stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IN 7/74 APRIL

D. GRUNE, L.G.L.T. MEERTENS, J.C. van VLIET & R. van VLIET
REMARKS ON THE TRANSPUT SECTION OF THE REVISED REPORT ON ALGOL 68

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): 68A15
ACM - Computing Reviews - category: 4.41, 5.24

Remarks on the transput section of the revised report on ALGOL 68

by

D. Grune, L.G.L.T. Meertens, J.C. van Vliet & R. van Vliet

ABSTRACT

This document is the deposit of a thorough study of the transput section of the Revised Report on the Algorithmic Language ALGOL 68. It is especially concerned with the topics 'positioning', 'lay-out routines' and 'mending'. This report contains, amongst others, a complete set of routines on formatless transput. The document is intended to be discussed at the meeting of the WG 2.1. Standing Subcommittee on ALGOL 68 Support, April 7-10, 1974 at Cambridge (England).

During the past months, the ALGOL 68 group at the Mathematical Centre have devoted the larger part of their time to studying the transput section of the Report. This has resulted in the unearthing of various phenomena, ranging from mildly surprising to severely shocking. Many of these were concerned with insufficient checking after a call of an event routine. Since then, a series of changes to the transput routines have been made or proposed, intended to remove the worst of these phenomena. In our opinion, this objective has only partially been reached; although the checking is performed much more thoroughly now (to some extent at the cost of the readability of the routines concerned), the reaction if these checks fail is often still surprising. In part A of the present document, this point is elaborated upon. Other remarks are given in parts B, C and D.

Since it was not clear to us how some of the objectionable situations could be properly mended within the present framework of routines, (which are complicated enough as they stand,) we have worked out our suggestions by designing ourselves a new set of routines for checking, positioning and mending, whose use is illustrated in put, get, put bin and get bin. The time has failed us to try and bring this set again in line with the present set of routines, although there seems to be some functional similarity between, e.g., our ensure logical file and the newly proposed check logical pos, or our ensure page and the newly proposed get good file. (We have not studied in detail the formatted transput routines either.)

Parts E, F and G contain, respectively, a summary of some properties in which our routines differ from the present or proposed system, the text of the routines, and a list of pre- and postconditions associated with the routines that may help in ascertaining their correctness. We hope that our effort may contribute to bringing the transput section into good shape.

A. To loop, or not to loop?

In this section it is argued that, in order to allow the user to supply a reasonable system of event routines, the following requirement should be met:

> If the user, through his mending, commits a crime different from the one that caused the event routine to be called, then this new crime will be signalled also by the calling of an event routine.

Examples are given where the present system of transput routines fails in this respect.

Consider the following situation:
. a transput routine is called;
. in the sequence of tests preceding the transput action, an "event" is detected;
. the corresponding event routine is called and returns true;
. again an event (possibly the same one) is detected.

The question is, what reaction to choose in this situation:
a) to call again an event routine, and so on, with the risk of a possibly infinite loop, or
b) to call undefined, even though the user's mending was perhaps quite reasonable and would not have caused a loop.

It appears that we have conflicting desires, on the one hand the desire to prevent loops, on the other hand the desire to allow any reasonable way of mending. The AB version of the transput routines contains some compromise between these two desires. For example, the following cases may cause loops (cf. the thesis by R. G. Fisker):

```
. proc mend char error = (ref file f, ref char sugg) bool:
  (sugg:= some unconvertible character; true)
  when called from put char;
. proc mend page = (ref file f) bool:
  (set(f, 1, 1, 81) ¢ line ended ∧ ¬ page ended ¢; true),
  proc mend line = (ref file f) bool:
  (set(f, 1, 61, 1) ¢ page ended ¢; true)
  when called from get good line/page;
. proc mend line = (ref file f) bool:
  (backspace(f); true)
  when called from skip spaces in get.
```

In many other cases, loops are prevented. The philosophy used is: "loops may only arise if the user, through his event routine, commits a different crime and returns true". Although it is a matter of taste whether in all three above cases the crimes committed are indeed different, this seems a sensible requirement. Nevertheless, the wish to allow any reasonable way of mending still remains. In a letter to the editors, the ALGOL 68 group of the Mathematical Centre have criticized the AB version of the transput routines in this respect. A particularly unpleasant consequence was the failure to accept as user mending for line end a call of newline (nota bene, the default mending), when line end was detected in get and another empty line happened to follow in the file. As a reaction to this criticism, other versions of the routines get good line and get good page have been proposed, in a document labelled RGF/CHL 74-03-22.

It appears to us, however, that the solution suggested there is rather ad hoc: further calls of event routines are allowed only if either a different file has been assigned to the file variable parameter, or if the current position has been advanced to at least the next line (page). Suppose now that a user wants to use the first 15 lines of his file as a circular buffer, and that he supplies

```
proc mend line = (ref file f) bool:
(if line number(f) = 15 then reset else new line fi(f); true),
```

or consider another user who, for reasons clear to him if not to us, wishes to process his file (of one page) line by line backwards, and therefore supplies

```
proc mend line = (ref file f) bool:
(set(f, 1, line number(f) - 1, 1); true).
```

In either case, the proposed changes to get good line do not suffice to help the user. Even in the case where the event routine closely mimics the default mending, as in

```
proc mend line = (ref file f) bool:
(if char number(f) > 0 then new line(f) else
set char number(f, 1) fi; true),
```

a call of get may still result in undefined being called.

In the system of routines described in the present document, a prior version of which has been circulated among the editors, loops can always arise when the event routines return true, even if no attempt whatsoever has been made to mend, the rationale behind this liberal (or unpaternalistic) attitude being, that this is the safest way to guarantee that no reasonable way of mending is excluded, that loops cannot be prevented in general anyway, and that the user who writes an event routine returning true better should know what he is doing. If, however, a system is preferred where the user is protected against himself, then this should under the least possible number of circumstances hinder the user who supplies a sophisticated set of event routines that would not loop in the first place. Such a system should be centred on the notion of different crime. In order to sketch how this may be described, we show the application to our routine ensure line.

```
op ? // = (bool ok, ref bool was ok) bool:
(ok | true |: was ok | was ok:= false | undefined; skip),

prio // = 2;

proc ? ensure line = (...) bool:
begin bool result, wok1, wok2, ref pos ocpos:= nil,
      int op:= 0, ol:= 0;
   while
      if result:= ensure page(f, s, test, 0)
      then
         if ref pos cpos = current pos(f);
            int p = p of cpos, l = l of cpos;
           ((cpos :≠: ocpos) ∨ p ≠ op ∨ l ≠ ol
           | wok1:= wok2:= true; ocpos:= cpos; op:= p; ol:= l);
           (1 - lmarg ≤ l // wok1) ∧ (l ≤ line bound(f) // wok2)
         then
...
```

This approach is, of course, more complicated than either of the solutions to loop always and not to loop ever, but it seems the only reasonable way to prevent the most striking kind of loops. The user does, in general, not know under what exact circumstances his event routines will be called, nor can he find out inside the event routines. He can only do his best to mend the situation and trust that if, in doing so, he commits a different, though possibly similar crime, this will be signalled to him in due time.

