** *max real* should be *L max real* twice. Also, the exponent should be converted using *string to int* instead of *string to L int*.

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*:

  ```
  while y + L .5 * L .1 ↑ after ≥ L 10 ↑ length do length +:= 1 od;
  ```

  assumes that integers may take on the same values as reals, for *L 10 ↑ length* has mode L int. This may well not be the case, thus yielding an integer overflow. Presumably, the intention has been to write *L 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 ≥ L 10.0 ↑ before do before +:= 1 od;
  ```

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

  ```
  while y / L 10.0 ≥ L 10.0 ↑ (before - 1) do before +:= 1 od; ,
  ```

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

  ```
  while y ≥ L 1.0 do y /:= L 10.0; before +:= 1 od;
  ```

  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

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

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 *expace* +:= *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, exspace := 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);
```

```
        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) * "." + 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.