Under the present system, the user who cherishes such trust is in for a few disappointments, apart from those mentioned above. Consider, e.g.,

```
proc mend physical file = (ref file f) bool:
(close(f); open(f, chain:= next of chain, channel(f)); true).
```

If the newly opened file happens, by accident, to be empty, then undefined is called, instead of again physical file mended. The user who tries to mend his logical file end by

```
proc mend logical file = (ref file f) bool:
(int p = page number(f), l = line number(f), c = char number(f);
compute(next bunch); put(f, next bunch);
set(f, p, l, c))
```

may expect that this will work fine, but then he may be mistaken also. If, e.g., put(f, next bunch) starts with newline or newpage, and f is compressible, then the continuation in, e.g., get char, will detect line end and call undefined. Or logical file end was detected from newline on an empty line and the next bunch starts with newpage so that the current position is left dangling two lines below the page end and the user will presumably go on reading the very line that he intended to skip - in any case, that he would have skipped, had put(f, next bunch) preceded the call of logical file mended.

B. Brow movers

In this part, some examples are given that may shock or surprise the user.

1. re: newpage/newline in RGF/CHL 74-03-22.

a) Suppose l of book bounds = 60, lpos = (1, 61, 1),
cpos = (1, 61, 0).
A call of get(f, new page) causes lpos to be moved to (2, 1, 1).

b) Suppose l of book bounds = 60, lpos = (1, 61, 1),
cpos = (1, 30, 10), f is a compressible file.
A call of put(f, new line) or put(f, new page) causes transgression of bounds in text[1][61].

c) Suppose lpos is inside the current page of a compressible file, at (1, 30, 40), say.
A call of put(f, new page) causes logical file mended to be called, with an infinite loop if the yield of that calling is true, and a call of undefined otherwise.

2. complementarity of input/output

a) Suppose l of book bounds = 3.

```
begin string r1:= "ab", r2:= "c", r3:= "", r4:= "def", q:= "?";
      make term(f, q);
      reset(f); put(f, (newline, r1, q, newline, r2, q,
       newline, r3, q, newline, r4, q));
      reset(f); get(f, (newline, r1, q, newline, r2, q,
       newline, r3, q, newline, r4, q))
end
```

This may well cause a call of logical file mended. The values assigned to the r's are:
r1:= "ab", r2:= "c", r3:= "def", r4:= skip.
Explanation: When r3 is written, the page is ended. Since r3 yields an empty string, the net effect is nihil. When r3 is to be read, however, the page end condition is mended by default, so that r3 receives the string written as r4. At the end of the physical file, it is similarly possible to write an empty string, but an attempt to read it back will cause a call of physical file mended. In our routines, this problem has been solved partially by requiring that even an empty string is put on a valid line. This solution is only partial, since on a compressible file a call of new page may cause a previously valid line to cease its existence. A full solution is only possible by introducing a flag line empty, which, if true, indicates that the line containing lpos may disappear if c of lpos = 1.
A side effect of requiring that the last line is valid would be to set line empty to false, and new page may only cut off the last line if line empty ∧ c of lpos = 1.

b) Suppose we have a flex[1 : n] char tape1, containing, among others, special characters new line char and new page char. We write this tape to a compressible file, as follows:

```
for i to n
do put(f,
    (char c = tape1[i]; c = new page char | new page
    |: c = new line char | new line
    | c))
od.
```

In order to get an image of the result (possibly intended for paper tape), we take into account the effect of line and page end, and the fact that a call of new page may imply a call of new line:

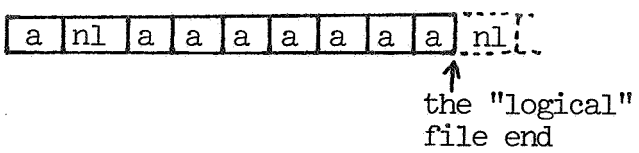
```
[1 : large] char tape2; int m:= 0; char last:= new page char;
proc store = (char c) void:
begin if c = new page char ∧ last ≠ new page char ∧
        last ≠ new line char then
          store(new line char)
      fi;
      tape2[m +:= 1]:= last:= c
end;
on line end(f, (ref file f) void: (store(new line char); false);
on page end(f, (ref file f) void: (store(new page char); false);
for i to n
do put(f,
    (char c = tape1[i]; store(c); c = new page char | new page
    |: c = new line char | new line
    | c))
od.
```

This process is idempotent in the sense that if we repeat the process after tape1:= tape2[1 : m], the result in tape2 is still the same.

We now expect to get back the same image also by

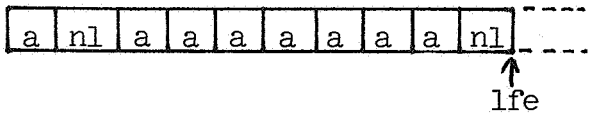
```
m:= 0; last:= new page char;
on logical file end(f, (ref file f) void: finish);
reset(f);
do char cc;
   get(f, cc); store(cc)
od; finish: skip
```

If, however, the last character output was on the last position of a line, then the new image tape2 will contain one new line char more, even though on output this new line has never, explicitly or implicitly, been given or detected. The reason is that line end is detected before logical file end, even if the logical file end is still on the same line. To use a picture, where nl stands for new line and a stands for any character:



the "logical" file end

The dotted square should, in read mood, effectively not exist, being beyond the logical file end. Under the present system, this situation is virtually undistinguishable from the one with



It appears to us that, in order to warrant complementary behaviour on input, the logical file end detection should take place before physical file/page/line end detection. This has been implemented in our system of routines.

c) Suppose that the flip and flop character are "1" and "0", respectively.
If we have

```
int i:= 13; bits b:= bin 13,
```

then put(f, (i, b)) results in something like

```
"...+130000001101"
```

so that get(f, (i, b)) will not work.
Now this is clearly a shame. Apart from the ref string-case, which is an odd man out, there is no reason why input and output should not be complementary in get and put. At present, an additional "." is inserted before a numeric value when not at the beginning of a line. This should be: an additional "." after a numeric value when not at the end of a line. As long as only numeric values are transput, the difference is not visible, but as soon as numeric and other values are interleaved, the difference becomes apparent - to the benefit of the user. Of course, this implies that in get, after a numeric value has been read, an additional ?"." has to be performed once.
This change is both so minor and simple to make and of such great practical value, that we strongly advocate that it be taken into consideration.

3. Premature decision in read sign.

Suppose we have a circular buffer containing signed integers. The event routine corresponding to on logical file end is: reset(f); true. Eventually, the logical file end is reached and an attempt is made to read the next integer. The course of events is as follows:

- the routine skip spaces is called and detects that no spaces are present;
- the routine read sign is called, which detects that no sign is present, so a default "+" is delivered;
- the routine read dig is called, resulting in a call of get char(via ↓) to read the first digit;
- logical file mended will be called, whereupon the file is reset, and get char might well yield a "+", "-" or ".";
- char error mended is called with a "0" as a suggestion to replace the erroneous character, which is a strange result.

In our system, the above is prevented in a natural way.

4. Opening and closing files.

a) How to write to a book whose field putting is false.

Suppose we have two channels, A and B, via which the same book may be linked(present twice in the chainbfile).
(put of A)(book) = true and (put of B)(book) = false.
Consider the following:

```
file f1, f2;
open(f1, "", B);    ¢ putting = false ¢
open(f2, "", A);    ¢ putting = true, but still, reading
                      via f1 is possible ¢
close(f1);          ¢ putting = false, but writing is
                      still possible via f2 ¢
```

We think it appropriate not to allow a book to be open for reading and writing simultaneously in a defined way. To accomodate this, we propose to introduce an integral field users which counts the number of times the book is opened. This field may then be inspected in the routine open(10.3.1.4.d).

```
10.3.1.1.bb - 1 ✖ for writing ??? simultaneously. ->
                  if it has been opened for writing.
                  Moreover, a book contains an integral field
                  users which counts the number of times
                  the book is opened. A book cannot be
                  opened for writing as long as it is
                  opened on some other file. ✖;
10.3.1.1.a + 3 ✖ ) -> , int users ¢ the number of times
                  the book is opened ¢) ✖.
```

Further consequences are to be found in the proposed versions of open, close, etc.

b) Relinquished books.

{And Tom Thumb sprinkled crumbs.}

At range exit, files declared within the range being left, are simply lost. If they were open, possibly no way of referencing their books remains. What happens with those books sprinkled on the heap? Obviously, the Report cannot specify this. A similar situation arises when one of the routines open, establish and create is called with an open file as parameter. Again, a reference to a book is in imminent danger of being lost. In this case, however, the Report suggests that the file first be closed (thus closing all copies of the file as well). Would not it be better to leave this situation unspecified as well?

c) Pointless protection.

Upon closing, locking or scratching a file, the book and the text of the book are set to nil, presumably in order to protect the book against illegal access. However:

a. this protection is superfluous, since none of the transput routines acts on a file whose field opened returns false;
b. this protection is incomplete, since possible copies of the file being closed, locked or scratched still have their fields referring to this book and this text.

Consequently, we omitted this attempt from our versions of close, lock and scratch.

d) Standout and standback.

For standout and standback, we have the following two desiderata:

a. the corresponding books should be empty at the start of the program;
b. they ought to be in write mood by default.

Therefore, both standout and standback should be created, instead of opened.

e) Default read mood for newly opened files.

In most situations, a file is either in write mood, or in read mood. If both yield false, the layout routines act as if the file is in write mood. Obviously, the normal thing after opening a file for which get possible is true, is to read what has been written to the book of that file previously. Now, a user may find out to his surprise that he is writing on such a file, even though he has never explicitly indicated so. A default set read mood in the routine open is therefore suggested, and has been implemented in our system. (Nota bene, problems arise if our remarks on alternating reading and writing in binary transput, and creating standout and standback, are rejected; e.g., binary output on standback would become illegal.)

f) How to reset a file if ¬ reset possible.

Consider the following:

```
file f; open(f, "", ch); .... ; close(f); open(f, "", ch);
                                  implicit resetting.
```

This may be hard to implement. A possible solution is to call lock if close is called and reset possible returns false.

g) Unwanted scope error.

If an associated file is closed or locked, a scope error results in the present system, thus preventing associated files from being chained in a chain of bfiles. We prefer either to act as if the routine scratch was called (implemented in our system) or to call undefined directly.

C. Binary transput.

1. Alternation of char and bin mood.

   At present, alternation of char and bin mood is allowed only for random access files. At first glance, this restriction seems sensible enough, since the operating system might have difficulty in performing binary transput other than in chuncks of "words", corresponding to several c-positions, starting and ending on word boundaries. On second thought, the operating system should be able to cope with such situations, because the c of cpos may be shifted by an arbitrary amount by means of space, backspace and set char number. Therefore, we suggest that one of two courses be taken:
   Either make alternation of char and bin mood on sequential access files defined, or
   make calling space, backspace or set char number, and possibly new line and new page, on sequential access files in bin mood cause undefined to be called.

2. Alternation of read and write mood.

   During character transput, reading and writing may be alternated arbitrarily on random access files as well as on sequential access files. During binary transput, however, this alternation is not permitted on sequential access files. Since we could not find a reason for this restriction, we left it out in our versions of put bin and get bin (or, more precise, set write mood and set read mood).

D. Miscellaneous.

1. At present, get, when reading a numeric value, will skip spaces in the following cases:
   i) before the first character;
   ii) after a sign;
   iii) after "$_{10}$" or "e";
   iv) before "i" or "|";
   v) after "i" or "$\underline{|}$".

   In all of these cases, transitions to a new line or page may occur, except in case ii. This seems unnecessarily difficult to teach. Therefore, we changed the routine read sign to allow such transitions in case ii also.

2. Should not a pseudo-comment in close and reset take into account that in some systems the closing or resetting of a file might imply something like writing a new line or new page, e.g., because an end of record marker is written before the end of file marker, which might be the way for a compressible file to indicate new line also?

3. To increase clarity, one could add to 10.3.1.3.aa:

   The routines, described in section 10.3 of this Report, are never called with a file as parameter, but only with a value of the mode specified by ref file. To some of the subnames of such a value, new values may be assigned by means of the routines provided (such as make conv, which is used to assign a new conv field (10.3.1.3.dd)). The other subnames (such as the one selected by cpos) are only assigned to by routines for opening and closing files (10.3.1.4); all of these subnames (except the one selected by chan) refer to fields which are names. As a consequence, when a copy is made of an open file, the fields referred to by corresponding subnames of this group will be the same name. An assignment to such a field (to the field itself, not to the subname referring to the field), will thus affect all copies of a given file in a like way.

4. For reasons of clarity:

   10.3.1.3.bb + 10 ✗ . -✗ .
   In general, read mood, write mood, char mood and bin mood characterize the last transput action performed on the file. However, after "establishing" (10.3.1.4.cc), opening and "resetting" (10.3.1.6.j) a file, default values are returned. ✗.

5. 10.3.1.1.aa + 2 ✗ may contain -> refers to a "text", i.e., ✗;
   10.3.1.1.aa + 4 ✗ pages -> "pages" ✗;
   10.3.1.1.aa + 4 ✗ lines -> "lines" ✗;
   10.3.1.2.aa + 4 ✗ whose fields are -> comprising fields which are ✗;
   10.3.1.2.bb - 4 ✗ yields -> returns ✗;
   10.3.1.2.bb + 2 ✗ in a book -> in the text of a book ✗;

6. As section 10.3.1.3.cc is intended to clarify the way in which event routines may be used, do not emphasize the problem of range structure that much, only lightly touch upon it. We propose:

```
10.3.1.3.cc + 19:33 ✖ If he writes ??? end. -> He has to write:
                      begin int n:= 0; file auxin:= intape; ¢ this
                         is necessary, since the scope of the
                         routine (ref file file) bool: goto f
                         is smaller than the scope of intape ¢
                         on logical file end(auxin, (ref file file)
                         bool: goto f);
                         do get(auxin, loc int); n +:= 1 od;
                         f: print(n)
                      end. ✖;
```

7. In the example after 10.3.1.3.hh, some errors have been made:

   i. The variable n is not initialized, so:

```
10.3.1.3.hh + 5 ✖ int n -> int n:= 0 ✖;
```

   ii. n should be increased only after a number has been read succesfully;

   iii. The program will call undefined since the event routine returns true without mending the logical file end. The simplest remedy seems to be:

```
10.3.1.3.hh + 14 ✖ bool ???; -> ✖;
10.3.1.3.hh + 16 ✖ (???)) -> goto end) ✖;
10.3.1.3.hh + 17 ✖ for ??? od -> for i do get(f1, x[i]); n:= i od ✖;
10.3.1.3.hh + 19 ✖ reset -> end: reset ✖;
```

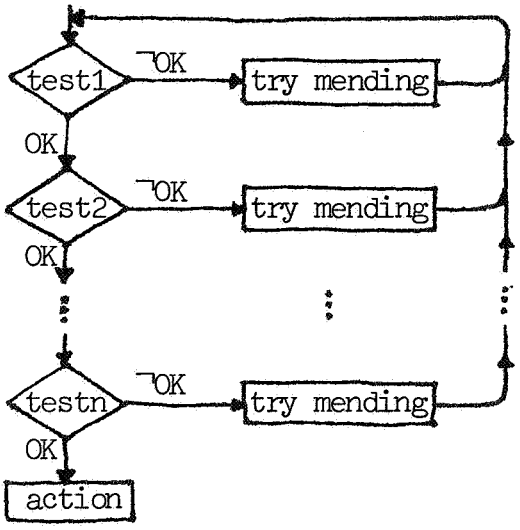
8. The chain of backfiles may also be reorganized after a call of lock, so:

```
10.4.2 + 10 ✖ closed -> closed or locked ✖;
```

E. Some properties of the routines

1. Checks on events

If the "requirement" for the satisfactory execution of a transput action is of the form test1 ∧ test2 ∧ ... ∧ testn, then this is implemented as:



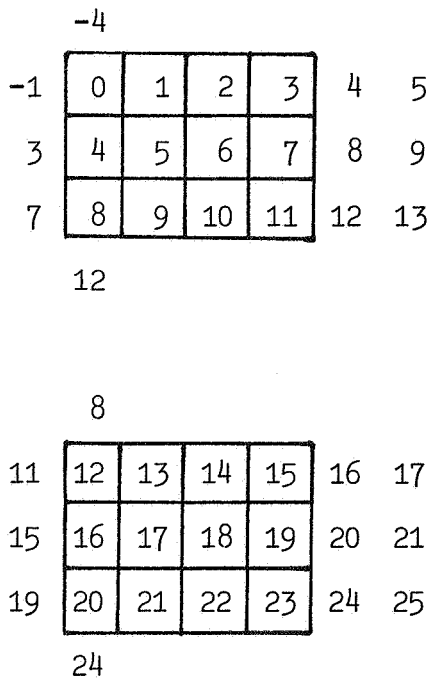
(See also part A.)
The tests are performed in a fixed order in each case, determined by

- opened (no mending possible);
- state OK (no mending possible);
- logical file OK (see B.2.b);
- p of cpos OK;
- l of cpos OK;
- c of cpos OK.

Of course, only those tests are performed that occur in the requirement at hand. (For set and set char number, no repetition of tests is made if the new cpos would be beyond lpos. They could easily be brought in the same framework as the other routines, but we have not attempted to do so.)
All checks are performed explicitly, so that implicit tests (as in the present book bounds, where the test catches some nasty cases,) are no longer needed.

2. Default mending

In all cases, the default reaction to be taken if the event routine returns false is determined before the event routine is called. Consider a book with block shaped text. We can "linearize" the text by the function
L(p, l, c) = ((p - 1) × line bound + (l - 1) × char bound + c - 1,
so that we get, e.g.,



Now, if the current position is (p, l, c), and a default reaction is chosen which would bring cpos to (p', l', c'), then L(p, l, c) = L(p', l', c'). If no such choice is possible, then undefined is chosen (see 4). There is an exception to this if l = page bound + 1 and c ≠ 1; such a position may, however, only be reached by means of set.

3. The moods

For an open file, read mood ≠ write mood. For a file which is opened on a channel such that get possible, the default is read mood (see B.4.e). Alternation of read/write mood in bin mood is possible on sequential files also (see C.2). The state when a transput action is performed or an event routine is called, is the same as at or specified by the call. For example, put(f, (a, b, c)) has the same meaning as put(f, a); put(f, b); put(f, c); Similarly, get(f, newline) cannot cause spaces to be written by newline, nor can put(f, newline) fail because the logical file end was encountered by newline, or leave part of the text undefined since a switch to bin mood occurred. However, if write mood and ¬ bin mood ∧ ¬ char mood, and space (or, on an incompressible file, newline or newpage) encounter the logical file end, then space characters are written and char mood is set to true.

4. The layout routines

If the transformation of cpos by space, backspace, newline or newpage is denoted by (p, l, c) -> (p', l', c'), then the requirement (see 1) corresponding to this amounts to: (p, l, c) or (p', l', c') is inside the text of the book. This, together with 2, implies, e.g., that if the elaboration of get(f, loc char) is

defined and does not evoke user mending, then get(f, space) transforms cpos in the same way. The routine set char number requires validity of the current line ($1 \leq l \leq$ line bound), so that set char number(f, char number(f) - 1) behaves similar to backspace(f).

5. Put and get

If upb numeric to string(numeric) > char bound, then put(f, numeric) calls undefined without first giving a newline. A valid position ($1 \leq l \leq$ line bound, $1 \leq c \leq$ char bound + 1) is required by put(f, "") (see B.2.a). In get, next pos requires ¬ logical file ended (see B.3).

F. The text of the routines

Opening and closing.

```
¢ the routines undefined, file available, idf ok, match and
  false are omitted here ¢

a) proc establish = (ref file file,
        string idf, channel chan, int p, l, c) int:
     begin down bfileprotect;
       PRIM book book:=
         (PRIM flex [1 : p] flex [1 : l] flex [1 : c] char,
          (1, 1, 1), idf, true, 1);
       if file available(chan) ∧ (put of chan)(book) ∧
         estab of chan ∧ ¬ (pos(p, l, c) beyond max pos of chan) ∧
         ¬ (pos(1, 1, 1) beyond pos(p, l, c)) ∧ idf ok(idf)
       then (opened of file | up gremlins | up bfileprotect);
         file:= clean file; book of file:= book;
         text of file:= text of book; chan of file:= chan;
         conv of file:= (standconv of chan)(book);
         ( ¬ bin possible(file) | set char mood(file));
         set write mood(file);
         0
       else up bfileprotect; undefined
       fi
     end;

b) proc create = (ref file file, channel chan) int:
     begin pos max pos = max pos of chan;
       establish(file, skip, chan, p of max pos, l of max pos,
       c of max pos)
     end;

c) proc open = (ref file file, string idf, channel chan) int:
     begin down bfileprotect;
       if file available(chan)
       then ref ref bfile bf:= chainbfile; bool found:= false;
         while (ref bfile (bf) :≠: nil) ∧ ¬ found
         do
           if match(idf, chan, book of bf)
           then found:= true
           else bf:= next of bf
           fi
         od;
         if ¬ found
         then up bfileprotect; undefined
         else ref book book:= book of bf;
           if putting of book ∨ (put of chan)(book) ∧
             users of book > 0
           then up bfileprotect; undefined
             ¢ in this case the book was already open for putting
               or someone was reading and putting is requested
               the system may either wait, or yield nonzero
               (indicating unsuccessful opening) immediately ¢
           else ((put of chan)(book) | putting of book:= true);
             users of book +:= 1; ref ref bfile (bf):= next of bf;
               ¢ remove bfile from chain ¢
```

```
            (opened of file | up gremlins | up bfileprotect);
            file:= clean file; book of file:= book;
            text of file:= text of book; chan of file:= chan;
            conv of file:= (standconv of chan)(book);
            set defaults(file);
            0
          fi
        fi
      else up bfileprotect; undefined
      fi
    end;
```

d)
```
  proc associate = (ref file file, ref [][][] char sss) void:
    begin proc t = (ref book a) bool: true;
      proc f = (ref book a) bool: false;
      channel chan = (t, t, t, t, f, f, f, bool: false,
        pos: (max int, max int, max int), skip, skip);
      (opened of file | down bfileprotect; up gremlins);
      file:= clean file;
      book of file:= heap book:= (skip, (upb sss + 1, 1, 1),
        skip, true, 1);
      text of file:= sss; chan of file:= chan
    end;
```

e)
```
  proc ? set char mood = (ref file f) void:
    if ¬ char mood of f
    then if ¬ bin mood of f ∨ set possible(f)
      then ref bool (char mood of f):= true;
        ref bool (bin mood of f):= false
      else undefined
      fi
    fi;
```

f)
```
  proc ? set bin mood = (ref file f) void:
    if ¬ bin mood of f
    then if bin possible(f) ∧ ( ¬ char mood of f ∨ set possible(f))
      then ref bool (char mood of f):= false;
        ref bool (bin mood of f):= true
      else undefined
      fi
    fi;
```

g)
```
  proc ? set read mood = (ref file f) void:
    if ¬ read mood of f
    then if get possible(f)
      then ref bool (read mood of f):= true;
        ref bo ol (write mood of f):= false
      else undefined
      fi
    fi;
```

```
h) proc ? set write mood = (ref file f) void:
     if ¬ write mood of f
     then if put possible(f)
       then ref bool (read mood of f):= false;
         ref bool (write mood of f):= true
       else undefined
       fi
     fi;

i) proc clean file = file:
     (skip, skip, skip, skip, skip,
      heap bool:= false, heap bool:= false, heap bool:= false,
      heap bool:= false, heap bool:= true,
      heap pos:= (1, 1, 1), "", skip,
      false, false, false, false, false, false,
      (ref file f, ref char a) bool: false);

j) proc set defaults = (ref file f) void:
     begin
       if get possible(f) then set read mood(f)
       elif put possible(f) then set write mood(f)
       else undefined
       fi;
       ref bool (bin mood of f):= false;
       ref bool (char mood of f):= ¬ bin possible(f);
       current pos(f):= (1, 1, 1)
     end;

k) proc close = (ref file f) void: disconnect(f, chainbfile);

l) proc lock = (ref file f) void: disconnect(f, lockedbfile);

m) proc scratch = (ref file f) void: disconnect(f, loc ref bfile:= nil);

n) proc ? disconnect = (ref file f, ref ref bfile bf) void:
     begin ensure open(f); ref bool (opened of f):= false;
       putting of book of f:= false; users of book of f -:= 1;
         c lpos may be set to a suitable value, e.g., the beginning
           of the next line, page or at the end of the physical
           file, inserting blanks or spaces if required c;
       case text of f in
         (flextext):
           down bfileprotect; bf:= PRIM bfile:= (book, bf);
           up gremlins
       esac
     end;
```

Position enquiries.

```
a) proc char number = (ref file f) int:
     (ensure open(f); c of current pos(f));

b) proc line number = (ref file f) int:
     (ensure open(f); l of current pos(f));

c) proc page number = (ref file f) int:
     (ensure open(f); p of current pos(f));

d) proc ? current pos = (ref file f) ref pos: cpos of f;

e) proc ? logical pos = (ref file f) ref pos: lpos of book of f;

f) mode ? state = struct(bool char mood, bin mood, read mood);

g) proc ? state = (ref file f) state:
     (char mood of f, bin mood of f, read mood of f);

h) proc ? page bound = (ref file f) int:
     upb [][][] char (text of f | (text t1): t1, (flextext t2): t2);

i) proc ? line bound = (ref file f) int:
     ¢ will only be called within a page specified by a valid
       page number ¢
     upb [][][] char (text of f | (text t1): t1, (flextext t2): t2)
       [p of current pos(f)];

j) proc ? char bound = (ref file f) int:
     ¢ will only be called within a line specified by a valid
       line number within a page specified by a valid page number ¢
     upb [][][] char (text of f | (text t1): t1, (flextext t2): t2)
       [p of current pos(f)][l of current pos(f)];

k) proc ? check logical pos = (ref file f) void:
     if ref pos cpos = current pos(f), pos lpos = logical pos(f);
       cpos beyond lpos
     then cpos:= lpos;
       ( ¬ (logical file mended of f)(f) | undefined)
     fi;

l) proc ? logical file ended = (ref file f) bool:
     ¢ will only be called with ¬ (cpos beyond lpos) ¢
     begin pos cpos = current pos(f), lpos = logical pos(f);
       p of cpos = p of lpos ∧ l of cpos = l of lpos ∧
       c of cpos = c of lpos
     end;


m) proc ? page ended = (ref file f) bool:
     l of current pos(f) = line bound(f) + 1;

n) proc ? line ended = (ref file f) bool:
     c of current pos(f) = char bound(f) + 1;
```

Ensure routines.

```
a) proc ? ensure open = (ref file f) void:
     if ¬ opened of f then undefined fi;

b) proc ? ensure state = (ref file f, state s) void:
     ¢ after a call of ensure state, the state is as specified
       by s ¢
     begin ensure open(f);
       (char mood of s | set char mood(f) |:
        bin mood of s | set bin mood(f));
       (read mood of s | set read mood | set write mood)(f)
     end;

c) proc ? ensure logical file =
                 (ref file f, state s, bool test) bool:
     ¢ if necessary, a test for the logical file being ended is
       made; the routine returns the value false iff
       test ∧ logical file ended ¢
     begin ensure state(f, s);
       (test | ¬ logical file ended(f) | true)
     end;

d) proc ? ensure page =
                  (ref file f, state s, bool test, int lmarg) bool:
     ¢ if the routine ensure logical file returns false,
       ensure page performs no check and simply returns false;
       otherwise, the page number is checked for being valid
       and the routine returns true ¢
     begin bool result;
       while
         if result:= ensure logical file(f, s, test)
         then
           if int p = p of current pos(f);
             1 - lmarg ≤ p ∧ p ≤ page bound(f)
           then false
           else
             ( ¬ (physical file mended of f)(f)
             | undefined; skip
             | true)
           fi
         else false
         fi
       do skip od;
       result
     end;
```

```
e) proc ? ensure line =
                (ref file f, state s, bool test, int lmarg) bool:
     ¢ if the routine ensure page returns false, ensure line
       performs no check and simply returns false; otherwise,
       the line number is checked for being valid and the
       routine returns true. In this case, a new page is given
       if necessary ¢
     begin bool result;
       while
         if result:= ensure page(f, s, test, 0)
         then
           if int l = l of current pos(f);
             1 - lmarg ≤ l ∧ l ≤ line bound(f)
           then false
           else bool legal = page ended(f);
             if ¬ (page mended of f)(f)
             then
               if legal
               then
                 (ensure page(f, s, test, 1)
                 | next page(f, s));
                 true
               else undefined; skip
               fi
             else true
             fi
           fi
         else false
         fi
       do skip od;
       result
     end;

f) proc ? ensure char =
                (ref file f, state s, bool test, int lmarg) bool:
     ¢ if the routine ensure line returns false, ensure char
       performs no check and simply returns false; otherwise,
       the character number is checked for being valid, where
       in any case a right margin of 1 is allowed, and the
       routine returns true ¢
     begin bool result;
       while
         if result:= ensure line(f, s, test, 0)
         then
           if int c = c of current pos(f);
             1 - lmarg ≤ c ∧ c ≤ char bound(f) + 1
           then false
           else ( ¬ (line mended of f)(f) | undefined; skip | true)
           fi
         else false
         fi
       do skip od;
       result
     end;
```

```
g) proc ? check pos = (ref file f, state s) bool:
     ¢ returns true if the current position is available for
       transput, and false otherwise ¢
     begin bool ended:= true;
       while
         if ¬ ensure char(f, s, read mood of s, 0)
         then (logical file mended of f)(f)
         elif line ended(f)
         then (line mended of f)(f)
         else ended:= false
         fi
       do skip od;
       ¬ ended
     end;

h) proc ? next pos = (ref file f, state s) void:
     ¢ ensures that the current position is available for
       transput ¢
     while
       if ¬ ensure char(f, s, read mood of s, 0)
       then ((logical file mended of f)(f) | true | undefined; skip)
       elif line ended(f)
       then
         ((line mended of f)(f)
         | true
         | (ensure line(f, s, read mood of s, 0) | nextline(f, s));
           true)
       else false
       fi
     do skip od;
```

Layout routines.

```
a) proc new page = (ref file f) void:
     begin state s = state(f);
       ensure page(f, s, read mood of s, 1); next page(f, s)
     end;

b) proc new line = (ref file f) void:
     begin state s = state(f);
       ensure line(f, s, read mood of s, 1); next line(f, s)
     end;

c) proc space = (ref file f) void:
     begin state s = state(f);
       while
         if ¬ ensure char(f, s, read mood of s, 1)
         then ((logical file mended of f)(f) | true | undefined; skip)
         elif line ended(f)
         then
           ((line mended of f)(f)
           | true
           | (ensure line(f, s, read mood of s, 0) | next line(f, s));
             true)
         else false
         fi
       do skip od;
       (logical file ended(f) | outch | skipch)(f)
     end;

d) proc backspace = (ref file f) void:
     begin state s = state(f);
       ensure char(f, s, false, 0); c of current pos(f) -:= 1
     end;

e) proc ? next page = (ref file f, state s) void:
     ¢ the current position is advanced to the next page ¢
     if ref pos cpos = current pos(f), lpos = logical pos(f);
       p of cpos = p of lpos
     then cpos:= lpos;
       if read mood of s
       then ( ¬ (logical file mended of f)(f) | undefined);
         ensure page(f, s, true, 1); next page(f, s)
       else
         if compressible(f)
         then (c of cpos ≠ 1 | next line(f));
           case text of f in
             (text t1): skip  ¢ will never be reached ¢,
             (flextext t2): (int p1 = p of lpos;
                             t2[p1]:= t2[p1][: 1 of lpos - 1])
           esac
         else while ¬ page ended(f) do next line(f, s) od
         fi;
         cpos:= lpos:= (p of cpos + 1, 1, 1)
       fi
     else cpos:= (p of cpos + 1, 1, 1)
     fi;
```

```
f) proc ? next line = (ref file f, state s) void:
     if ref pos cpos = current pos(f), lpos = logical pos(f);
       p of cpos = p of lpos ∧ l of cpos = l of lpos
     then c of cpos:= c of lpos;
       if read mood of s
       then ( ¬ (logical file mended of f)(f) | undefined);
         ensure line(f, s, true, 1); next line(f, s)
       else
         if compressible(f)
         then
           case text of f in
             (text t1): skip  ¢ will never be reached ¢,
             (flextext t2): (int p1 = p of lpos, l1 = l of lpos;
                             t2[p1][l1]:= t2[p1][l1][: c of lpos - 1])
           esac
         else while ¬ line ended(f) do outch(f) od
         fi;
         cpos:= lpos:= (p of cpos, l of cpos + 1, 1)
       fi
     else cpos:= (p of cpos, l of cpos + 1, 1)
     fi;

g) proc ? outch = (ref file f) void:
     if bin mood of f
     then ref pos cpos = current pos(f);
       c of cpos +:= 1; logical pos(f):= cpos
     else put char(f, ".")
     fi;

h) proc ? skipch = (ref file f) void: c of current pos(f) +:= 1;

i) proc set = (ref file f, int p, l, c) void:
     begin ensure open(f);
       if set possible(f)
       then current pos(f):= (p, l, c); check logical pos(f)
       else undefined
       fi
     end;

j) proc set char number = (ref file f, int c) void:
     begin ensure line(f, state(f), false, 0);
       c of current pos(f):= c; check logical pos(f)
     end;

k) proc reset = (ref file f) void:
     (reset possible(f) | set defaults(f) | undefined);
```

Formatless output.

```
a) proc put = (ref file f,
        [] union(outtype, proc(ref file) void) x) void:
     begin state ps = (true, false, false); ensure state(f, ps);
       for i to upb x
       do case x[i] in
            (proc(ref file) void pf):
              (ensure state(f, ps); pf(f)),
            (outtype ot):
              begin [] simplout y = straightout ot;
                ¢ proc L real conv = (L real r) string:
                    float(r, L real width + L exp width + 4,
                          L real width - 1, L exp width + 1) ¢;
                for j to upb y
                do case y[j] in
                     ¢ (union(number, L compl) nc):
                         begin string s =
                           case nc in
                             (L int k): whole (k, L int width + 1),
                             (L real r): L real conv (r),
                             (L compl z): L real conv (re z) + ".i" +
                                          L real conv (im z)
                           esac;
                           while ensure char(f, ps, false, 0);
                             int c = c of current pos(f), n = upb s;
                             if n > char bound(f)
                             then undefined; skip
                             elif c + (c = 1 | n | n + 1) >
                               char bound(f) + 1
                             then
                               ( ¬ (line mended of f)(f) |
                                  put(f, new line));
                               true
                             else false
                             fi
                           do skip od;
                           (c of current pos(f) ≠ 1 | put char(f, "."));
                           for k to upb s
                           do (check pos(f, ps) | put char(f, s[k]) |
                                undefined)
                           od
                         end ¢ ¢ numeric ¢,
                     (bool b):
                       (next pos(f, ps); put char(f, (b | flip | flop))),
                     ¢ (L bits lb):
                         for k to L bits width
                         do next pos(f, ps);
                           put char(f, ((F of lb)[k] | flip | flop))
                         od ¢,
```

```
                    (char k):
                      (next pos(f, ps); put char(f, k)),
                    ([] char ss):
                      if lwb ss > upb ss
                      then ensure char(f, ps, false, 0)
                      else for k from lwb ss to upb ss
                        do next pos(f, ps); put char(f, ss[k]) od
                      fi
                  esac
                od
              end
            esac
          od
        end;
```

b)

```
proc put char = (ref file f, char char) void:
  begin ref pos cpos = current pos(f), lpos = logical pos(f);
    int p = p of cpos, l = l of cpos, ref int c = c of cpos,
    char k, bool found:= false;
    case text of f in
      (text): (k:= char; found:= true),
      (flextext):
        for i to upb F of conv of f while ¬ found
        do struct(char internal, external) key =
            (F of conv of f)[i];
          (internal of key = char |
              k:= external of key; found:= true)
        od
    esac;
    if found
    then (text of f | (text t1): t1[p][l][c],
            (flextext t2): t2[p][l][c]):= k; c +:= 1;
      if cpos beyond lpos
      then lpos:= cpos
      elif ¬ set possible(f) ∧ pos(p of lpos, l of lpos, 1)
            beyond cpos
      then lpos:= cpos;
        (compressible(f) | c ... c)
      fi
    else
      if ¬ (char error mended of f)(f, k:= ".")
      then undefined; k:= "."
      fi;
      (check pos(f, (true, false, false)) |
          put char(f, k) | undefined)
    fi
  end;
```

Formatless input.

```
a) proc get = (ref file f,
               [] union(intype, proc(ref file) void) x) void:
     begin state gs = (true, false, true); ensure state(f, gs);
       for i to upb x
       do case x[i] in
            (proc(ref file) void pf):
              (ensure state(f, gs); pf(f)),
            (intype it):
              begin [] simplin y = straightin it; char k;
                bool k empty;
                op ? = (string s) bool:
                  if ¬ k empty
                  then k empty:= char in string(k, loc int, s)
                  elif ¬ ensure char(f, s, true, 0)
                  then false
                  elif line ended(f)
                  then false
                  else get char(f, k);
                    k empty:= char in string(k, loc int, s)
                  fi;
                op ? = (char c) bool: ? string (c);
                prio ↓ = 8;
                op ↓ = (string s, char c) char:
                  if (k empty | next pos(f, gs); get char(f, k));
                    k empty:= true; char in string(k, loc int, s)
                  then k
                  else char sugg:= c;
                    if (char error mended of f)(f, sugg)
                    then (char in string(sugg, loc int, s) | sugg |
                      undefined; c)
                    else undefined; c
                    fi
                  fi;
                op ↓ = (char s, c) char: string (s) ↓ c;

                proc skip spaces = void:
                  while (k empty | next pos(f, gs)); ? "." do skip od;
                proc read dig = string:
                  (string t:= "0123456789" ↓ "0";
                   while ? "0123456789" do t plusab k od; t);
                proc read sign = char:
                  (skip spaces; char t= ( ? "+-" | k | "+");
                   skip spaces; t);
                proc read num = string:
                  (char t = read sign; t + read dig);
                proc read real = string:
                  (string t:= read sign;
                   ( ? "." | k empty:= false | t plusab read dig);
                   ( ? "." | t plusab "." + read dig);
                   ( ? "₁₀e" | t plusab "₁₀" + read num); t);
```

```
                for j to upb y
                do bool incomp:= false; k empty:= true;
                  case y[j] in
                    ¢ (ref L int ii):
                        incomp:= ¬ string to L int(read num, 10, ii) ¢,
                    ¢ (ref L real rr):
                        incomp:= ¬ string to L real(read real, rr) ¢,
                    ¢ (ref L compl zz):
                        (incomp:= ¬ string to L real(read real, re of zz);
                         skip spaces; "i|"↓"|";
                         incomp:= incomp ∨
                                ¬ string to L real(read real, im of zz)) ¢,
                    (ref bool bb):
                      (skip spaces; bb:= (flip + flop) ↓ flop = flip),
                    ¢ (ref L bits lb):
                        for i to L bits width
                        do skip spaces;
                          (F of lb)[i]:= (flip + flop) ↓ flop = flip
                        od ¢,
                    (ref char cc):
                      (next pos(f, gs); get char(f, cc)),
                    (ref [] char ss):
                      if lwb ss > upb ss
                      then ensure char(f, gs, true, 0)
                      else
                        for i from lwb ss to upb ss
                        do next pos(f, gs); get char(f, ss[i]) od
                      fi,
                    (ref string ss):
                      begin string t:= "";
                        while
                          if check pos(f, gs)
                          then char k; get char(f, k);
                            if char in string(k, loc int, term of f)
                            then k empty:= false
                            else true
                            fi
                          else false
                          fi
                        do t +:= k od;
                        ss:= t
                      end
                  esac;
                  ( ¬ k empty | backspace(f));
                  if incomp
                  then ( ¬ (value error mended of f)(f) | undefined)
                  fi
                od
              end
            esac
          od
        end;
```

```
b) proc ? get char = (ref file f, ref char char) void:
     begin ref pos cpos = current pos(f);
       int p = p of cpos, l = l of cpos, c = c of cpos;
       c of cpos +:= 1;
       char:= case text of f in
         (text t1): t1[p][l][c],
         (flextext t2):
           begin char k:= t2[p][l][c]; bool found:= false;
             for i to upb F of conv of f while ¬ found
             do struct(char internal, external) key =
                  (F of conv of f)[i];
                (external of key = k |
                 k:= internal of key; found:= true)
             od;
             if found
             then k
             else
               if (char error mended of f)(f, k:= ".")
               then k
               else undefined; "."
               fi
             fi
           end
       esac
     end;
```

Binary transput.

```
a) proc put bin = (ref file f, [] outtype ot) void:
     begin state pbs = (false, true, false); ensure state(f, pbs);
       for k to upb ot
       do [] simplout y = straightout ot[k];
         for j to upb y
         do [] char bin = to bin(f, y[j]);
           for i to upb bin
           do nextpos(f, pbs);
             ref pos cpos = cpos of f, lpos = lpos of book of f;
             case text of f in
               (flextext t2):
                 t2[p of cpos][l of cpos][c of cpos]:= bin[i]
             esac;
             c of cpos +:= 1;
             if cpos beyond lpos then lpos:= cpos
             elif ¬ set possible(f)
                  ∧ pos(p of lpos, l of lpos, 1) beyond cpos
             then lpos:= cpos;
               (compressible(f) | c ... c)
             fi
           od
         od
       od
     end;

b) proc get bin = (ref file f, [] intype it) void:
     begin state gbs = (false, true, true); ensure state(f, gbs);
       for k to upb it
       do [] simplin y = straightin it[k];
         for j to upb y
         do
           simplout yj = case y[j] in
             ↓ (ref L int i): i ↑,
             ↓ (ref L real r): r ↑,
             ↓ (ref L compl z): z ↑,
             (ref bool b): b,
             ↓ (ref L bits lb): lb ↑,
             (ref char c): c,
             (ref [] char s): s,
             (ref string ss): ss
           esac;
           [1 : upb(to bin(f, yj))] char bin;
           for i to upb bin
           do next pos(f, gbs); bin[i]:=
             case text of f in
               (flextext t2):
                 t2[p of cpos][l of cpos][c of cpos]
             esac;
             c of cpos +:= 1
           od;
           case ... esac
         od
       od
     end;
```

G. Pre- and postconditions

In the sequel, "(f)", "of f" and "of book of f" have been omitted for brevity's sake.

A precondition of a routine is a condition that must hold when that routine is called in order to warrant satisfactory elaboration. Only such routines are treated here as may be called from within the transput routines. The other routines are not protected by ?, and are therefore available to the user; they have as precondition the trivially satisfied condition, true. However, when any of the routines get possible, etc., is called from within the transput routines, the condition opened is always satisfied, so that implementation by direct inspection of the field concerned is possible.

A postcondition of a routine is a condition that holds when the calling of that routine is completed, where it is assumed that a call of undefined implies non-completion. If a routine "performs no action", then any condition satisfied at the calling still holds at the completion.

A general invariant, that should be understood to be part of any pre- and postcondition, is given by:

```
opened => : ¬ (cpos beyond lpos)
          : write mood ≠ read mood
          Either
            . 1 ≤ p of lpos ≤ page bound
            . 1 ≤ l of lpos ≤ line bound
            . 1 ≤ c of lpos ≤ char bound + 1
          or
            . 1 ≤ p of lpos ≤ page bound
            . l of lpos = line bound + 1
            . c of lpos = 1
          or
            . p of lpos =page bound + 1
            . l of lpos = 1
            . c of lpos = 1
```

with one exception: at the calling of check logical pos, it is possible that cpos beyond lpos.

Position enquiries

routine	precondition	postcondition
d) current pos	. opened	performs no action
e) logical pos	. opened	performs no action
g) state		performs no action
h) page bound	. opened	performs no action
i) line bound	. opened . 1≤p of cpos≤page bound	performs no action
j) char bound	. opened . 1≤p of cpos≤page bound . 1≤l of cpos≤line bound	performs no action
k) check logical pos	. opened	
l) logical file ended	. opened	performs no action, yields true iff cpos coincides with lpos
m) page ended	. opened . 1≤p of cpos≤page bound	performs no action, yields true iff l of cpos=line bound+1
n) line ended	. opened . 1≤p of cpos≤page bound . 1≤l of cpos≤line bound	performs no action, yields true iff c of cpos=char bound+1

Ensure routines

All routines of this section have a trivially satisfied precondition, their only function being to guarantee a certain postcondition.

routine	postcondition
a) ensure open	. opened
b) ensure state(f, s)	. opened . state = s
c) ensure logical file(f, s, test) (In this and the following three routines, Case B of the postcondition can apply only if test is true, i.e., if read mood of s (except when called from backspace).)	. opened . state = s Case A: the yield is true: . test => ¬ logical file ended Case B: the yield is false: . logical file ended
d) ensure page(f, s, test, lmarg)	. opened . state = s Case A: the yield is true: . test => ¬ logical file ended . 1-lmarg≤p of cpos≤page bound Case B: the yield is false: . logical file ended
e) ensure line(f, s, test, lmarg)	. opened . state = s Case A: the yield is true: . test => ¬ logical file ended . 1 ≤ p of cpos ≤ page bound . 1-lmarg≤l of cpos≤line bound Case B: the yield is false: . logical file ended
f) ensure char(f, s, test, lmarg)	. opened . state = s Case A: the yield is true: . test => ¬ logical file ended . 1 ≤ p of cpos ≤ page bound . 1 ≤ l of cpos ≤ line bound . 1-lmarg≤c of cpos≤char bound+1 Case B: the yield is false: . logical file ended

g) check pos(f, s)

. opened
. state = s
Case A: the yield is true:
  . read mood => ¬logical file ended
  . 1 ≤ p of cpos ≤ page bound
  . 1 ≤ l of cpos ≤ line bound
  . 1 ≤ c of cpos ≤ char bound
Case B: the yield is false:
Either
  . read mood ∧ logical file ended
or
  . read mood => ¬logical file ended
  . 1 ≤ p of cpos ≤ page bound
  . 1 ≤ l of cpos ≤ line bound
  . line ended

h) next pos(f, s)

. opened
. state = s
. read mood => ¬logical file ended
. 1 ≤ p of cpos ≤ page bound
. 1 ≤ l of cpos ≤ line bound
. 1 ≤ c of cpos ≤ char bound

Layout routines

routine	precondition	postcondition
d) backspace		. opened . ¬ logical file ended . 1≤p of cpos≤page bound . 1≤l of cpos≤line bound . 0≤c of cpos≤char bound
e) next page(f, s)	. opened . state = s Either . 0≤p of cpos≤page bound or . read mood . p of cpos=p of lpos or both	. opened . state = s . 1≤p of cpos≤page bound+1 . l of cpos = 1 . c of cpos = 1
f) next line(f, s)	. opened . state = s Either . 1≤p of cpos≤page bound . 0≤l of cpos≤line bound or . read mood . p of cpos=p of lpos . l of cpos=l of lpos or both	. opened . state = s . 1≤p of cpos≤page bound . 1≤l of cpos≤line bound+1 . c of cpos = 1
g) outch (It is assumed that "." is convertible.)	. opened . write mood . logical file ended . 1≤p of cpos≤page bound . 1≤l of cpos≤line bound . 1≤c of cpos≤char bound	. opened . write mood . logical file ended . 1≤p of cpos≤page bound . 1≤l of cpos≤line bound . 2≤c of cpos≤char bound+1
h) skipch	. opened . ¬ logical file ended . 1≤p of cpos≤page bound . 1≤l of cpos≤line bound . 0≤c of cpos≤char bound	. opened . 1≤p of cpos≤page bound . 1≤l of cpos≤line bound . 1≤c of cpos≤char bound+1

Formatless output

routine	precondition	postcondition
b) put char(f, s)	. opened . write mood . char mood . 1≤p of cpos≤page bound . 1≤l of cpos≤line bound . 1≤c of cpos≤char bound	. opened . write mood . char mood . 1≤p of cpos≤page bound . 1≤l of cpos≤line bound . 2≤c of cpos≤char bound+1

(Moreover, if c is convertible, then logical file ended before implies logical file ended afterwards.)

d) get char	. opened . read mood . char mood . ¬ logical file ended . 1 ≤ p of cpos ≤ page bound . 1 ≤ l of cpos ≤ line bound . 1 ≤ c of cpos ≤ char bound